Correctness of Communication Protocols

A Case Study

Jørgen F. Søgaard-Andersen

Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby, Denmark

Nancy A. Lynch

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
USA

Butler W. Lampson

Cambridge Research Laboratory
Digital Equipment Corporation
Cambridge, MA 02139
USA

November 1993

Appears as Technical Report MIT/LCS/TR-589, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA, and as Technical Report ID-TR: 1993-129, Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark.

Contents

Chapter 1

Introduction

During the past few years, the technology for formal specification and verification of communication protocols has matured to the point where we believe that it now provides practical assistance for protocol design and validation. Several models for distributed systems in general and communication protocols in particular have been developed, and recent advances include formal models that allow reasoning about untimed systems as well as timed systems, e.g., [AL92a, GSSL93, LV93a, LV93b].

In connection with these models a host of proof techniques have been developed for proving that one protocol implements another. One class of proof techniques is the simulation techniques (including refinement mappings, and forward and backward simulations) [AL91, GSSL93, Jon91, LV92, LV93a, LV93b].

In this work, we show how one approach to formal specification and verification of distributed systems—the live (timed) I/O automata of [GSSL93]—can be used to verify an important class of communication protocols—those for *reliable at-most-once message delivery*.

Thus, the report has two main parts: first, the formal framework of [GSSL93] is presented and augmented with additional theory (including a new temporal logic). Second, we consider the verification example. The purpose of our work is to provide better understanding, documentation and proof for the relaible at-most-once message delivery protocols, and to test the adequacy of the formal framework.

Formal Framework

When formally developing new protocols or proving correctness of existing ones with respect to some specification, a stepwise approach is usually used: the specification is given in a very abstract manner in which abstract data types are used and where possibly no distributed structure is present. In a series of *development steps* this specification is *refined* (or *implemented*) by introducing more low-level data types and by introducing a distributed view of the system, where different nodes (protocol entities) are connected by more or less reliable channels.

By using a formal approach to systems specification, it is possible to prove formally that a low-level (concrete) protocol correctly implements the high-level (abstract) specification. Such a proof is performed by proving that each level in the step-wise development is correct with respect to (i.e., implements) the next more abstract level. This approach to verification implies that the task of proving correctness of a complicated protocol is split into more manageable subtasks, and this greatly reduces the complexity of the overall proof.

The models of [GSSL93] for untimed and timed systems use an automaton (or state machine) to express *safety* properties. A safety property ensures that the system never does anything wrong by specifying the steps the system is *allowed* to perform during execution. However, a safety requirement does not guarantee that the system does anything at all. For that purpose the models of [GSSL93] contain an extra *liveness* condition. The liveness condition restricts the long-term behavior of the system by specifying what *must* eventually happen. An example of a liveness condition is the requirement that each process in a parallel system be given fair chances to proceed. In timed systems it is furthermore possible to specify *timing* requirements like deadlines, response times, etc..

The models of [GSSL93] are entirely *semantic*: they describe an abstract view of how distributed systems behave when executed. Thus, they do not offer any *syntax* for writing down objects of the models. Such a syntax is presented in this work:

- For writing down the automaton part of the models we use a Pascal-like notation which makes our specifications look close to traditional ways of describing protocols for distributed systems.
- The liveness part of the models is specified using the language of an extended *temporal logic* that we develop. This approach has the advantage that parts of the proofs of correctness can be performed using rules of the logic.

An important property of the models of [GSSL93] is that they are *compositional*. This means that each component (e.g., node) in a complex system can be specified separately and that we can implement each component separately and yet obtain an implementation of the entire system. This enables a modular approach to systems specification and verification.

We test the adequacy of the models and proof techniques by formalizing two existing protocols for solving the at-most-once message delivery problem and showing how these protocols can be proved correct.

The At-Most-Once Message Delivery Problem

The *at-most-once message delivery* problem is that of delivering a sequence of messages submitted by a user at one location to a user at another location. Ideally, we would like to insist that all messages be delivered in the order in which they are sent, each exactly once, and that an acknowledgement be returned for each delivered message.[1]

Unfortunately, it is expensive to achieve these goals in the presence of failures (e.g., node crashes). In fact, it is impossible to achieve them at all unless some change is made to the stable state (i.e., the state that survives a crash) each time a message is delivered. To permit less expensive solutions, we weaken the statement of the problem slightly. We allow some messages to be lost when a node crash occurs; however, no messages should otherwise be lost, and those messages that are delivered should not be reordered or duplicated. (The specification is weakened in this way because message loss is generally considered to be less damaging than duplicate delivery.) Now it is required that the user receive either an acknowledgement that the message has been delivered, or in the case of crashes, an indication that the message might have been lost.

There are various ways to solve the at-most-once message delivery problem. All are based on the idea of tagging a message with an identifier and transmitting it repeatedly to overcome the

[1] Our definition of at-most-once message delivery is different from what some people call at-most-once message delivery in that we include acknowledgements and require messages to be delivered in order.

unreliability of the channel. The receiver[2] keeps a stock of "good" identifiers that it has never accepted before; when it sees a message tagged with a good identifier, it accepts it, delivers it, and removes that identifier from the set. Otherwise, the receiver just discards the message, perhaps after acknowledging it. In order for the sender to be sure that its message will be delivered rather than discarded, it must tag the message with a good identifer. What makes the implementations tricky is that the receiver will be keeping track of at least some of its good identifiers in volatile (non-stable) memory, which gets lost in case the receiver node crashes. But the sender does not immediately learn about the crash, so it may go on using these identifers and thus transmit messages that the receiver will reject. Different protocols use different methods to keep the sender and the receiver more or less in agreement about what identifiers to use.

A desirable property, which is not directly related to correctness, is that the implementations offer a way of cleaning up "old" information when this cannot affect the future behavior.

In this work, we consider two protocols that are important in practice: the Clock-Based Protocol (which we call C) of Liskov, Shrira and Wroclawski [LSW91] and the Five-Packet Handshake Protocol (which we call H) of Belsnes [Bel76]. The latter is the standard protocol for setting up network connections, used in TCP, ISO TP-4, and many other transport protocols. It is sometimes called the three-way handshake, because only three packets are needed for message delivery; the additional packets are required for acknowledgement and cleaning up the state. The former protocol was developed as an example to show the usefulness of clocks in network protocols [Lis91] and has been implemented at M.I.T.. Both protocols are sufficiently complicated that formal specification and proof seem useful.

Survey of the Example

We express both protocols, H and C, as well as the formal specification S of the at-most-once message delivery problem, in terms of the models of [GSSL93].

Although the two protocols appear to be quite different, we have found that both can be expressed formally as implementations of a common *Generic Protocol* G, which, in turn, is an implementation of the problem specification. To prove that G implements the specification, for proof-technical reasons we introduce an additional level of abstraction, the *Delayed-Decision Specification D*. This is depicted in Figure 1.1. Introducing intermediate levels of abstraction, like G and D, is a general proof strategy that allows large, complicated proofs to be split into smaller and more managerable subproofs.

The specification S is stated in the *untimed* model of [GSSL93] whereas the Clock-Based Protocol C uses the timed model. This apparent model inconsistency is resolved by considering S to be a timed system that does not put any constraints in real time. In [GSSL93] certain *embedding results* provide the formal basis for moving between the timed and untimed model.

In this report we provide almost complete proofs of correctness. Some parts of the proofs are omitted however but we treat all different kinds of proofs and provide informal justification for the missing parts.

Outline of the Report

The report is structured as follows. In Part I we consider the formal framework: Chapter 2 gives a brief introduction to the models of [GSSL93] and the embedding results. Chapters 3 and

[2] We denote by "receiver" the protocol entity that is situated on the receiver node, and use phrases like "the user at the receiver end" to denote the user that communicates with the receiver. Correspondingly for "sender".

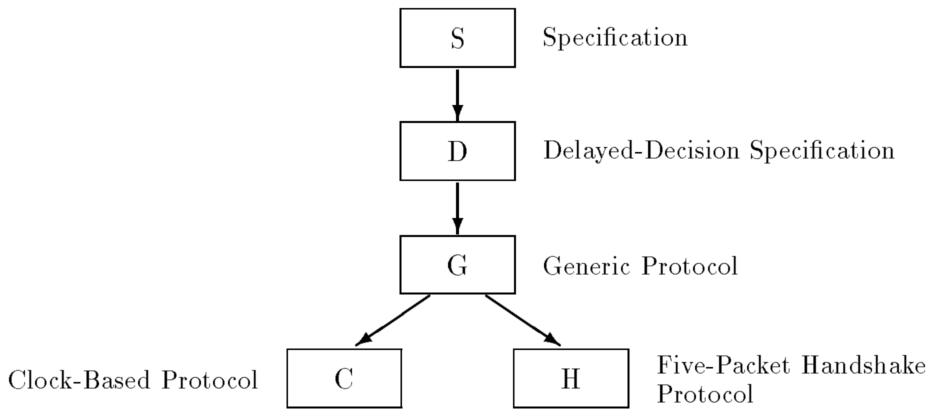


Figure 1.1

Overview of the levels of abstraction.

4 describe the syntax we use for specifying systems: first, in Chapter 3, we define an extended temporal logic, and then, in Chapter 4, we specifically show how this temporal logic is used to specify liveness conditions. Chapter 5 describes the proof techniques we use when proving correctness of the protocols. These techniques are mainly taken from [GSSL93].

The remaining part of the report Part II deals with the at-most-once message delivery example. First, in Chapter 6, we present the formal specification S of the at-most-once message delivery problem. In Chapter 7 we present the Delayed-Decision Specification D and show that it correctly implements S. Chapters 8—10 then formally specify the G, H, and C levels and consider their correctness.

Finally, in Chapter 11, we give concluding remarks.

The report contains three appendices. Appendix A introduces some basic notation and should be read before the rest of the report. Appendix B and Appendix C contain proofs of certain results in the main parts of the report.

Acknowledgements

We thank Hans Henrik Løvengreen for his valuable criticism and useful comments on this report, and for his contribution to the definition of the temporal logic developed in this report.

This work is supported in part at the Technical University of Denmark by the Danish Research Academy and the Danish Technical Research Council. Supported at MIT by NSF grants CCR-89-15206 and 9225124-CCR, by DARPA contracts N00014-89-J-1988 and N00014-92-J-4033, and by ONR contract N00014-91-J-1046.

Part I

The Formal Framework

Chapter 2

The Model

To make this report self-contained, we give a brief presentation of the operational models for distributed systems that are developed in [GSSL93]. We give all formal definitions and results that are needed but refer to [GSSL93] for details about, e.g., proofs and for a more thorough treatment of the models.

We first present the model for untimed systems. Then the model for timed systems is presented, and finally we show how an untimed system can be thought of as a timed system that allows time to pass arbitrarily.

2.1 The Model for Untimed Systems

The model for untimed systems, called *live I/O automata*, which is developed in [GSSL93] consists of an automaton part (or state machine), with a labeled transition relation, and a liveness condition. The automaton specifies the possible steps of the system, i.e., it specifies what is *allowed* to happen, thus, the safety of the system. The liveness condition restricts the long-term behavior of the system by specifying what *must* eventually happen.

The liveness condition can be seen as a way of restricting the way the automaton is "executed" whenever it is working properly. A liveness condition for a system of two parallel processes might require that each component be given the possibility of making progress infinitely often. In this way executions where one component wishes to proceed but is never given a chance are ruled out. This kind of liveness is known as weak fairness and is implemented on a physical machine by executing the parallel processes on separate processors or by using a fair scheduler. In the examples in this work we will see examples of more complicated liveness requirements.

As mentioned above the automaton part has a labeled transition relation. This means that each step of the automaton is labeled by a name, called an *action*. The set of actions are partitioned into *external* and *internal* actions, where only the external actions are visible from the environment. The model is *event*-based in the sense that communication between parallel components of a system or between system and environment is modeled by joint actions. That is, communication is modeled as the joint executions of steps labeled by the same action. Thus, the states cannot be observed. For this reason correctness is based on the sequences of external actions (called *traces*) that can occur when the system is working properly, i.e., when its liveness condition is satisfied.

To express a notion of system vs. environment, the external actions are partitioned into input and output actions, i.e., an I/O distinction is introduced. Intuitively output (and internal) actions are controlled by the system, and are thus called *locally-controlled* actions, whereas input actions are controlled by the environment of the system. Since a system cannot control its environment, live I/O automata are required to be *environment-free* which intuitively means that no matter which inputs the environment provides during execution, the system can perform locally-controlled actions and in this way satisfy its liveness condition. Thus, the environment-freedom requirement ensures that live I/O automaton do not have liveness conditions like: "sooner or later input a arrives".

The environment-freedom requirement also implies that the automaton part of a live I/O automaton must be *input-enabled* which means that the automaton should be able to receive any input in any state.

Even though our live I/O automaton model is not as general as a model without I/O distinction and the environment-freedom requirement, a large number of systems can be specified using this model. In particular many distributed systems have a clear distinction between the output from the system and the input from the environment, and furthermore such systems are usually designed to be able to receive input at any time since processes are usually connected by networks that are not capable of buffering messages. In [GSSL93] a technical justification of environment-freedom is offered. This justification deals with the fact that without I/O distinction and environment-freedon, a trace-based correctness notion as the one mentioned above is not adequate in that it cannot form the base of a notion of implementation that corresponds to our intuition. Furthermore, there exists simpler proof techniques for live I/O automata than for more general models.

We first present the automaton part, called *safe I/O automata*. Then we add the liveness condition, discuss the notion of implementation, and state an important *substitutivity* property of the model.

2.1.1 Safe I/O Automata

Definition 2.1 (Safe I/O Automaton)

A *safe I/O automaton* A consists of four components:

- A set $states(A)$ of states.
- A nonempty set $start(A)$ of start states ($start(A) \subseteq states(A)$).
- An action signature $sig(A) = (in(A), out(A), int(A))$ of disjoint sets of input, output, and internal actions, respectively. Denote by $ext(A)$ the set $in(A) \cup out(A)$ of external actions, by $local(A)$ the set $out(A) \cup int(A)$ of locally-controlled actions, and by $acts(A)$ the set $ext(A) \cup int(A)$ of actions.
- A transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$. The transition relation $steps(A)$ must have the property that for each state $s \in states(A)$ and each input action $a \in in(A)$ there exists a state $s' \in states(A)$ such that $(s, a, s') \in steps(A)$. A is said to be *input-enabled.*

■

An action a is *enabled* in a state s if there exists a state s' such that (s, a, s') is a step, i.e., $(s, a, s') \in steps(A)$. A set $\mathcal{A}$ of actions is said to be enabled in state s if there exists an action

$a \in \mathcal{A}$ such that a is enabled in s. An action or set of actions which is not enabled in a state s is said to be disabled in s.

An *execution fragment* α of a safe I/O automaton A is a (finite or infinite) sequence of alternating states and actions starting with a state and, if the execution fragment is finite, ending in a state

$$\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$$

where each $(s_i, a_{i+1}, s_{i+1}) \in \mathit{steps}(A)$. Denote by $\mathit{fstate}(\alpha)$ the first state of α and, if α is finite, denote by $\mathit{lstate}(\alpha)$ the last state of α. Furthermore, denote by $\mathit{frag}^*(A)$, $\mathit{frag}^\omega(A)$, and $\mathit{frag}(A)$ the sets of finite, infinite and all execution fragments of A, respectively. An *execution* is an execution fragment whose first state is a start state. Denote by $\mathit{exec}^*(A)$, $\mathit{exec}^\omega(A)$ and $\mathit{exec}(A)$ the sets of finite, infinite and all execution of A, respectively. A state s of A is *reachable* if there exists a finite execution of A that ends in s.

A finite execution fragment $\alpha_1 = s_0 a_1 s_1 \cdots a_n s_n$ of A and an execution fragment $\alpha_2 = s_n a_{n+1} s_{n+1} \cdots$ of A can be *concatenated*. In this case the concatenation, written $\alpha_1 \frown \alpha_2$, is the execution fragment $s_0 a_1 s_1 \cdots a_n s_n a_{n+1} s_{n+1} \cdots$. Clearly, $\alpha_1 \frown \alpha_2$ is an execution iff α_1 is an execution.

An execution fragment α_1 of A is a *prefix* of an execution fragment α_2 of A, written $\alpha_1 \leq \alpha_2$, if either $\alpha_1 = \alpha_2$ or α_1 is finite and there exists an execution fragment α_1' of A such that $\alpha_2 = \alpha_1 \frown \alpha_1'$.

Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be an execution fragment. The *length* of α is the number of actions occurring in α. Thus,

$$|\alpha| \triangleq \begin{cases} n & \text{if } \alpha \text{ is finite and ends in } s_n \\ \infty & \text{if } \alpha \text{ is infinite} \end{cases}$$

Define the ith *prefix* and ith *suffix* of α, for $0 \leq i \leq |\alpha|$[1], as

$$\begin{aligned} \alpha|_i &\triangleq s_0 a_1 s_1 \cdots a_i s_i \\ {}_i|\alpha &\triangleq \begin{cases} s_i a_{i+1} s_{i+1} \cdots & \text{if } i < |\alpha| \\ s_{|\alpha|} & \text{if } \alpha \text{ is finite and } i = |\alpha| \end{cases} \end{aligned}$$

The *trace* of an execution fragment α of A, written $\mathit{trace}_A(\alpha)$, or just $\mathit{trace}(\alpha)$ when A is clear, is the list obtained by restricting α to the set of external actions of A, i.e., $\mathit{trace}(\alpha) = \alpha \upharpoonright \mathit{ext}(A)$. For a set E of executions of A, denote by $\mathit{traces}_A(E)$, or just $\mathit{traces}(E)$ when A is clear from context, the set of traces of the executions in E. We say that β is a trace of A if there exists an execution α of A with $\mathit{trace}(\alpha) = \beta$. Denote by $\mathit{traces}^*(A)$, $\mathit{traces}^\omega(A)$ and $\mathit{traces}(A)$ the sets of finite, infinite and all traces of A, respectively. Note, that a finite trace might be the trace of an infinite execution. Furthermore, for any list l of actions of A, define $\mathit{trace}_A(l)$, or just $\mathit{trace}(l)$ when A is clear from context, to be $l \upharpoonright \mathit{ext}(A)$.

When specifying complex distributed systems, it is important to be able to specify each process separately and then obtain the specification of the entire system as the *parallel composition* of the specifications of the processes. This modular approach greatly reduces the complexity of specifying large systems. The parallel composition operator in this model uses a synchronization style where automata synchronize on their common actions and evolve independently on the others. It is required that each external action be under the control of at most one automaton,

[1] The index i ranges over the natural numbers so if $|\alpha| = \infty$, then $i \leq |\alpha|$ is the same as $i < |\alpha|$.

thus, parallel composition is defined only for *compatible* safe I/O automata. Compatibility requires that each action be an output action of at most one safe I/O automaton. Furthermore, to avoid action name clashes, compatibility requires that internal action names be unique.

Definition 2.2 (Parallel composition of safe I/O automata)

Safe I/O automata $A_1, \ldots, A_N$ are *compatible* if for all $1 \leq i, j \leq N$ with $i \neq j$

1. $out(A_i) \cap out(A_j) = \emptyset$
2. $int(A_i) \cap acts(A_j) = \emptyset$

The *parallel composition* $A_1 \parallel \cdots \parallel A_N$ of compatible safe I/O automata $A_1, \ldots, A_N$ is the safe I/O automaton A such that

1. $states(A) = states(A_1) \times \cdots \times states(A_N)$
2. $start(A) = start(A_1) \times \cdots \times start(A_N)$
3. $out(A) = out(A_1) \cup \cdots \cup out(A_N)$
4. $in(A) = (in(A_1) \cup \cdots \cup in(A_N)) \setminus out(A)$
5. $int(A) = int(A_1) \cup \cdots \cup int(A_N)$
6. $((s_1, \ldots, s_N), a, (s'_1, \ldots, s'_N)) \in steps(A)$ iff for all $1 \leq i \leq N$
   (a) if $a \in acts(A_i)$ then $(s_i, a, s'_i) \in steps(A_i)$
   (b) if $a \notin acts(A_i)$ then $s_i = s'_i$

■

The executions of the parallel composition of compatible safe I/O automata $A = A_1 \| \ldots \| A_n$ can be *projected* to the component automata. First, for any state s of A, denote by $s \lceil A_i$ the state of A_i obtained by projecting s to A_i. Then, for any execution α of A denote by $\alpha \lceil A_i$ the execution of A_i obtained from α by projecting the states in α to A_i and by removing each action not in $acts(A_i)$ together with the state preceding the action.

Parallel composition is typically used to build complex systems based on simpler components. Some actions are meant to represent internal communications between the subcomponents of the complex system. The *action hiding* operator allows us to change some external actions into internal ones.

Definition 2.3 (Action hiding)

Let A be a safe I/O automaton and let $\mathcal{A}$ be a set of actions such that $\mathcal{A} \subseteq local(A)$. Then define $A \setminus \mathcal{A}$ to be the safe I/O automaton such that

1. $states(A \setminus \mathcal{A}) = states(A)$
2. $start(A \setminus \mathcal{A}) = start(A)$
3. $in(A \setminus \mathcal{A}) = in(A)$

4. $out(A \setminus \mathcal{A}) = out(A) \setminus \mathcal{A}$

5. $int(A \setminus \mathcal{A}) = int(A) \cup \mathcal{A}$

6. $steps(A \setminus \mathcal{A}) = steps(A)$

■

The final operator on safe I/O automata is *action renaming*. Several processes might be identical except for their actions' names. A classical example is given by the processes of a token ring communication network. Such processes could be easily specified by first defining a generic process and then creating an instance for each process through renaming of the actions. Action renaming can also be used to resolve name clashes that lead to incompatibilities in Definition 2.2.

Definition 2.4 (Action renaming)
A mapping ρ from actions to actions is *applicable* to a safe I/O automaton A if it is injective and $acts(A) \subseteq dom(\rho)$. Given a safe I/O automaton and a mapping ρ applicable to A, we define $\rho(A)$ to be the safe I/O automaton such that

1. $states(\rho(A)) = states(A)$

2. $start(\rho(A)) = start(A)$

3. $in(\rho(A)) = \rho(in(A))$

4. $out(\rho(A)) = \rho(out(A))$

5. $int(\rho(A)) = \rho(int(A))$

6. $steps(\rho(A)) = \{(s, \rho(a), s') \mid (s, a, s') \in steps(A)\}$

■

2.1.2 Live I/O Automata

We have now described the safety component of a live I/O automaton. The liveness condition should specify which executions of a safe I/O automaton are considered to represent a properly working system. For this reason a liveness condition, in this model, is a subset of the executions of the safe I/O automaton. However, a liveness condition is used to restrict the long-term behavior of a system, i.e., to specify what must happen sooner or later. Thus, *any* finite execution of the safe I/O automaton should have an extension in the liveness condition. In other words, no matter what the safe I/O automaton has done up to some time, there is still a way for it to behave properly according to the liveness condition.

This definition of a liveness condition only ensures that the liveness condition does not introduce more safety than is already specified by the safe I/O automaton. It does not, however, capture the fact that a live I/O automaton must not constrain its environment. To express this idea (the *environment-freedom* condition) formally, we set up a *game* between the system and its environment, and the system is then environment-free if it can win the game no matter what moves the environment performs, i.e., if the system has a winning strategy. The environment moves by providing any finite number of input actions, and the system moves by performing a local step, i.e., a step labeled by a locally-controlled action, or by making no step (a $\perp$ move).

The fact that the environment is allowed to provide any finite number of input actions at any move expresses that the environment can be arbitrarily but not infinitely fast compared to the system. Note also that the environment provides *actions* and not steps. This is because the environment has no control over the state of the system: the environment provides the action and the system decides which of the possible states it should reach in response.

The behavior of the system during the game is determined by a *strategy*. A strategy is a pair (g, f) of functions, where g determines which state to reach in response to an input action, and f determines the moves of the system. The notion of strategy is formalized as follows.

Definition 2.5 (Strategy)

Consider any safe I/O automaton A. A *strategy* defined on A is a pair of functions (g, f) where $g : exec^*(A) \times in(A) \to states(A)$ and $f : exec^*(A) \to (local(A) \times states(A)) \cup \{\perp\}$ such that

1. $g(\alpha, a) = s$ implies $(lstate(\alpha), a, s) \in steps(A)$
2. $f(\alpha) = (a, s)$ implies $(lstate(\alpha), a, s) \in steps(A)$

■

The moves of the environment during the game are represented as an infinite sequence $\mathcal{I}$, called an *environment sequence*, of input actions interleaved with infinitely many λ symbols. The symbol λ is used to represent the points at which the system is allowed to move. The occurrence of infinitely many λ symbols in an environment sequence guarantees that each environment move consists of only finitely many input actions.

Remember from the discussion above that after any finite execution the system should still have a way of behaving properly. This is reflected in the following definition of the *outcome* of a strategy.

Definition 2.6 (Outcome of a strategy)

Let A be a safe I/O automaton and (g, f) a strategy defined on A. Define an *environment sequence* for A to be any infinite sequence of symbols from $in(A) \cup \{\lambda\}$ with infinitely many occurrences of λ. Then define $R_{(g,f)}$, the *next-function induced by* (g, f), as follows: for any finite execution α of A and any environment sequence $\mathcal{I}$ for A,

$$R_{(g,f)}(\alpha, \mathcal{I}) = \begin{cases} (\alpha a s, \mathcal{I}') & \text{if } \mathcal{I} = \lambda\mathcal{I}',\ f(\alpha) = (a, s) \\ (\alpha, \mathcal{I}') & \text{if } \mathcal{I} = \lambda\mathcal{I}',\ f(\alpha) = \perp \\ (\alpha a s, \mathcal{I}') & \text{if } \mathcal{I} = a\mathcal{I}',\ g(\alpha, a) = s \end{cases}$$

Let α be any finite execution of A and $\mathcal{I}$ any environment sequence for A. The *outcome sequence of* (g, f) *given* α *and* $\mathcal{I}$ is the unique infinite sequence $(\alpha^n, \mathcal{I}^n)_{n \geq 0}$ that satisfies:

- $(\alpha^0, \mathcal{I}^0) = (\alpha, \mathcal{I})$ and
- For all $n > 0$, $(\alpha^n, \mathcal{I}^n) = R_{(g,f)}(\alpha^{n-1}, \mathcal{I}^{n-1})$.

Note, that $(\alpha^n)_{n\geq 0}$ forms a chain ordered by *prefix*.

The *outcome* $\mathcal{O}_{(g,f)}(\alpha,\mathcal{I})$ of the strategy (g,f) given α and $\mathcal{I}$ is the execution $\lim_{n\rightarrow\infty}\alpha^n$, where $(\alpha^n,\mathcal{I}^n)_{n\geq 0}$ is the outcome sequence of (g,f) given α and $\mathcal{I}$ and the limit is taken under prefix ordering.

■

It is easy to see that any outcome of a strategy is an execution of the safe I/O automaton. The concepts of strategies and outcomes are used to define formally the *environment-freedom*-property.

Definition 2.7 (Environment-freedom)

A pair (A,L), where A is a safe I/O automaton and $L\subseteq \mathit{exec}(A)$, is *environment-free* if there exists a strategy (g,f) defined on A such that for any finite execution α of A and any environment sequence $\mathcal{I}$ for A, the outcome $\mathcal{O}_{(g,f)}(\alpha,\mathcal{I})$ is an element of L. The strategy (g,f) is called an *environment-free strategy* for (A,L).

■

Clearly, if a pair (A,L) is environment-free, then any finite execution of A has an extention in L. Finally we can present the notion of *live I/O automaton*.

Definition 2.8 (Live I/O automata)

A *live I/O automaton* is a pair (A,L) where A is a safe I/O automaton and $L\subseteq \mathit{exec}(A)$ such that (A,L) is environment-free. We refer to the executions in L as the *live executions* of (A,L). Similarly the traces in *traces*(L) are referred to as the *live traces* of (A,L).

■

In Chapter 4 we will define some standard liveness conditions, like weak fairness, for safe I/O automata and show once and for all that the resulting pairs are environment-free.

The operators on safe I/O automata can now be extended to live I/O automata. For parallel composition the liveness condition for a composed system consists of all those executions whose projection to the components yield live executions of the components. That corresponds to the intuitive idea that a composed system works properly if all components work properly.

Definition 2.9 (Parallel composition of live I/O automata)

Live I/O automata $(A_1,L_1),\ldots,(A_N,L_N)$ are *compatible* if the safe I/O automata $A_1,\ldots,A_N$ are compatible.

The *parallel composition* $(A_1,L_1)\parallel\cdots\parallel(A_N,L_N)$ of compatible live I/O automata $(A_1,L_1),\ldots,(A_N,L_N)$ is defined to be the pair (A,L) where $A=A_1\parallel\cdots\parallel A_N$ and $L=\{\alpha\in \mathit{exec}(A)\mid \alpha\lceil A_1\in L_1,\ldots,\alpha\lceil A_N\in L_N\}$.

■

Definition 2.10 (Action hiding of live I/O automata)

Let (A, L) be a live I/O automaton and let $\mathcal{A}$ be a set of actions such that $\mathcal{A} \subseteq \mathit{local}(A)$. Then define $(A, L) \setminus \mathcal{A}$ to be the pair $(A \setminus \mathcal{A}, L)$.

■

Definition 2.11 (Action renaming of live I/O automata)

A mapping ρ from actions to actions is *applicable* to a live I/O automaton (A, L) if it is applicable to A. Let α be any execution of A. Define $\rho(\alpha)$ to be the sequence that results from replacing each occurrence of every action a in α by $\rho(a)$. Given a live I/O automaton (A, L) and a mapping ρ applicable to (A, L), we define $\rho((A, L))$ to be the pair $(\rho(A), \rho(L))$.[2]

■

An important property of the operators is that they are closed for live I/O automata in the sense that they produce new live I/O automata.

Proposition 2.12 (Closure of parallel composition)

Let $(A_1, L_1), \ldots, (A_N, L_N)$ be compatible live I/O automata. Then $(A_1, L_1) \parallel \cdots \parallel (A_N, L_N)$ is a live I/O automaton.

■

Proposition 2.13 (Closure of action hiding)

Let (A, L) be a live I/O automaton and let $\mathcal{A} \subseteq \mathit{local}(A)$. Then $(A, L) \setminus \mathcal{A}$ is a live I/O automaton.

■

Proposition 2.14 (Closure of action renaming)

Let (A, L) be a live I/O automaton and let ρ be a mapping applicable to (A, L). Then $\rho((A, L))$ is a live I/O automaton.

■

2.1.3 Correctness

The notion of correct implementation between live I/O automata is based on their live traces. A live I/O automaton (A, L) is said to correctly implement a live I/O automaton (B, M), with the same input and output actions, if all live traces of (A, L) are also live traces of (B, M). This correctness notion ensures that whatever (A, L) does, (B, M) could have done the same. That is, (A, L) does nothing wrong which in other words means that (A, L) satisfies the safety specified by (B, M). Furthermore, the correctness notion also guarantees that (A, L) in fact does something because the correctnotion is based on *live* traces, i.e., traces where something "good" happens.

Sometimes one is not interested in the liveness of a system and therefore specifies a system as a safe I/O automaton. One safe I/O automaton is said to *safely implement* a safe I/O

[2] As notational convention we allow a function to be applied to *subsets* of elements from the domain of the function. The result is then the set obtained by applying the function to each element of the subset. Thus, $\rho(L) = \{\rho(\Sigma) \mid \Sigma \in L\}$.

automaton B, with the same input and output actions, if all traces of A are also traces of B. This notion of safe implementation does not guarantee that A does anything at all. In fact, a safe I/O automaton A with one state, no local steps, and "self-loop" steps for each of its input actions, is a safe implementation of any safe I/O automaton with the same input and output actions. The notion of safe implementation trivially extends to live I/O automata.

Definition 2.15 (Implementation relations)

Given two live I/O automata (A, L) and (B, M) such that $in(A) = in(B)$ and $out(A) = out(B)$, define the following implementation relations:

Safe:	$A \sqsubseteq_{\mathrm{S}} B$	iff	$traces(A) \subseteq traces(B)$
Safe:	$(A, L) \sqsubseteq_{\mathrm{S}} (B, M)$	iff	$A \sqsubseteq_{\mathrm{S}} B$
Correct:	$(A, L) \sqsubseteq_{\mathrm{L}} (B, M)$	iff	$traces(L) \subseteq traces(M)$

■

The symbol $\sqsubseteq_{\mathrm{S}}$ indicates that this relation is based on Safe traces. Similarly $\sqsubseteq_{\mathrm{L}}$ is based on Live traces. All implementation relations are clearly preorders.

2.1.4 Substitutivity

An important property of the model is that it allows a modular approach to systems specification and verification. If, for instance, a system S is made up of several parallel components, it is possible to implement separately each component of S and yet obtain an implementation of S. This is usually referred to as the substitutivity of the implementation relations with respect to the parallel composition operator. Similar results exist for the other two operators as stated in the following proposition.

Proposition 2.16 (Substitutivity)

Let $(A_i, L_i), (B_i, M_i)$, $i = 1, \ldots, N$, be live I/O automata with $in(A_i) = in(B_i)$ and $out(A_i) = out(B_i)$, and let $\sqsubseteq_X$ be one relation among $\sqsubseteq_{\mathrm{S}}$ and $\sqsubseteq_{\mathrm{L}}$. If, for each i, $(A_i, L_i) \sqsubseteq_X (B_i, M_i)$, then

1. *if $(A_1, L_1), \ldots, (A_N, L_N)$ are compatible and $(B_1, M_1), \ldots, (B_N, M_N)$ are compatible then* $(A_1, L_1) \| \cdots \| (A_N, L_N) \sqsubseteq_X (B_1, M_1) \| \cdots \| (B_N, M_N)$.
2. *if $\mathcal{A} \subseteq local(A_1)$ and $\mathcal{A} \subseteq local(B_1)$ then* $(A_1, L_1) \setminus \mathcal{A} \sqsubseteq_X (B_1, M_1) \setminus \mathcal{A}$
3. *if ρ is a mapping applicable to both A_1 and B_1 then* $\rho((A_1, L_1)) \sqsubseteq_X \rho((B_1, M_1))$

■

Note, in Part 1 of the proposition, that even though $(A_1, L_1), \ldots, (A_N, L_N)$ are compatible, then the specifications $(B_1, M_1), \ldots, (B_N, M_N)$ are not compatible if they contain internal actions that collide with already existing actions of other components. Thus, we must require that also $(B_1, M_1), \ldots, (B_N, M_N)$ be compatible. However, in practice the problem is usually solved by choosing brand new names for new internal actions in an implementation. Similar considerations apply to Parts 2 and 3.

2.2 The Model for Timed Systems

The timed model, called *live timed I/O automata*, is very similar to the untimed model in that it consists of an automaton part (*safe timed I/O automaton*) and a liveness condition. Each state of the safe timed I/O automaton has an associated time, returned by the mapping $.now$, and a certain *time-passage* action ν representing the passage of time. The steps of a safe timed I/O automaton are restricted such that time-passage steps must increase time and all other steps must not change time. Thus, all other steps than time-passage steps are thought of as occurring instantaneously. There are a few other restrictions representing natural properties of time.

2.2.1 Safe Timed I/O Automata

Times are specified using a *dense* time domain $\mathsf{T} = \mathsf{R}^{\geq 0}$, i.e., the set of non-negative reals.

Definition 2.17 (Safe timed I/O automata)

A *safe timed I/O automaton* A consists of five components

- A set $states(A)$ of states.
- A nonempty set $start(A)$ of start states ($start(A) \subseteq states(A)$).
- A mapping $.now_A : states(A) \to \mathsf{T}$ (called $.now$ when A is clear from context), indicating the current time in a given state.
- An action signature $sig(A) = (in(A), out(A), int(A))$ of disjoint sets of input, output, and internal actions, respectively. Denote by $ext(A)$ the set $in(A) \cup out(A) \cup \{\nu\}$ of external actions, where ν is a special *time-passage* action, by $vis(A)$ the set $in(A) \cup out(A)$ of visible actions, by $local(A)$ the set $out(A) \cup int(A)$ of locally-controlled actions, and by $acts(A)$ the set $ext(A) \cup int(A)$ of actions.
- A transition relation $steps(A) \subseteq states(A) \times acts(A) \times states(A)$.

A must be input-enabled and satisfy the following five axioms

S1 If $s \in start(A)$ then $s.now = 0$.

S2 If $(s, a, s') \in steps(A)$ and $a \neq \nu$, then $s'.now = s.now$.

S3 If $(s, \nu, s') \in steps(A)$ then $s'.now > s.now$.

S4 If $(s, \nu, s') \in steps(A)$ and $(s', \nu, s'') \in steps(A)$, then $(s, \nu, s'') \in steps(A)$.

To be able to state the last axiom, the following auxiliary definition is needed. Let I be an interval of T. Then a function $\omega : I \to states(A)$ is an *A-trajectory*, sometimes called *trajectory* when A is clear from context, if

1. $\omega(t).now = t$ for all $t \in I$, and
2. $(\omega(t), \nu, \omega(t')) \in steps(A)$ for all $t, t' \in I$ with $t < t'$.

That is, ω assigns to each time t in the interval I a state having the given time t as its *now* component. The assignment is done in such a way that time-passage steps can span between any pair of states in the range of ω. Denote $inf(I)$ and $sup(I)$ by $ftime(\omega)$ and $ltime(\omega)$, respectively. If I is left closed, then denote $\omega(ftime(\omega))$ by $fstate(\omega)$. Similarly, if I is right closed, then denote $\omega(ltime(\omega))$ by $lstate(\omega)$. If I is closed, then ω is said to be an A-trajectory from $fstate(\omega)$ to $lstate(\omega)$. An A-trajectory ω whose domain $dom(\omega)$ is a singleton set $[t, t]$ is also denoted by the set $\{\omega(t)\}$.

The final axiom then becomes

S5 If $(s, \nu, s') \in steps(A)$ then there exists an A-trajectory from s to s'.

■

Axiom **S1** states that time must be 0 in any start state. Axiom **S2** says that non-time-passage steps occur instantaneously, at a single point in time. In this framework, operations with some duration in time are modeled by a start action and an end action. Axiom **S3** says that time-passage steps cause time to increase. Axiom **S4** gives a natural property of time, namely that if time can pass in two steps, then it can also pass in a single step. Finally, Axiom **S5** says that if time can pass from time t to time t', then it is possible to associate states with all times in the interval in a consistent way. This axiom opens the possibility of specifying *hybrid* systems, i.e., systems where the state can change coutinuously when time passes. However, in the systems we will look at in this work the states consists of a "basic" state and a *now* variable, and the basic state does not change during time-passage.

2.2.1.1 Timed Executions

The notions of executions and traces and operations on these carry over from the untimed setting. However, executions do not adequately capture the behavior of a system since they do not tell us what states the system goes through during time-passage. For this reason a notion of *timed executions* is introduced.

A *timed execution fragment* Σ of a safe timed I/O automaton A is a (finite or infinite) sequence of alternating A-trajectories and actions in $vis(A) \cup int(A)$, starting in a trajectory and, if the sequence is finite, ending in a trajectory

$$\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$$

such that the following holds for each index i:

1. If ω_i is not the last trajectory in Σ, then its domain is a closed interval. If ω_i is the last trajectory of Σ (when Σ is a finite sequence), then its domain is a left-closed interval (and either open or closed to the right).

2. If ω_i is not the last trajectory of Σ, then $(lstate(\omega_i), a_{i+1}, fstate(\omega_{i+1})) \in steps(A)$.

A *timed execution* is a timed execution fragment $\omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ for which $fstate(\omega_0)$ is a start state.

If Σ is a timed execution fragment, then define $ftime(\Sigma)$ and $fstate(\Sigma)$ to be $ftime(\omega_0)$ and $fstate(\omega_0)$, respectively, where ω_0 is the first trajectory of Σ. Also, define $ltime(\Sigma)$ to be the supremum of the union of the domains of the trajectories of Σ. Finally, if Σ is a finite sequence where the domain of the last trajectory ω is a closed interval, define $lstate(\Sigma)$ to be $lstate(\omega)$.

2.2.1.2 Finite, Admissible, and Zeno Timed Executions

The timed executions and timed execution fragments of a safe timed I/O automaton can be partitioned into *finite*, *admissible*, and *Zeno* timed executions and timed execution fragments.

A timed execution (fragment) Σ is defined to be *finite*, if it is a finite sequence and the domain of the last trajectory is closed. A timed execution (fragment) Σ is *admissible* if $ltime(\Sigma) = \infty$. Finally, a timed execution (fragment) Σ is *Zeno* if it is neither finite nor admissible.

There are basically two types of Zeno timed executions: those containing infinitely many occurrences of non-time-passing actions but for which there is a finite upper bound on the times in the domains of the trajectories, and those containing finitely many occurrences of non-time-passing actions and for which the domain of the last state set is right-open. Thus, Zeno timed executions represent executions of a safe timed I/O automaton where an infinite amount of activity occurs in a bounded period of time. (For the second type of Zeno timed executions, the infinitely many time-passage steps needed to span the right-open interval should be thought of a the "infinite amount of activity".)

There are idealized processes that natually exhibit Zeno behaviors. As an example consider a ball which is bouncing on the floor and is losing a fraction of its energy at each bounce. Ideally the ball will bounce infinitely many times within a finite amount of time. Note, however, that the safe timed I/O automaton model cannot suitably model this process since there is no way of specifying what happens after the ball stops bouncing. On the other hand, Zeno behaviors will not occur in the computer systems we usually want to specify.

Below we will be mostly interested in the admissible timed executions since they correspond to our intuition that time is a force beyond our control that happens to approach infinity.

Denote by $t\text{-}frag^*(A)$, $t\text{-}frag^\infty(A)$, $t\text{-}frag^Z(A)$, and $t\text{-}frag(A)$ the sets of finite, admissible, Zeno, and all timed execution fragments of A. Similarly, denote by $t\text{-}exec^*(A)$, $t\text{-}exec^\infty(A)$, $t\text{-}exec^Z(A)$, and $t\text{-}exec(A)$ the sets of finite, admissible, Zeno, and all timed executions of A.

A finite timed execution fragment $\Sigma_1 = \omega_0 a_1 \omega_1 \cdots a_n \omega_n$ of A and a timed execution fragment $\Sigma_2 = S\omega_n' a_{n+1}\omega_{n+1} a_{n+2}\omega_{n+2}\cdots$ of A can be *concateneted* if $lstate(\Sigma_1) = fstate(\Sigma_2)$. The concatenation, written $\Sigma_1 \frown \Sigma_2$, is defined to be $\Sigma = \omega_0 a_1 \omega_1 \cdots a_n(\omega_n \frown \omega_n') a_{n+1}\omega_{n+1} a_{n+2}\omega_{n+2}\cdots$, where $(\omega \frown \omega')$ is defined to be $\omega(t)$ if t is in $dom(\omega)$, and $\omega'(t)$ if t is in $dom(\omega')\setminus dom(\omega)$. It is easy to see that Σ is a timed execution fragment of A.

The notion of timed prefix, called *t-prefix*, for timed execution fragments is defined as follows. A timed execution fragment Σ_1 of A is a *t-prefix* of a timed execution fragment Σ_2 of A, written $\Sigma_1 \leq_t \Sigma_2$, if either $\Sigma_1 = \Sigma_2$ or Σ_1 is finite and there exists a timed execution fragment Σ_1' of A such that $\Sigma_2 = \Sigma_1 \frown \Sigma_1'$. Likewise, Σ_1 is a *t-suffix* of Σ_2 if there exists a finite timed execution fragment Σ_1' such that $\Sigma_2 = \Sigma_1' \frown \Sigma_1$.

Define $\Sigma \lhd t$, read "Σ before t", for all $t \geq ftime(\Sigma)$, to be the t-prefix of Σ that includes exactly all states with times not bigger than t.

Likewise, define $\Sigma \rhd t$, read "Σ after t", for all $t < ltime(\Sigma)$ or all $t \leq ltime(\Sigma)$ when Σ is finite, to be the t-suffix of Σ that includes exactly all states with times not smaller than t.

2.2.1.3 Timed Traces

In the untimed setting automata are compared based on their traces. This turns out to be inadequate in the timed setting because traces do not capture the invisible nature of time-passage actions and furthermore do not contain information about the time of occurrence of the visible actions. For this reason a notion of *timed traces* is introduced. We first define the notion of *timed sequence*.

A *timed sequence* over a set K is defined to be a (finite or infinite) sequence δ over $K \times \mathsf{T}$ in which the second components (the *time* components) are nondecreasing. Define δ to be *Zeno* if it is infinite and the limit of the time components is finite. For any nonempty timed sequence δ, define *ftime*(δ) to be the time component of the first pair in δ.

Now, let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be a timed execution fragment of a safe timed I/O automaton A. For each a_i, define the *time of occurrence* t_i to be *ltime*(ω_{i-1}), or equivalently, *ftime*(ω_i). Then, define *t-seq*(Σ) to be the timed sequence consisting of the actions in Σ paired with their time of occurrence:

$$\textit{t-seq}(\Sigma) = (a_1, t_1)(a_2, t_2)\cdots$$

Then *t-trace*(Σ), the *timed trace* of Σ, is defined to be the pair

$$\textit{t-trace}(\Sigma) \triangleq (\textit{t-seq}(\Sigma) \restriction (\textit{vis}(A) \times \mathsf{T}), \textit{ltime}(\Sigma))$$

Thus, *t-trace*(Σ) records the occurrences of *visible* actions together with their time of occurrence, and the limit time of the timed execution fragment. The timed trace suppresses both internal and time-passage actions.

Let *t-traces*$^*(A)$, *t-traces*$^\infty(A)$, *t-traces*$^Z(A)$, and *t-traces*(A) denote the sets of timed traces of A obtained from finite, admissible, Zeno, and all timed executions of A, respectively.

2.2.1.4 Operations on Safe Timed I/O Automata

As in the untimed setting, there are three operators defined on safe (timed) I/O automata. These are *parallel composition*, *action hiding*, and *action renaming*. The definitions are similar to the ones in the untimed setting except that special care has to be taken concerning the handling of time. For instance, in the parallel composition, all components must agree on real time.

Definition 2.18 (Parallel composition)

Safe timed I/O automata $A_1, \ldots, A_N$ are *compatible* if for all $1 \leq i, j \leq N$ with $i \neq j$

1. $\textit{out}(A_i) \cap \textit{out}(A_j) = \emptyset$
2. $\textit{int}(A_i) \cap \textit{acts}(A_j) = \emptyset$

The *parallel composition* $A_1 \| \cdots \| A_N$ of compatible safe timed I/O automata $A_1, \ldots, A_N$ is the safe timed I/O automaton A such that

1. $\textit{states}(A) = \{(s_1, \ldots, s_N) \in \textit{states}(A_1) \times \cdots \times \textit{states}(A_N) \mid s_1.\textit{now}_{A_1} = \cdots = s_N.\textit{now}_{A_N}\}$
2. $\textit{start}(A) = \textit{start}(A_1) \times \cdots \times \textit{start}(A_N)$
3. $(s_1, \ldots, s_N).\textit{now}_A = s_1.\textit{now}_{A_1}$ $(= s_2.\textit{now}_{A_2} = \cdots = s_N.\textit{now}_{A_N})$
4. $\textit{out}(A) = \textit{out}(A_1) \cup \cdots \cup \textit{out}(A_N)$
5. $\textit{in}(A) = (\textit{in}(A_1) \cup \cdots \cup \textit{in}(A_N)) \setminus \textit{out}(A)$
6. $\textit{int}(A) = \textit{int}(A_1) \cup \cdots \cup \textit{int}(A_N)$

7. $((s_1, \ldots, s_N), a, (s'_1, \ldots, s'_N)) \in \mathit{steps}(A)$ iff for all $1 \leq i \leq N$
   (a) if $a \in \mathit{acts}(A_i)$ then $(s_i, a, s'_i) \in \mathit{steps}(A_i)$
   (b) if $a \notin \mathit{acts}(A_i)$ then $s_i = s'_i$

■

Note, how Condition 7 of the definition captures both time-passage steps (where all components participate) and other steps (where a subset of the components participate).

Just like (ordinary) execution fragments can be projected to components in a composed system, it is possible to define projection on timed execution fragments. If $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ is a timed execution fragment of a safe timed I/O automaton $A = A_1 \| \cdots \| A_N$, define $\Sigma \lceil A_i$ to be the timed execution fragment of A_i obtained by first projecting each state in the range of each trajectory to A_i, and then, for each action $a_j \notin \mathit{acts}(A_i)$, removing a_j and merging the two (projected) trajectories to the left and right of a_j. (Thus, if none of the actions belongs to A_i, the result is one big trajectory representing time-passage of A_i.)

Action hiding and action renaming for safe timed I/O automata can also be defined.

Definition 2.19 (Action hiding)

Let A be a safe timed I/O automaton and let $\mathcal{A}$ be a set of actions such that $\mathcal{A} \subseteq \mathit{local}(A)$. Then define $A \setminus \mathcal{A}$ to be the safe timed I/O automaton such that

1. $\mathit{states}(A \setminus \mathcal{A}) = \mathit{states}(A)$
2. $\mathit{start}(A \setminus \mathcal{A}) = \mathit{start}(A)$
3. $.\mathit{now}_{A \setminus \mathcal{A}} = .\mathit{now}_A$
4. $\mathit{in}(A \setminus \mathcal{A}) = \mathit{in}(A)$
5. $\mathit{out}(A \setminus \mathcal{A}) = \mathit{out}(A) \setminus \mathcal{A}$
6. $\mathit{int}(A \setminus \mathcal{A}) = \mathit{int}(A) \cup \mathcal{A}$
7. $\mathit{steps}(A \setminus \mathcal{A}) = \mathit{steps}(A)$

■

Definition 2.20 (Action renaming)

A mapping ρ from actions to actions is *applicable* to a safe timed I/O automaton A if it is injective, $\mathit{acts}(A) \subseteq \mathit{dom}(\rho)$, and $\rho(\nu) = \nu$. Given a safe timed I/O automaton and a mapping ρ applicable to A, define $\rho(A)$ to be the safe timed I/O automaton with

1. $\mathit{states}(\rho(A)) = \mathit{states}(A)$
2. $\mathit{start}(\rho(A)) = \mathit{start}(A)$
3. $.\mathit{now}_{\rho(A)} = .\mathit{now}_A$
4. $\mathit{in}(\rho(A)) = \rho(\mathit{in}(A))$
5. $\mathit{out}(\rho(A)) = \rho(\mathit{out}(A))$

6. $int(\rho(A)) = \rho(int(A))$

7. $steps(\rho(A)) = \{(s, \rho(a), s') \mid (s, a, s') \in steps(A)\}$

■

2.2.2 Live Timed I/O Automata

In the untimed setting a liveness condition for a safe I/O automaton A is a subset of the executions of A such that a special environment-freedom condition is satisfied. Similarly, in the timed setting a liveness condition for a safe timed I/O automaton is a set of *timed* executions such that a special timed version of the environment-freedom condition is satisfied.

As in the untimed setting the environment-freedom condition is stated in terms of a game between the system and its environment.

The notion of strategy is similar to the one used for the untimed case. However, the presence of time has a strong impact on the kind of interactions that can occur between an automaton and its environment.

In the untimed case the environment is allowed to provide any finite number of input actions at each move, whereas the system is allowed to perform at most one of its locally-controlled steps at each move. In this way it is taken into account that the environment can be arbitrarily fast with respect to a system, however, not infinitely fast. In the timed case there is no need to assume the environment to be arbitrarily fast because each action occurs at a specific time. Therefore, the relative speeds of the system and the environment are given by their timing constraints. As a consequence the moves of the environment in the timed setting are input actions associated with their time of occurrence. Thus, the behavior of the environment during the game can be represented as a timed sequence over input actions.

If a strategy in the timed setting decides to let time pass, it has to specify explicitly all intermediate states since the system must be able to respond to possible inputs during such a time-passage phase. Remember, that in our model it is generally not possible to deduce deterministically states at intermediate times given a time-passage step.

Definition 2.21 (Strategy)

Consider any safe timed I/O automaton A. A *strategy* defined on A is a pair of functions (g, f) where $g : t\text{-}exec^*(A) \times in(A) \to states(A)$ and $f : t\text{-}exec^*(A) \to (traj(A) \times local(A) \times states(A)) \cup traj(A)$, where $traj(A)$ denotes the set of A-trajectories, such that

1. $g(\Sigma, a) = s$ implies $\Sigma a\{s\} \in t\text{-}exec^*(A)$

2. $f(\Sigma) = (\omega, a, s)$ implies $\Sigma \frown \omega a\{s\} \in t\text{-}exec^*(A)$

3. $f(\Sigma) = \omega$ implies $\Sigma \frown \omega \in t\text{-}exec^\infty(A)$

4. f is *consistent*, i.e., if $f(\Sigma) = (\omega, a, s)$, then, for each t, $ftime(\omega) \leq t \leq ltime(\omega)$, $f(\Sigma \frown (\omega \lhd t)) = (\omega \rhd t, a, s)$, and, if $f(\Sigma) = \omega$, then, for each t, $ftime(\omega) \leq t < ltime(\omega)$, $f(\Sigma \frown (\omega \lhd t)) = \omega \rhd t$.

For notational convenience define $f(\Sigma).trj \triangleq \begin{cases} \omega & \text{if } f(\Sigma) = (\omega, a, s) \\ \omega & \text{if } f(\Sigma) = \omega \end{cases}$

■

A strategy is a pair of function (g, f). Function f takes a finite timed execution and decides how the system behaves till its next locally-controlled action under the assumption that no input are received in the meantime, whereas function g decides what state to reach whenever some input is received. Condition 1 states that g returns a "legal" next state given the input. Conditions 2 and 3 give two possibilities for the system moves given by f: either f specifies time-passage followed by a local step, or f specifies that the system simply lets time pass forever. Note, that f specifies all states during time passage. This is because, as mentioned above and as we shall see formally below, a move given by f might be interrupted by input actions, and in that case it is necessary to know the current state when the inputs arrive. The consistency condition (Condition 4) for f says that, whenever after a finite timed execution Σ the system decides to behave according to $\omega a\{s\}$ or ω, after performing a part of ω the system would decide to behave according to the rest of the step $\omega a\{s\}$ or ω. The consistency condition is fundamental for the substitutivity results below.

The game between the system and the environment works as follows. The environment can provide any input at any time, while the system lets time pass and provides locally-controlled actions according to its strategy. If an input arrives, the system will perform its current step till the time at which the input occurs, and then use function g to compute the state to reach after the input has occurred.

In the timed setting the system might decide to perform a step at the same time at which the environment provides some input. Such situations are modeled as nondeterministic choices. As a consequence, the *outcome*, i.e., the result of the game, for a timed strategy is a *set* of timed executions.

Definition 2.22 (Outcome of a strategy)

Let A be a safe timed I/O automaton and (g, f) a strategy defined on A. Define a *timed environment sequence* for A to be a timed sequence over $in(A)$, and define a timed environment sequence $\mathcal{I}$ for A to be *compatible* with a timed execution fragment Σ of A if either $\mathcal{I}$ is empty, or Σ is finite and $ltime(\Sigma) \leq ftime(\mathcal{I})$. Then define $R_{(g,f)}$, the *next-relation induced by* (g, f), as follows: for any $\Sigma, \Sigma' \in t\text{-}exec(A)$ and any $\mathcal{I}, \mathcal{I}'$ compatible with Σ, Σ', respectively, $((\Sigma, \mathcal{I}), (\Sigma', \mathcal{I}')) \in R_{(g,f)}$ iff

$$(\Sigma', \mathcal{I}') = \begin{cases} (\Sigma \frown \omega a\{s\}, \mathcal{I}) & \text{where } \Sigma \text{ is finite, } \mathcal{I} = \varepsilon,\ f(\Sigma) = (\omega, a, s), \\[2ex] (\Sigma \frown \omega, \mathcal{I}) & \text{where } \Sigma \text{ is finite, } \mathcal{I} = \varepsilon,\ f(\Sigma) = \omega, \\[2ex] (\Sigma \frown \omega a\{s\}, \mathcal{I}) & \text{where } \Sigma \text{ is finite, } \mathcal{I} = (b, t)\mathcal{I}'', f(\Sigma) = (\omega, a, s), \\ & \qquad\qquad ltime(\omega) \leq t, \\[2ex] (\Sigma \frown \omega' a\{s'\}, \mathcal{I}'') & \text{where } \Sigma \text{ is finite, } \mathcal{I} = (a, t)\mathcal{I}'',\ f(\Sigma).trj = \omega, \\ & \qquad\qquad ltime(\omega) \geq t,\ \ \omega' = \omega \lhd t,\ \ g(\Sigma \frown \omega', a) = s', \text{ or} \\[2ex] (\Sigma, \mathcal{I}) & \text{where } \Sigma \text{ is not finite.} \end{cases}$$

Let Σ be a finite timed execution of A, and $\mathcal{I}$ be a timed environment sequence for A *compatible* with Σ.

An *outcome sequence of* (g, f) *given* Σ *and* $\mathcal{I}$ is an infinite sequence $(\Sigma^n, \mathcal{I}^n)_{n \geq 0}$ that satisfies:

- $(\Sigma^0, \mathcal{I}^0) = (\Sigma, \mathcal{I})$ and

- for all $n > 0$, $((\Sigma^{n-1}, \mathcal{I}^{n-1}), (\Sigma^n, \mathcal{I}^n)) \in R_{(g,f)}$.

Note, that $(\Sigma^n)_{n\geq 0}$ forms a chain ordered by *t-prefix*.

The *outcome* $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I})$ of the strategy (g, f) given Σ and $\mathcal{I}$ is the set of timed executions Σ' for which there exists an outcome sequence $(\Sigma^n, \mathcal{I}^n)_{n\geq 0}$ of (g, f) given Σ and $\mathcal{I}$ such that $\Sigma' = \lim_{n\to\infty} \Sigma^n$.

■

In the definition of outcome of a strategy (g, f), the next-relation $R_{(g,f)}$ determines allowable moves based on incoming inputs or performance of locally-controlled actions. In this way the outcome sequences of (g, f) given some Σ and $\mathcal{I}$ are determined step by step.

In the definition of $R_{(g,f)}$, the first, second, and third cases deal with different situations where no input occurs during the system move chosen by f; the fourth case, instead, takes care of new incoming inputs; finally, the fifth case of the above definition is needed for technical reasons to generate a fixpoint in the outcome sequences since the second case generates an admissible timed execution. Note, that the third and fourth cases might both be applicable whenever an input occurs exactly at the same time at which the system decides to perform a locally-controlled action. This is the reason for which the outcome is a *set* of timed executions.

Assume that the liveness condition for a safe timed I/O automaton could consist of Zeno timed executions only. If another safe timed I/O automaton has a liveness condition consisting of admissible timed executions, both of these systems could never work properly when composed in parallel since the first system would keep time from passing beyond some bound, which could never yield live timed execitions of the second system. (Remember that all components in a parallel composition have to agree on real time.)

In this model this problem is solved by restricting attention to admissible timed executions since these timed executions correspond to our intuition that time grows unboundedly. Thus, in a live timed I/O automaton a liveness condition is a nonempty subset of the admissible timed executions.

However, a problem arises as illustrated by the following example, which is due to Lamport: Consider two almost identical safe timed I/O automata with the following characteristics. They both have one input action and one output action, and if they receive an input before 12 o'clock they will issue an output after exactly half the time between the input was received and 12 o'clock. Otherwise no output will be issued. To break the symmetry, one of the safe timed I/O automata will unconditionally issue an output some time before 12 o'clock. Both of these safe timed I/O automata have a nonempty set of admissible timed executions, so adopt these sets to be the liveness conditions of the safe timed I/O automata, respectively. Now, compose these systems in parallel by connecting the output of one system to the input of the other, and vice versa. Then the resulting system has no admissible timed executions but only Zeno timed executions where time is constrained from passing beyond 12 o'clock. Seen from any of the components the other component prevents time from passing, and none of the components will behave properly in the parallel composition. Thus, the parallel composition would not be an element of the model (since it has no admissible timed executions), which contradicts the requirement that the parallel composition operator be closed for live timed I/O automata.

The problem illustrated in the example arises because the two components *collaborate* on performing the Zeno timed executions. To solve the problem, systems that can collaborate in this fashion need to be excluded from the model. We do this by identifying a special class of

Zeno timed executions, the *Zeno-tolerant* timed executions. A Zeno-tolerant timed execution is a Zeno timed execution containing infinitely many input actions but only finitely many locally-controlled actions. We denote by $t\text{-}exec^{Zt}(A)$ the set of Zeno-tolerant timed executions of a safe time I/O automaton A.

The Zeno-tolerant timed executions represent Zeno behaviors that are exclusively due to a Zeno environment. Thus, there is no collaboration between system and environment. This gives rise to a notion of *Zeno-tolerant strategy*.

Definition 2.23 (Zeno-tolerant strategy)

A strategy (g, f) defined on a safe timed I/O automaton A is said to be *Zeno-tolerant* if, for every finite timed execution $\Sigma \in t\text{-}exec^*(A)$ and every timed environment sequence $\mathcal{I}$ for A compatible with Σ, $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq t\text{-}exec^\infty(A) \cup t\text{-}exec^{Zt}(A)$.

■

Thus, any Zeno timed execution in an outcome of a Zeno-tolerant strategy is Zeno-tolerant and thus represents a behavior that is Zeno only because of Zeno inputs from the environment. Note, that in the Lamport example above it is not possible to find a Zeno-tolerant strategy defined on any of the two components: if one component behaves in a Zeno fashion, the other component will collaborate, and the resulting outcome cannot contain Zeno-tolerant timed executions.

We are now ready to present the timed definition of environment-freedom.

Definition 2.24 (Environment-freedom)

A pair (A, L), where A is a safe timed I/O automaton and $L \subseteq t\text{-}exec(A)$, is *environment-free* iff there exists a Zeno-tolerant strategy (g, f) defined on A such that for each finite timed execution Σ of A and each timed environment sequence $\mathcal{I}$ for A compatible with Σ, $\mathcal{O}_{(g,f)}(\Sigma, \mathcal{I}) \subseteq L$. The pair (g, f) is called an *environment-free* strategy for (A, L).

■

A pair (A, L) is environment-free if, after any finite timed execution and with any (Zeno or non-Zeno) sequence of input actions, it can behave according to some admissible or Zeno-tolerant timed execution in A.

This leads to the definition of *live timed I/O automata*, where the liveness condition contains only admissible timed executions, but where the strategy is allowed to yield *Zeno-tolerant* outcomes when given a Zeno timed environment sequence.

Definition 2.25 (Live timed I/O automata)

A *live timed I/O automaton* is a pair (A, L), where A is a safe timed I/O automaton and $L \subseteq t\text{-}exec^\infty(A)$, such that the pair $(A, L \cup t\text{-}exec^{Zt}(A))$ is environment-free.

■

2.2.2.1 Operations on Live Timed I/O Automata

The parallel composition, action hiding, and action renaming operators defined for safe timed I/O automata are now extended to live timed I/O automata in a fashion similar to the way the operators were extended in the untimed setting.

Definition 2.26 (Parallel composition of live timed I/O automata)

Live timed I/O automata $(A_1, L_1), \ldots, (A_N, L_N)$ are *compatible* iff the safe timed I/O automata $A_1, \ldots, A_N$ are compatible.

The *parallel composition* $(A_1, L_1)\| \cdots \|(A_N, L_N)$ of compatible live timed I/O automata $(A_1, L_1), \ldots, (A_N, L_N)$ is defined to be the pair (A, L) where $A = A_1 \| \cdots \| A_N$ and $L = \{\Sigma \in t\text{-}exec^{\infty}(A) \mid \Sigma\lceil A_1 \in L_1, \ldots, \Sigma\lceil A_N \in L_N\}$.

■

Definition 2.27 (Action hiding of live timed I/O automata)

Let (A, L) be a live timed I/O automaton and let $\mathcal{A}$ be a set of actions such that $\mathcal{A} \subseteq local(A)$. Then define $(A, L) \setminus \mathcal{A}$ to be the pair $(A \setminus \mathcal{A}, L)$.

■

Definition 2.28 (Action renaming of live timed I/O automata)

A mapping ρ from actions to actions is *applicable* to a live timed I/O automaton (A, L) if it is applicable to A. Let Σ be a timed execution of (A, L). Define $\rho(\Sigma)$ to be the sequence that results from replacing each occurrence of every action a in Σ by $\rho(a)$. Given a live timed I/O automaton and a mapping ρ applicable to (A, L), define $\rho((A, L))$ to be the pair $(\rho(A), \rho(L))$.

■

As expected the three operators above are closed for live timed I/O automata in the sense that they produce a new live timed I/O automaton. This is a consequence of the environment-freedom property.

Lemma 2.29 (Closure of timed parallel composition)

Let $(A_1, L_1), \ldots, (A_N, L_N)$ be compatible live timed I/O automata. Then the parallel composition $(A_1, L_1)\| \ldots \|(A_N, L_N)$ is a live timed I/O automaton.

■

Lemma 2.30 (Closure of action hiding)

Let (A, L) be a live timed I/O automaton and let $\mathcal{A} \subseteq local(A)$. Then $(A, L) \setminus \mathcal{A}$ is a live timed I/O automaton.

■

Lemma 2.31 (Closure of action renaming)

Let (A, L) be a live timed I/O automaton and let ρ be a mapping applicable to (A, L). Then $\rho((A, L))$ is a live timed I/O automaton.

■

2.2.3 Correctness

In the timed setting the *safe* and *correct* implementation relations are based on *timed* traces.

Definition 2.32 (Timed implementation relations)

Given two live timed I/O automata (A, L) and (B, M) such that $in(A) = in(B)$ and $out(A) = out(B)$, define the following implementation relations:

Safe:	$A \sqsubseteq_{\mathrm{St}} B$	iff	$\textit{t-traces}(A) \subseteq \textit{t-traces}(B)$
Safe:	$(A, L) \sqsubseteq_{\mathrm{St}} (B, M)$	iff	$A \sqsubseteq_{\mathrm{St}} B$
Correct:	$(A, L) \sqsubseteq_{\mathrm{Lt}} (B, M)$	iff	$\textit{t-traces}(L) \subseteq \textit{t-traces}(M)$

■

2.2.4 Substitutivity

The timed model, like the untimed model, offers a modular approach to systems specification and verification as stated by the following substitutivity results.

Proposition 2.33 (Substitutivity)

Let $(A_i, L_i), (B_i, M_i)$, $i = 1, \ldots, N$, be live timed I/O automata with $in(A_i) = in(B_i)$ and $out(A_i) = out(B_i)$, and let $\sqsubseteq_X$ be one relation among $\sqsubseteq_{\mathrm{St}}$ and $\sqsubseteq_{\mathrm{Lt}}$. If, for each i, $(A_i, L_i) \sqsubseteq_X (B_i, M_i)$, then

1. *if $(A_1, L_1), \ldots, (A_N, L_N)$ are compatible and $(B_1, M_1), \ldots, (B_N, M_N)$ are compatible then*
   $(A_1, L_1) \| \cdots \| (A_N, L_N) \sqsubseteq_X (B_1, M_1) \| \cdots \| (B_N, M_N)$.
2. *if $\mathcal{A} \subseteq local(A_1)$ and $\mathcal{A} \subseteq local(B_1)$ then*
   $(A_1, L_1) \setminus \mathcal{A} \sqsubseteq_X (B_1, M_1) \setminus \mathcal{A}$
3. *if ρ is a mapping applicable to both A_1 and B_1 then*
   $\rho((A_1, L_1)) \sqsubseteq_X \rho((B_1, M_1))$

■

2.3 Embedding Results

The untimed model is used to specify systems where the actual amount of time that passes between actions is considered unimportant. Many problems in distributed computing can be stated and solved using this model. However, it is not possible to state anything about, e.g., response times. It is implicitly assumed that the final implementation on a physical machine is "fast enough" for practical usage.

An untimed system can be thought of as a timed system that allows arbitrary time-passage, as long as possible liveness restrictions are satisfied. This indicates that our timed model is, in some sense, more general than our untimed model, and that we could use the timed model for all purposes. However, the timed model is more complicated than the untimed model due to the time-passage action, the *.now* component, etc., and furthermore it does not seem natural to have to deal with time, when the problem to be solved does not mention time at all.

Thus, it is preferable to work within the untimed model as much as possible and only switch to the timed model when it is needed. The work in this report shows how the untimed specification (of the at-most-once message delivery problem) is implemented by a system that assumes upper time bounds on certain process steps and channel delays. Figure 2.1 depicts such a stepwise

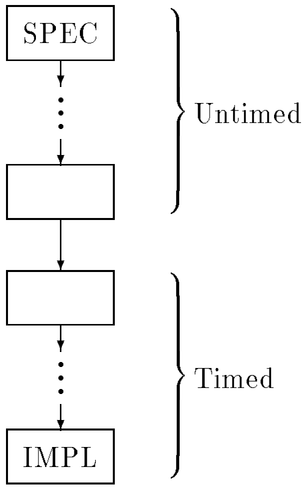


Figure 2.1

A stepwise development from an untimed specification to a timed implementation.

development. The question is of course what it means to implement an untimed specification by a timed implementation. Our approach is to convert the untimed levels to the timed model by applying an operator, called *patient*, that adds arbitrary time-passage steps as mentioned above. We then have an *Embedding Theorem* which states that if a concrete level implements an abstract level in the untimed model, then the *patient* version of the concrete level implements the *patient* version of the abstract level in the timed model, and vice versa. Thus, the first part of the stepwise development of Figure 2.1 can be carried out entirely in the simpler untimed model, and the last part in the timed model. In the intermediate development step which goes from untimed to timed, one must prove that the timed level implements the *patient* version of the untimed level. The embedding lemma can then be applied to show that the implementation IMPL implements the *patient* version of the specification SPEC.

We start by defining a *patient safe I/O automaton.*

Definition 2.34 (Patient safe I/O automaton)

Let A be a safe I/O automaton where $\nu \notin acts(A)$. Then define $patient(A)$ to be the safe timed I/O automaton with

- $states(patient(A)) = states(A) \times \mathsf{T}$

  If $s = (s', t)$ is a state of $patient(A)$, we let $s.basic$ denote s'.

- $start(patient(A)) = start(A) \times \{0\}$

- $.now_{patient(A)}(s, t) = t$

- $ext(patient(A)) = ext(A) \cup \{\nu\}$

- $in(patient(A)) = in(A)$

- $out(patient(A)) = out(A)$
- $int(patient(A)) = int(A)$
- $steps(patient(A))$ consists of the steps
  - $\{((s,t), a, (s',t)) \mid (s,a,s') \in steps(A)\}$
  - $\{((s,t), \nu, (s,t')) \mid t' > t\}$

■

In order to state what it means to apply the *patient* operator to a live I/O automaton, we need the following auxiliary definition of what it means to *untime* a timed execution: Let A be a safe I/O automaton with $\nu \notin acts(A)$ and let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be a timed execution of $patient(A)$. Then define

$$untime(\Sigma) = (fstate(\omega_0).basic)a_1(fstate(\omega_1).basic)a_2(fstate(\omega_2).basic)\cdots$$

Similarly, let $\gamma = ((a_1,t_1)(a_2,t_2)\cdots,t)$ be a timed trace of $patient(A)$. Then define

$$untime(\gamma) = a_1 a_2 \cdots$$

The notion of a *patient live I/O automaton* can now be defined. For any live I/O automaton (A,L), the patient live I/O automaton of (A,L) should be the live timed I/O automaton whose safety part is $patient(A)$ and whose liveness part consists of all those admissible executions that, when being made *untime*d, are live according to L. Thus, the liveness condition of the patient live I/O automaton allows time to pass arbitrarily, as long as the liveness prescribed by L is satisfied.

Definition 2.35 (Patient live I/O automaton)

Let (A,L) be a live I/O automaton with $\nu \notin acts(A)$. Then, define $patient_A(L) = \{\Sigma \in t\text{-}exec^\infty(patient(A)) \mid untime(\Sigma) \in L\}$ and define $patient(A,L)$, the *patient live I/O automaton* of (A,L), to be the pair $(patient(A), patient_A(L))$.

■

It can be proved that for any live I/O automaton (A,L), $patient(A,L)$ is a live timed I/O automaton.

Lemma 2.36

Let (A,L) be a live I/O automaton. Then $patient(A,L)$ is a live timed I/O automaton.

■

We now state the Embedding Theorem, thus that the safe and correct implementation relations for live I/O automata coincide with the safe and correct implementation relations for the patient versions of the live I/O automata.

Theorem 2.37 (Embedding Theorem)

Let (A,L) and (B,M) be live I/O automata with $\nu \notin (acts(A) \cup acts(B))$. Then

1. $(A, L) \sqsubseteq_{\mathrm{S}} (B, M)$ *iff* $\mathit{patient}(A, L) \sqsubseteq_{\mathrm{St}} \mathit{patient}(B, M)$.
2. $(A, L) \sqsubseteq_{\mathrm{L}} (B, M)$ *iff* $\mathit{patient}(A, L) \sqsubseteq_{\mathrm{Lt}} \mathit{patient}(B, M)$.

■

Finally we state a result which is important when doing specification and verification in a modular fashion. Namely, the *patient* operator commutes with the three operators on safe and live (timed) I/O automata. First, let $\equiv_{\mathrm{St}}$ and $\equiv_{\mathrm{Lt}}$ denote the *kernels* of the preorders $\sqsubseteq_{\mathrm{St}}$ and $\sqsubseteq_{\mathrm{Lt}}$, respectively.[3]

Proposition 2.38

Let (A, L) and $(A_1, L_1), \ldots, (A_N, L_N)$ be live I/O automata and let $\equiv_X$ be one of $\equiv_{\mathrm{St}}$ and $\equiv_{\mathrm{Lt}}$.

1. *Let $(A_1, L_1), \ldots, (A_N, L_N)$ be compatible. Then,*
   $\mathit{patient}((A_1, L_1)\| \cdots \|(A_N, L_N)) \equiv_X \mathit{patient}(A_1, L_1)\| \cdots \|\mathit{patient}(A_N, L_N)$
2. *Let $\mathcal{A} \subseteq \mathit{local}(A)$. Then,*
   $\mathit{patient}((A, L) \setminus \mathcal{A}) \equiv_X \mathit{patient}(A, L) \setminus \mathcal{A}$
3. *Let ρ be an action mapping applicable to A and let ρ_ν be $\rho \cup [\nu \mapsto \nu]$. Then,*
   $\mathit{patient}(\rho(A, L)) \equiv_X \rho_\nu(\mathit{patient}(A, L))$

■

This concludes the introduction to the basic models of untimed and timed systems that we will use in this work.

[3]The kernel of a preorder $\sqsubseteq$ is defined to be the equivalence $\equiv$ defined by $x \equiv y \ \triangleq\ x \sqsubseteq y \wedge y \sqsubseteq x$.

Chapter 3

A Temporal Logic with Step Formulas

Chapter 2 defined the models of distributed systems we use in this work. One component of the models is the liveness condition which is a set of (timed) executions. Since such sets may be infinite (and each execution in the set may be an infinite sequence), it is necessary to have some way of denoting them without explicitly having to write down any executions. For this purpose we shall use a *temporal logic* which will be able to express properties of (ordinary) executions of safe (timed) I/O automata. Exactly how this temporal logic is used to specify liveness conditions for timed and untimed systems will be one of the issues of Chapter 4. This chapter is devoted to defining the temporal logic.

In [MP92], Manna and Pnueli develop a temporal logic and give several examples of its use. For two reasons we cannot use their temporal logic directly. First, Manna and Pnueli evaluate temporal formulas over sequences of states and not over sequences of alternating states and actions. Second, they only deal with infinite sequences of states whereas (even live) executions of our systems may be finite. In a section below we show, however, how our temporal logic is related to that of [MP92].

The first reason suggests that maybe Lamport's Temporal Logic of Actions (TLA) [Lam91] could be used. However, TLA is still state based in the sense that the semantics of a TLA formula is a set of sequences of *states*. Actions are in TLA merely state changes. It is possible that by having special TLA variables ranging over action names we could use TLA. However, due to the inherent importance of actions in our approach, we chose to develop our own temporal logic dealing with actions in a more intuitive manner.

The rest of this chapter is organized as follows: In order to be able to state and prove results in this and later chapters, we start by introducing notions of *stuttering* and *stuttering-equivalence* in Section 3.1. Sections 3.2–3.4 then introduce the basic building blocks of our temporal logic: first, in Section 3.2, we introduce the notion of *state functions* and the special notion of *state predicates*. Section 3.3 then describes the notion of *state transition functions*, which are state functions that are evaluated over *pairs* of states. Finally, in Section 3.4, we introduce the important notion of *step formulas*. A step formula is a boolean valued function which is evaluated over steps. Thus, step formulas can express properties of both the states and the action of a step.

Sections 3.5 and 3.6 now introduce the formulas of our temporal logic, i.e., the *temporal* *formulas*, by first, in Section 3.5, giving some basic *temporal operators* and then, in Section 3.6, defining some important derived operators. In Section 3.7 we see how temporal formulas can be seen as formulas over safe (timed) I/O automata, and Section 3.8 deals with *satisfaction* and *validity* as well as validity with respect to safe (timed) I/O automata or sets of executions.

Sections 3.9 and 3.10 provide results, mainly about special *stuttering-insensitive* formulas, which will prove very important in the next chapter.

Then, in Section 3.11 we compare out temporal logic with that of Manna and Pnueli [MP92]. Finally, in order for our temporal logic to be useful for proving correctness of the protocols in the second part of this report, Section 3.12 provides certain *rules* of the logic. We do not in this work attempt to develop a completely axiomatized temporal logic, but merely state the rules we have found useful. Further research should investigate a basic set of rules of our temporal logic.

Even though, strictly speaking, executions are only defined with respect to specific automata, we will in this chapter use the term “execution” to denote any alternating sequence of states and actions. As usual we let α range over executions.

3.1 Stuttering

For technical reasons which will become clear below, we introduce a notion of *stuttering steps* and *stuttering-equivalence* of executions.

Denote by ζ a special *stuttering* action. We will assume that ζ cannot be used as an ordinary action of any safe (timed) I/O automaton. Below we will let $\mathcal{A}$ denote an arbitrary set of actions and, hence, it will always be the case that $\zeta \notin \mathcal{A}$. A *stuttering step* is any triple of the form (s, ζ, s), where s is a state.

Since ζ can never be an action of a safe (timed) I/O automaton A, it can never occur in any execution of A. However, we will allow stuttering steps to occur in the more broad sense of executions used in this chapter. As we shall see below, we will not be able in temporal formulas to refer to the stuttering actions in executions, but it turns out to be important to be able to evaluate temporal formulas over executions possibly containing stuttering.

Define $\natural\alpha$ to be the execution obtained by replacing every maximal (finite or infinite) sequence $s\zeta s\zeta s\cdots$ in α by the single state s. Thus, the $\natural$ operator removes all stuttering. Now, define two executions α_1 and α_2 to be *stuttering-equivalent*, written $\alpha_1 \simeq \alpha_2$, if $\natural\alpha_1 = \natural\alpha_2$.

For any execution $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ define

$$\widehat{\alpha} \triangleq \begin{cases} \alpha & \text{if } \alpha \text{ is infinite} \\ s_0 a_1 s_1 a_2 s_2 \cdots a_n s_n \zeta s_n \zeta s_n \cdots & \text{if } \alpha \text{ is finite and ends in } s_n \end{cases}$$

Thus, if α is finite, $\widehat{\alpha}$ is the infinite execution obtained by concatenating infinite stuttering at the end of α. Clearly, $\alpha \simeq \widehat{\alpha}$.

3.2 States, State Functions, and State Predicates

In Chapter 2 we defined the state space of a safe (timed) I/O automaton to be any set of individual states. We did not assume any structure of these states but merely assumed that states are names. In practical examples, especially those presented in this work, the state space will be described as a mapping from state variables to their values. Thus, a safe (timed) I/O automaton is assumed to contain a number of (typed) state variables, and the individual states are then distinguished by having different assignments of values to these state variables. For this reason the temporal logic defined below will reference states using variable names. This approach is also used in [MP92, Lam91]. Below we will let $\mathcal{V}$ denote a set of variables. Furthermore, in order to avoid the complexity of carrying around the types of the variables, we assume that the type of a variable is given implicitly by the name of the variable. For example, i, j and k will typically range over the natural numbers.

We assume that we have a language for writing *state functions*—using variables, constants, standard operators, boolean connectives, and quantification—that can be evaluated over states. We will not give a language for writing down state functions since such languages are fairly standard. We refer to, e.g., [MP92] for a more thorough treatment of state functions.

A state function *over* $\mathcal{V}$ is a state function whose *free* variables are a subset of $\mathcal{V}$. If f is a state function over $\mathcal{V}$, then clearly f is also a state function over $\mathcal{V} \cup \mathcal{V}'$, where $\mathcal{V}'$ is any set of variables. For any state function f over $\mathcal{V}$ and any $\mathcal{V}$-state s (i.e., any assignment of proper values to all variables in $\mathcal{V}$), we let $s[\![f]\!]$ denote the value of f in state s.

A *state predicate* over $\mathcal{V}$ is a boolean valued state function over $\mathcal{V}$. Below we shall see that state predicates are a special case of a more general notion of *step formula*.

3.3 State Transition Functions

A *state transition function* f over $\mathcal{V}$ is a state function over $\mathcal{V} \cup \mathcal{V}^\circ$, where $\mathcal{V}^\circ$ is the set obtained by tagging each variable in $\mathcal{V}$ with $^\circ$. State transition functions over $\mathcal{V}$ are evaluated over pairs (s, s') of $\mathcal{V}$-states. The variables in $\mathcal{V}$ refer to state variables in s and variables in $\mathcal{V}^\circ$ refer to the corresponding state variables in s'. Formally, the value of a state transition function f over $\mathcal{V}$ in a pair s, s' of $\mathcal{V}$-states, written $(s, s')[\![f]\!]$, is defined as

$$(s, s')[\![f]\!] \;\triangleq\; (s \cup [x^\circ \mapsto s'(x) \mid x \in \mathcal{V}])[\![f]\!]$$

Action Functions and State Transition Predicates

An *action function* f over $(\mathcal{V}, \mathcal{A})$ is a state transition function over $\mathcal{V}$ that yields a subset of the actions in $\mathcal{A}$ when evaluated in any pair of $\mathcal{V}$-states. Note, that the stuttering action ζ can never be in the range of an action function.

A *state transition predicate* P over $\mathcal{V}$ is any boolean valued state transition function over $\mathcal{V}$.

3.4 Step Formulas

A *step formula* over $(\mathcal{V}, \mathcal{A})$ is a formula that can be evaluated over triples (s, a, s'), where s and s' are $\mathcal{V}$-states and $a \in \mathcal{A} \cup \{\zeta\}$, i.e., step formulas are evaluated over (possibly stuttering) steps.

There are two kinds of step formulas: those based on action functions and those based on state transition predicates. We consider these two possibilities and in each case we define what it means for a step formula P to *hold* in (s, a, s'), written $(s, a, s') \models P$.

If f is an action function over $(\mathcal{V}, \mathcal{A})$, then $\langle f \rangle$ is a step formula over $(\mathcal{V}, \mathcal{A})$, and we define

$$(s, a, s') \models \langle f \rangle \qquad \text{iff} \qquad a \in (s, s')[\![f]\!]$$

Since ζ can never be in the range of f, the step formula $\langle f \rangle$ can never hold in a stuttering step.

A state transition predicate P over $\mathcal{V}$ is also a step formula over $(\mathcal{V}, \mathcal{A})$, where $\mathcal{A}$ is an arbitrary set of actions, and we define

$$(s, a, s') \models P \qquad \text{iff} \qquad (s, s')[\![P]\!] = \mathit{true}$$

3.4.1 State Predicates

A state predicate P over $\mathcal{V}$ can now be seen as a special case of a step formula, namely a state transition predicate over $\mathcal{V}$ that does not mention any variables in $\mathcal{V}^\circ$. Thus, consistent with the normal semantics of state predicates, we define what it means for a state predicate P over $\mathcal{V}$ to *hold* in a $\mathcal{V}$-state s, written $s \models P$,

$$s \models P \qquad \text{iff} \qquad (s, s)[\![P]\!] = \mathit{true}$$

When defining temporal formulas below, we deal with step formulas and thereby also state predicates.

3.5 Temporal Formulas

An execution $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ over $(\mathcal{V}, \mathcal{A})$ is an execution where each s_i is a $\mathcal{V}$-state and each $a_i \in \mathcal{A} \cup \{\zeta\}$ such that if $a_i = \zeta$ then $s_{i-1} = s_i$. (Thus, stuttering actions can only occur in executions if they are part of stuttering steps.) Below we define the notion of *temporal formulas P over* $(\mathcal{V}, \mathcal{A})$, and what it means for such a formula to *hold* at position $j \in \mathsf{N}$ in an execution α over $(\mathcal{V}, \mathcal{A})$, written $(\alpha, j) \models P$. (If α is finite, it is thought of as being extended with stuttering such that we can also define what it means for P to hold at positions $j \geq |\alpha|$.)

A temporal formula over $(\mathcal{V}, \mathcal{A})$ contains only free variables in $\mathcal{V}$ and can only mention actions in $\mathcal{A}$. Thus, a temporal formula over $(\mathcal{V}, \mathcal{A})$ is also a temporal formula over $(\mathcal{V} \cup \mathcal{V}', \mathcal{A} \cup \mathcal{A}')$, where $\mathcal{V}'$ is any set of variables and $\mathcal{A}'$ is any set of actions.

Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ below.

Step Formulas

Any step formula P over $(\mathcal{V}, \mathcal{A})$ is also a temporal formula over $(\mathcal{V}, \mathcal{A})$ and we have,

$$\begin{array}{lll} (\alpha, j) \models P & \text{iff} & (0 \leq j < |\alpha| \quad \text{and} \quad (s_j, a_{j+1}, s_{j+1}) \models P) \text{ or} \\ & & (j \geq |\alpha| \quad \text{and} \quad (s_{|\alpha|}, \zeta, s_{|\alpha|}) \models P) \end{array}$$

Thus, for all positions j in α (except the last one if α is finite), P has to hold for the step starting in state s_j. If α is finite and j is greater than or equal to the last position in α, P has to hold for the step that stutters the last state.

The *Next* Operator

If P is a temporal formula over $(\mathcal{V}, \mathcal{A})$, then so is $\bigcirc P$, read *next* P.

$$(\alpha, j) \models \bigcirc P \qquad \text{iff} \qquad (\alpha, j+1) \models P$$

The *Unless* (*Waiting-for*) Operator

If P and Q are temporal formulas over $(\mathcal{V}, \mathcal{A})$, then so is $P \mathcal{W} Q$, read P *unless* (or *waiting-for*) Q.

$$(\alpha, j) \models P \mathcal{W} Q \quad \text{iff} \quad \begin{array}{l} \text{either there exists a } k \geq j \text{, such that } (\alpha, k) \models Q, \\ \qquad \text{and for every } i \text{ with } j \leq i < k,\ (\alpha, i) \models P, \\ \text{or else for all } i \text{ with } i \geq j,\ (\alpha, i) \models P \end{array}$$

Quantification

If P is a temporal formula over $(\mathcal{V}, \mathcal{A})$, then $(\forall x : P)$ and $(\exists x : P)$ are temporal formulas over $(\mathcal{V} \setminus \{x\}, \mathcal{A})$.

For any $\mathcal{V}$-state s denote by s_v^x, where v is assumed to be in the type of the variable x, the $(\mathcal{V} \cup \{x\})$-state obtained from s by either, if $x \in \mathcal{V}$, changing the value of x in s to v, or, if $x \notin \mathcal{V}$, extending s with a mapping from x to v. Thus, $s_v^x \triangleq (s \setminus \{x\}) \cup [x \mapsto v]$. For any execution $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ over $(\mathcal{V}, \mathcal{A})$, let α_v^x denote the execution $(s_0)_v^x a_1 (s_1)_v^x a_2 (s_2)_v^x \cdots$ over $(\mathcal{V} \cup \{x\}, \mathcal{A})$. With this definition, we can define the semantics of *universal quantification*.

$$(\alpha, j) \models \forall x : P \quad \text{iff} \quad \text{for all values } v,\ (\alpha_v^x, j) \models P$$

Thus, P must, for arbitrary (proper) values v, hold for the execution where x is assigned the value v in *every* state. This is in [MP92] and [Lam91] known as quantification over *rigid* variables since the variable has a constant value during the execution. In [MP92] and [Lam91] quantification over a *program* variable x allows x to vary during the execution. We do not consider that kind of quantification in this work.

Existential quantification is defined in a similar fashion.

$$(\alpha, j) \models \exists x : P \quad \text{iff} \quad \text{there exists a value } v \text{ such that } (\alpha_v^x, j) \models P$$

Boolean Operators

We give the standard definition of implication and negation. The remaning boolean operators will be derived from these below.

If P and Q are temporal formulas over $(\mathcal{V}, \mathcal{A})$, then so is $P \Longrightarrow Q$, and we have

$$(\alpha, j) \models (P \Longrightarrow Q) \quad \text{iff} \quad (\alpha, j) \models P \ \text{ implies that } \ (\alpha, j) \models Q$$

If P is a temporal formula over $(\mathcal{V}, \mathcal{A})$, then so is $\neg P$, and we have

$$(\alpha, j) \models \neg P \quad \text{iff} \quad (\alpha, j) \not\models P$$

Since we allow boolean operators in both state functions and temporal formulas, there might be an ambiguity as to how such boolean operators should be interpreted in a given temporal formula. For example, $R \triangleq \bigcirc(x = 1 \Longrightarrow y = 2)$ can be regarded as obtained by A) applying the *next* operator to the step formula $(x = 1 \Longrightarrow y = 2)$, or B) first applying the temporal *implies* operator to the two step formulas $x = 1$ and $y = 2$, and then applying the *next* operator to the result. It turns out that either interpretation leads to the same result as to whether the formula holds at a certain position in an execution. However, to avoid confusion we adopt the convention that step formulas in temporal formulas are always "as large as possible", thus, we consider R in the example to be produced as described in case A).

3.6 More Temporal Formulas

The rest of the temporal operators can be described syntactically from $\mathcal{W}$, $\Longrightarrow$ and $\neg$. Below we assume that P and Q are temporal formulas over $(\mathcal{V}, \mathcal{A})$. The formulas we define are then also temporal formulas over $(\mathcal{V}, \mathcal{A})$.

More Boolean Operators

Disjunction and conjunction are defined in the standard way.

$$P \vee Q \triangleq (\neg P) \Longrightarrow Q$$

$$P \wedge Q \triangleq \neg((\neg P) \vee (\neg Q))$$

The *Inclusive Unless* Operator

The $\mathcal{W}$ operator defined above requires a formula P to hold forever or, if another formula Q holds at some point, at least up to but not necessarily including the point where Q starts to holds. Often we need to express that P also holds in the state where Q starts to hold. For this reason we introduce the *inclusive unless* operator $\mathcal{W}_i$ defined as

$$P \, \mathcal{W}_i \, Q \triangleq P \, \mathcal{W} \, (P \wedge Q)$$

The *Always* Operator

To express that a formula holds forever, we define $\Box P$, read *always* P.

$$\Box P \triangleq P \, \mathcal{W} \, \mathit{false}$$

The *Eventually* Operator

To express that sooner or later a temporal formula holds, we define $\Diamond P$, read *eventually* P.

$$\Diamond P \triangleq \neg \Box (\neg P)$$

The (*Inclusive*) *Until* Operator

The unless operator expresses that a temporal formula P holds at least until another temporal formula Q starts to hold, but it does not require that Q eventually holds. (If Q does not hold eventually, P should hold forever). To express that Q is required to hold eventually, we define $P \, \mathcal{U} \, Q$, read P *until* Q.

$$P \, \mathcal{U} \, Q \triangleq (\Diamond Q) \wedge (P \, \mathcal{W} \, Q)$$

There is also an inclusive version of the until operator.

$$P \, \mathcal{U}_i \, Q \triangleq (\Diamond Q) \wedge (P \, \mathcal{W}_i \, Q)$$

The *Leads-To* Operator

The *leads-to* operator is an important temporal operator which expresses that during an execution, if P holds at some point, then Q will hold at a later (or the same) point. Thus, $P \leadsto Q$, read P *leads to* Q, is defined as

$$P \leadsto Q \triangleq \Box (P \Longrightarrow (\Diamond Q))$$

3.6.1 Precedence

To avoid excessive use of parentheses, we use the following convention regarding the precedence (binding power) of the temporal operators. The operators in the group

$$\bigcirc \quad \Box \quad \Diamond \quad \neg$$

have equal precedence but higher precedence than the operators

$$\wedge \quad \vee$$

which, in turn, have equal precedence but higher precedence than the operators

$$\Longrightarrow \quad \mathcal{W} \quad \mathcal{W}_i \quad \mathcal{U} \quad \mathcal{U}_i \quad \leadsto$$

which have equal precedence.

3.7 Functions and Temporal Formulas over Automata

For any safe (timed) I/O automata A whose state space is defined by state variables, denote by $\mathit{variables}(A)$ the set of state variables of A. We say that f is a state function or state transition function over A if f is a state function or state transition function over $\mathit{variables}(A)$, respectively. Similarly, f is said to be an action function over A if it is an action function over $(\mathit{variables}(A), \mathit{acts}(A))$. This notion trivially extends to step formulas and temporal formulas.

3.8 Satisfaction and Validity

An execution α over $(\mathcal{V}, \mathcal{A})$ is said to *satisfy* a temporal formula P over $(\mathcal{V}, \mathcal{A})$, written $\alpha \models P$, if and only if P holds at position 0 of α, thus

$$\alpha \models P \qquad \text{iff} \qquad (\alpha, 0) \models P$$

A temporal formula P over $(\mathcal{V}, \mathcal{A})$ is said to be *valid*, written $\models P$, if every execution α over $(\mathcal{V}, \mathcal{A})$ satisfies P, thus

$$\models P \qquad \text{iff} \qquad \text{for all } \alpha \text{ over } (\mathcal{V}, \mathcal{A}),\ \alpha \models P$$

We also introduce a notion of validity relative to a set E of executions over $(\mathcal{V}, \mathcal{A})$. A temporal formula P over $(\mathcal{V}, \mathcal{A})$ is then E-*valid*, written $E \models P$, if every execution of E satisfies P, thus

$$E \models P \qquad \text{iff} \qquad \text{for all } \alpha \in E,\ \alpha \models P$$

This notion extends to A-*validity*, where A is a safe (timed) I/O automaton. Then, for any temporal formula P over A, P is said to be A-*valid*, written $A \models P$, if every execution of A satisfies P, thus

$$A \models P \qquad \text{iff} \qquad \text{for all } \alpha \in \mathit{exec}(A),\ \alpha \models P$$

3.9 Finite vs. Infinite Executions

Above α has ranged over infinite as well as finite executions. In this section we prove that the question whether a temporal formula P holds at position j in execution α is equivalent to the question whether P holds at position j in $\widehat{\alpha}$. This result is, of course, due to the semantics of step formulas which has a special case dealing with stuttering steps.

Lemma 3.1

Let P be a temporal formula over $(\mathcal{V}, \mathcal{A})$. Then, for all executions α over $(\mathcal{V}, \mathcal{A})$ and all $j \geq 0$,

$$(\alpha, j) \models P \qquad \textit{iff} \qquad (\widehat{\alpha}, j) \models P$$

Proof

In Appendix B.

■

3.10 Stuttering-Insensitive Temporal Formulas

A temporal formula P over $(\mathcal{V}, \mathcal{A})$ is *stuttering-insensitive* if, for arbitrary executions α_1 and α_2 over $(\mathcal{V}, \mathcal{A})$ with $\alpha_1 \simeq \alpha_2$, $\alpha_1 \models P$ if and only if $\alpha_2 \models P$. Thus, if P is stuttering-insensitive and holds for α, it holds for all executions that can be obtained from α by adding or removing stuttering.

Below, in Proposition 3.4, we prove that certain types of temporal formulas are stuttering-insensitive. However, first we need two technical lemmas.

Lemma 3.2

Let P be a temporal formula over $(\mathcal{V}, \mathcal{A})$ and $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ an arbitrary infinite execution over $(\mathcal{V}, \mathcal{A})$. Then, for all $j \geq 0$ and all $i \leq j$

$$(\alpha, j) \models P \qquad \textit{iff} \qquad ({}_{j-i}|\alpha, i) \models P$$

Proof

In Appendix B.

■

Lemma 3.3

Let α and α' be infinite executions such that $\alpha \simeq \alpha'$. Then, for all $k \geq 0$, there exists a $k' \geq 0$ such that

1. ${}_{k}|\alpha \simeq {}_{k'}|\alpha'$
2. *for all $0 \leq i' < k'$, there exists an i with $0 \leq i < k$ such that ${}_{i}|\alpha \simeq {}_{i'}|\alpha'$*

Proof

In Appendix B.

■

We can now characterize certain temporal formulas which are stuttering-insensitive. State predicates are always stuttering-insensitive. This is because stuttering-equivalent executions will always start in the same state. General state transition predicates are not, however, stuttering-insensitive in general. This is due to the fact that stuttering-equivalent executions do not necessarily agree on the first step. All state transition predicates that hold in all stuttering steps are, however, stuttering-insensitive. Also, step formulas of the form $\langle f \rangle$ are not stuttering-insensitive, but $\Diamond\langle f \rangle$ is.

For the temporal operators, formulas of the form $\bigcirc P$ are not stuttering-insensitive in general. Assume for instance that $\alpha_1 = s_0 a_1 s_1 a_2 s_2 \cdots$ and $\alpha_1 = s_0 \zeta s_0 a_1 s_1 a_2 s_2 \cdots$. Then $\alpha_1 \simeq \alpha_2$. Assume that $(\alpha_1, j) \models P$ only if $j = 1$. Then $\alpha_1 \models P$ but $\alpha_2 \not\models P$. Thus, $\bigcirc P$ is not stuttering-insensitive. However, all other temporal operators yield stuttering-insensitive temporal formulas when applied to stuttering-insensitive formulas.

Proposition 3.4

1. *Every state predicate P is stuttering-insensitive.*
2. *If P is a state transition predicate such that for all states s, $(s, \zeta, s) \models P$, then P is stuttering-insensitive.*
3. *If f is an action function, then $\Diamond\langle f \rangle$ is stuttering-insensitive.*
4. *If P and Q are stuttering-insensitive, then*
   (a) $P \mathcal{W} Q$,
   (b) $\forall x : P$,
   (c) $\exists x : P$,
   (d) $\neg P$, *and*
   (e) $P \Longrightarrow Q$

   are all stuttering-insensitive.

Proof

In Appendix B.

■

3.11 Comparison with Manna and Pnueli's Temporal Logic

The temporal logic of Manna and Pnueli [MP92] is state based in the sense that temporal formulas are evaluated over sequences of states, i.e., with no actions interleaved. These sequences (*computations*) must be infinite; terminating computations are made infinite by appending infinite stuttering at the end.

As Lemma 3.1 indicates we could also have chosen to deal with infinite executions only: any temporal formula in our temporal logic is satisfied by a finite execution α if and only if the temporal formula is satisfied by the infinite execution obtained by appending infinite stuttering at the end of α. This indicates that the use of infinite computations only in [MP92] as opposed to our use of both finite and infinite executions is not an important difference between the two logics.

The real difference lies in the important role of actions in our logic. We need to be able to express properties of the actions occurring in executions. However, as the following discussion indicates, several results of [MP92] carry over to our logic.

Consider any (infinite) execution

$$\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$$

This execution can be encoded as the following state based computation:

$$\sigma = (s_0, a_1, s_1)(s_1, a_2, s_2) \cdots$$

Thus, each state of σ is a triple. Specifically, states of σ are assignments of the form:

$$\begin{array}{llll} [& x_1 & \mapsto & v_1, \\ & \ldots & & \\ & x_n & \mapsto & v_n, \\ & act & \mapsto & a, \\ & x'_1 & \mapsto & v'_1, \\ & \ldots & & \\ & x'_n & \mapsto & v'_n \quad] \end{array}$$

where the variable assignments to $x_1, \ldots, x_n$ represent the first state in a triple, the special variable act holds the action of the triple, and the variable assignments to $x'_1, \ldots, x'_n$ represent the last state in the triple.

Now, any valid temporal formula of [MP92] holds, in particular, for computations, where each state has the form (s, a, s') such that the last state of each triple coincides with the first pair of the next triple. Thus, valid formulas of [MP92] hold specifically for all computations that are encodings of our executions.

In order for such validity results of [MP92] to carry over to our temporal logic, it is important that the operators of [MP92] that we also use have a similar semantics in the two temporal logics, but this is easy to see. In fact, we have been guided by the temporal logic of [MP92] when defining the semantics of our temporal operators.

Note, that since our notion of execution in the encoding into computations is more restrictive than general computations, validities in our logic do not carry over to the temporal logic of [MP92].

3.12 Rules and Meta Rules

Temporal logics, or any logic for that matter, usually contain inference rules which allow validities to be inferred from other validities. This is however not the way we shall use our temporal logic in the verification examples in this work. Typically, we are given a particular execution α which satisfies a temporal formula P and then have to show that α satisfies another temporal formula Q. Thus, our proofs will be proofs of satisfaction as opposed to proofs of validity.

So, for our purpose inference rules are not very useful. Instead we shall use rules of the form of valid implications.

$$\models P \Longrightarrow Q$$

Such a rule (together with the definition of implication) allows us to conclude $\alpha \models Q$ from $\alpha \models P$.

We now present the rules that we use in our correctness proofs below. We do not present simple rule like, e.g., manipulation of Boolean operators or rules like

Par: $\models (\Box P) \Longrightarrow P$

but implicitly use such rules in our proofs. An approach like TLA [Lam91] has invested a lot of effort in finding rules that are typically used when proving systems correct. Such an investigation still needs to be done for our temporal logic. Thus, we present the rules we have found a need for in the particular examples presented in this work and leave the more general investigation for further research. We do not prove that the rules are actually validities but we note that this should follow easily from an encoding into the temporal logic of [MP92] as described in Section 3.11. In the rules we let $P(k)$ denote a formula with k free. Then, e.g., $P(0)$ is the formula obtained from $P(k)$ by replacing all free occurrences of k with 0.

MP: $\models (((P_1 \wedge \ldots \wedge P_k) \Longrightarrow Q) \wedge P_1 \wedge \cdots \wedge P_k) \Longrightarrow Q$

MP1: $\models (\Box(P \Longrightarrow Q) \wedge \Diamond P) \Longrightarrow \Diamond Q$

Pro1: $\models (\forall k : \exists k' : (k > k' \wedge P(k)) \leadsto P(k')) \Longrightarrow \Diamond P(0)$

Pro2: $\models (\Box(P \Longrightarrow (Q \mathcal{W} R)) \wedge (\Box Q \Longrightarrow \Diamond S) \wedge ((Q \wedge S) \leadsto R)) \Longrightarrow (P \leadsto R)$

Ind: $\models ((P(0) \leadsto Q) \wedge \forall k : (k > 0 \Longrightarrow \exists k' : (k' < k \wedge (P(k) \leadsto P(k') \vee Q)))) \Longrightarrow$
$\forall n : (P(n) \leadsto Q)$

Unl: $\models (\Box(P \Longrightarrow \neg Q) \wedge (P \mathcal{W}_i Q)) \Longrightarrow \Box P$

Unl1: $\models (\Box(P \Longrightarrow (Q \mathcal{W}_i R)) \wedge (\Box Q \Longrightarrow \Diamond S)) \Longrightarrow \Box(P \Longrightarrow (\Diamond R \vee \Box\Diamond S))$

The rules allow us to prove that a given execution satisfies a formula, provided it satisfies another formula. We shall be using other rules, called *meta rules*, which cannot be stated as validities. For instance, if $\alpha \models \Box P$ and α' is a suffix of α, then $\alpha' \models \Box P$. Again, we present the meta rules we have found useful in our particular examples, and leave an investigation of a "complete" set of meta rules as well as proofs of our meta rules for further research. We note, however, that many of the meta rules can be proved using Lemma 3.2.

Lemma 3.5

1. *If $\alpha \models \Box P$ and α' is a suffix of α, then $\alpha' \models \Box P$.*
2. *If, for all suffixes α' of α, $\alpha' \models P$, then $\alpha \models \Box P$.*
3. *If $\alpha \models \Diamond P$, then there exists a suffix α' of α such that $\alpha' \models P$.*
4. *If there exists a suffix α' of α and $\alpha' \models P$, then $\alpha \models \Diamond P$.*
5. *If, for any proper constant v, $\alpha \models P(v)$, then $\alpha \models \forall k : P(k)$.*
6. *If $\alpha \models \forall k : P$, then, for any proper constant v, $\alpha \models P(v)$.*
7. *If, for some proper constant v, $\alpha \models P(v)$, then $\alpha \models \exists k : P(k)$.*
8. *If $\alpha \models \exists k : P(k)$, then there exists a proper constant v such that $\alpha \models P(v)$.*

■

Since, in our proofs below, we shall use the different parts of Lemma 3.5 extensively, sometimes we use several parts at once and then simply refer to the lemma and not the particular parts.

This concludes the introduction to our temporal logic. The temporal logic is especially designed so that formulas are evaluated over executions of safe (timed) I/O automata. This allows us to use the temporal logic to specify liveness conditions of live (timed) I/O automata and use the rules of the temporal logic in correctness proofs. Exactly how we use the temporal logic for specifying liveness conditions is one of the issues of the next chapter.

Chapter 4

Specifying Systems

Chapter 2 introduced our basic models of timed and untimed systems. The models are entirely *semantic*: they describe the operational *meaning* of a system, that is, how a system behaves when executed.

A live I/O automaton consists of mathematical objects like sets and lists. However, these sets and lists may be infinite, which indicates that a direct enumeration is not feasible. Thus, we need a language or some *syntax*, other than standard mathematical notation, for writing down elements of our models. This chapter describes the syntax we use.

Furthermore, we describe how the effect of semantic operators (like parallel composition) is reflected in the syntax. For instance, we shall use the language of the temporal logic of Chapter 3 for specifying liveness conditions. We then show, e.g., that under certain circumstances if the liveness of two systems are described by temporal formulas Q_A and Q_B, respectively, then the liveness of the composed system is described by $Q_A \wedge Q_B$. This is important since it enables us to obtain a syntactic specification of the composed system directly from the specification of the component systems.

The rest of this chapter is organized as follows. We first, in Section 4.1, deal with untimed systems and then, in Section 4.2, show how timed systems can be specified. Finally Section 4.3 proves important embedding results.

4.1 Specifying Untimed Systems

4.1.1 Safe I/O Automata

Safe I/O automata will be specified using the *precondition-effect* style normally used for specifying the I/O automata of [LT87, LT89].

This style assumes that the state space of the safe I/O automaton is described as a mapping from state variable names to their values. Thus, the state space of a safe I/O automaton will be described by listing the state variable names together with their types. The start states of a safe I/O automaton are then specified by giving the possible values the state variables can assume initially.

As an example, consider the specification of a one-place buffer with the following functions: a message m can be placed in the buffer by the input action $send(m)$ and removed from the buffer by the output action $receiver(m)$. (The environment is thought of as *sending* messages to the buffer and *receiving* them from the buffer.) If a new message is sent to the buffer before the previous message is passed on to the receiver, a special *overflow* flag is set, which leads to an output action *overflow*. Initially the buffer is empty and the overflow flag is not set. Thus, the state space and start state of this safe I/O automaton is described as:

Variable	**Type**	**Initially**	**Description**
buf	$Msg \cup \{\perp\}$	$\perp$	The one-place buffer. The symbol $\perp$ denotes the empty buffer.
of	Bool	*false*	The overflow flag. A value of *true* denotes overflow.

We denote by $variables(A)$ the set of state variables of the safe I/O automaton A. We use the normal record-notation for referencing the values of state variables in a given state. For instance, the value of state variable *buf* in state s is denoted by $s.buf$. Formally, since s is a mapping from variables to values, we have $s.buf \triangleq s(buf)$.

The action signature of the one-place buffer is described as follows:

Input:
  $send(m)$, $m \in Msg$
Output:
  $receive(m)$, $m \in Msg$
  overflow
Internal:
  none

Thus, even though there might be infinitely many actions (*Msg* might be infinite), we use only finitely many *action generator functions* to describe these actions. (The action generator functions are assumed to be disjoint and their union to be injective).

It now only remains to show how to define the transition relation. Generally, for each action generator function we define one or more *step rules*. For example, in the case of the action generator function *send* above we might want to define two step rules based on some partition of the messages *Msg* into Msg_1 and Msg_2. Then one step rule would define steps labeled with actions from $\{send(m) \mid m \in Msg_1\}$, and the other would define steps labeled with actions from $\{send(m) \mid m \in Msg_2\}$. The sets Msg_1 and Msg_2 could even be overlapping, in which case we introduce nondeterminism of the *send* steps. A step rule has the form

$agf(x, y, \ldots)$
  Precondition:
    P
  Effect:
    E

where *agf* is an action generator function over the variables x, y, etc., P is a *precondition*, and E is an *effect clause*.

The precondition P is a state predicate over the state variables of the system and the variables x, y, etc.. A particular action, say $agf(1, 2, \ldots)$, is then enabled in state s, if P holds in s after replacing free occurrences of x with 1, free occurrences of y with 2, and so on.

The effect clause E uses a Pascal-like style of assignments. Thus, the effect clause consists of a list of assignments (one per line) of the form

$$v := e$$

where v is a state variable and e is an expression (state function)—of the same type as v—over the state variables and the variables x, y, etc.. Again, for a particular action $agf(1, 2, \ldots)$ we must replace free occurrences of x with 1, free occurrences of y with 2, and so on, in the expression e. If e' denotes this instantiated expression, then if s is the state before the assignment, the result of executing the assignment is the state s' obtained by changing the value of v to $s[\![e']\!]$. Thus, $s' \triangleq (s \setminus \{v\}) \cup [v \mapsto s[\![e']\!]]$. The result of executing a list of assignments

$assignment_1$
...
$assignment_n$

is obtained by first executing $assignment_1$, then $assignment_2$, and so on. Thus, the state will be changed in an sequential manner, but remember that this is just a convenient way of describing the post-state of the step, namely the state after the last assignment. In TLA [Lam91] the effects of steps are given by directly relating the values of the individual state variables in the pre- and post-states, but we have chosen this more program-like notation.

To make some assignments conditional we use an if-then-else construct. An example of such a construct is,

```
if P then
   assignment_1
   assignment_2
else
   assignment_3
   assignment_4
```

where P is a state predicate. The semantics is of course that if P holds when control has reached the if-statement, then assignments 1 and 2 are executed (in that order); otherwise assignments 3 and 4 are executed. Note, that we use indentation to indicate the end of the if-then-else construct. This means that

```
if P then
   assignment_1
   assignment_2
else
   assignment_3
assignment_4
```

is different from the previous if-then-else construct in that this construct first executes either assignments 1 and 2 or assignment 3 depending on the value of P, and then, unconditionally, executes assignment 4. We omit the else part of an if-then-else construct if it contains no assignments.

The format of the effect-clause described so far does not allow nondeterminism for a particular action. To specify such nondeterminism we will use *optional* assignments of the form

optionally $x := e$

with the meaning that nondeterministically either the assignment is or is not executed.

We could have been more formal in defining the syntax and semantics of assignments, etc., but since such syntax and semantics are standard, we have chosen to keep the exposition at a more intuitive level.

Finally, we note that step rules may contain variables which are not state variables or variables occurring in action generator functions. Such variables can be thought of as constants, and we then effectively defines a step rule for each proper value of the constant. An example is the following step rule, where n is such an extra variable.

$agf(x, y, \ldots)$
  Precondition:
    $\ldots \wedge 0 \leq n < 10$
  Effect:
    $\ldots$
    $[76\ v := x + n^2$
    $\ldots$

Safe I/O automata must be input-enabled (cf. Definition 2.1). This is ensured by omitting the preconditions for input actions. This has the same meaning as a precondition of *true*. The definition of the transition relation for the one-place buffer now looks like:

$send(m)$
  Effect:
    if $buf \neq \bot$ then
      $of := true$
    $buf := m$

$receive(m)$
  Precondition:
    $buf = m$
  Effect:
    $buf := \bot$

$overflow$
  Precondition:
    $of = true$
  Effect:
    $of := false$

An operational way to read such a definition is as follows. The definition for $send(m)$ says that if the buffer receives a new message m when buf is not empty, the overflow bif of is set. After that the new message is placed in buf (and a possible previous message will thus be overwritten). The one-place buffer can perform a $receive(m)$ step if m is the message in the buffer. The result is to empty the buffer. Finally, $overflow$ can be signaled if the overflow flag of is set, and the result is that of gets reset to $false$.

4.1.1.1 Operations on Safe I/O Automata

In Section 2.1.1 we defined the three operators (parallel composition, action hiding, and action renaming) on safe I/O automata. Below we explain how the safe I/O automata resulting from applying these operators can be described using syntax derived from the description of the safe I/O automata to which the operators were applied.

We start by considering parallel composition of safe I/O automata. In Definition 2.2, which defines parallel composition, we defined a notion of *compatibility* for safe I/O automata. This notion deals with guaranteeing that each action in a composed system be controlled by at most one component and that internal actions be unique. Definition 2.2 also says that the state space of a composed system is the cartesian product of the component state spaces. This means that if we want to reference the value of a certain state variable of one component, we first have to extract the state of the component from the total state. This becomes even more cumbersome if several levels of parallel composition are used. In order to avoid dealing with these not very interesting details of extracting component states of component states, etc., we will extend the notion of compatibility to also include the requirement that the sets of state variables of the component systems be disjoint. In this way a state s of the composed system can be uniquely described by an assignment of values to the total set of state variables in the system such that the value of any state variable x in s agrees with the value of x in the state of the component to which x belongs. (More precisely, such a "flat" assignment of values to state variables is isomorphic to the state define by the parallel composition operator in Chapter 2.) Thus, if s_i describes the state of the ith component as a mapping from state variables of this component to their values, the state of the composed system is described by the mapping $s_1 \cup \cdots \cup s_N$.

Thus, below we shall use the following definition of compatibility (cf. Definition 2.2): Safe I/O automata $A_1, \ldots, A_N$ are *syntactically compatible* if for all $1 \leq i, j \leq N$ with $i \neq j$

1. $out(A_i) \cap out(A_j) = \emptyset$
2. $int(A_i) \cap acts(A_j) = \emptyset$
3. $variables(A_i) \cap variables(A_j) = \emptyset$.

Note that the first two conditions have not changed. Below we let "compatibility" refer to "syntactical compatibility".

This notion of compatibility trivially extends to live I/O automata (cf. Definition 2.9). A consequence of this way of looking at the state space of a composed system is that for compatible safe I/O automata $A_1, \ldots, A_N$, the set of state variables of $A = A_1 \| \cdots \| A_N$ is given by $variables(A) = variables(A_1) \cup \cdots \cup variables(A_N)$.

Thus, the state variables (together with types and initial values) of a composed system can be described by writing the lists of state variables for the components one below the other. In a similar fashion it is easy to list the action signature of the composed system.

The question is, how can the description of the steps of the composed system be derived from the description of the steps of the components? Remember, from Definition 2.2, that in each step of the composed system several components might participate (each executing state changes described locally for the action of that step) whereas all other components do not change their state. Also remember, that the action of the step is locally-controlled by at most one component. That is, either the action is an input action for all participating components, or it is locally-controlled by one component and an input action for the remaining participating components. Then, if the step rules for $send(m)$ in three components, one of which controls the actions, are described by

$send(m)$	$send(m)$	$send(m)$
Precondition:	Effect:	Effect:
P_1	E_2	E_3
Effect:		
E_1		

then the $send(m)$ steps of the composed system can be described by

$send(m)$
  Precondition:
    P_1
  Effect:
    E_1
    E_2
    E_3

Note, that the order of the three effect clauses is unimportant since E_1, E_2, and E_3 mention disjoint sets of state variables.

Since the construction of the step rules of the composed system is so simple, we usually omit the explicit construction and instead refer to the step rules of the components.

For action hiding the situation is much simpler (cf. Definition 2.3). If, for instance, A is a safe I/O automaton and $\mathcal{A}$ is a set of locally-controlled actions of A, the syntactic description of $A \setminus \mathcal{A}$ is obtained from the syntactic description of A by simply moving the action generator functions describing output actions in $\mathcal{A}$ from the list of action generator function describing output actions to the list of action generator functions describing internal actions. Of course, if only some of the actions described by an action generator function are hidden, the action generator function will have to be split. For example, if $\mathit{send\text{-}nat}(i)$, where $i \in \mathsf{N}$, is an action generator function for output actions of A, and $\mathcal{A} = \{\mathit{send\text{-}nat}(i) \mid i \geq 100\}$, then $\mathit{send\text{-}nat}(i)$, $0 \leq i < 99$, will be in the listing of output actions of $A \setminus \mathcal{A}$ and $\mathit{send\text{-}nat}(i)$, $i \geq 100$, will be in the listing of internal actions of $A \setminus \mathcal{A}$.

Finally, for action renaming we use mappings of the form $[\mathit{send}(m) \mapsto \mathit{send\text{-}message}(m) \mid m \in \mathit{Msg}] \cup \cdots$, where, intuitively, each entire action generator function is being renamed. In this case each action generator function is simply replaced according to the action mapping in the syntactic descriptions of the action signature and the steps.

In the remainder of this work we shall assume that the syntactic changes to safe (timed) I/O automata reflecting semantic operations on these are well understood and concentrate on the more interesting aspects of defining liveness.

4.1.2 Live I/O Automata

We specify a liveness condition L for a safe I/O automaton A indirectly in terms of a temporal formula Q over A in the following way:

$$L = \{\alpha \in \mathit{exec}(A) \mid \alpha \models Q\} \tag{4.1}$$

That is, the liveness condition L consists of all the executions of A that satisfy a certain temporal formula Q. Of course, we have to make sure that what we define is in fact a liveness condition for A, i.e., we must make sure that any finite execution of A can be extended to an execution in L. We shall refer to any temporal formula Q over A that defines a liveness condition L for A as a *liveness formula* for A. Moreover, we call the liveness formula *environment-free* for A if (A, L) is environment-free and thus is a live I/O automaton.

Given a liveness formula Q for A, we shall refer to the liveness condition defined by (4.1) as the liveness condition for A *induced* by Q.

4.1.2.1 Operations on Live I/O Automata

In Section 2.1.2 we defined the three operators (parallel composition, action hiding, and action renaming) on live I/O automata. If our approach with specifying liveness using temporal formulas should have any practical relevance, it is important that the environment-free liveness formulas inducing the liveness conditions for the resulting live I/O automata can be obtained directly from the environment-free liveness formulas for the original live I/O automata.

This section proves that this is the fact given a few restrictions. As always we start by the result for parallel composition, which requires three preliminary lemmas the first of which embodies the complexity of the proof.

To help us state and prove the results below, we first define a notion of restriction of an execution over $(\mathcal{V}, \mathcal{A})$ to $(\mathcal{V}', \mathcal{A}')$. This notion is not similar to the notion of projection of executions to automata as defined in Chapter 2 since it introduces stuttering steps for actions not in $\mathcal{A}'$, whereas the definition in Chapter 2 simply removes such steps. Below we shall, however, see how the two notions are related.

For any $\mathcal{V}$-state s, $s \restriction \mathcal{V}'$, where $\mathcal{V}' \subseteq \mathcal{V}$, is the $\mathcal{V}'$-state obtained from the mapping s by restricting the domain to $\mathcal{V}'$.

Then, for any execution α over $(\mathcal{V}, \mathcal{A})$, define $\alpha \restriction (\mathcal{V}', \mathcal{A}')$, where $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{A}' \subseteq \mathcal{A}$, to be the execution over $(\mathcal{V}', \mathcal{A}')$ obtained from α by replacing each state s in α with $s \restriction \mathcal{V}'$ and replacing each action $a \notin \mathcal{A}'$ with ζ.

Lemma 4.1

Let P be a temporal formula over $(\mathcal{V}', \mathcal{A}')$. Then, for all pairs $(\mathcal{V}, \mathcal{A})$ with $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{A}' \subseteq \mathcal{A}$, all executions α over $(\mathcal{V}, \mathcal{A})$, and all $j \geq 0$,

$$(\alpha \restriction (\mathcal{V}', \mathcal{A}'), j) \models P \qquad \textit{iff} \qquad (\alpha, j) \models P$$

Proof

In Appendix B.

■

We now give an alternative characterization of the projection operator $\lceil$ on executions defined in Section 2.1.1. For any execution α of a safe I/O automaton $A_1 \| \cdots \| A_N$, define

$$\alpha \restriction A_i \;\triangleq\; \alpha \restriction (\mathit{variables}(A_i), \mathit{acts}(A_i))$$

Then $\alpha \lceil A_i = \natural(\alpha \restriction A_i)$ and clearly we have $\alpha \lceil A_i \simeq \alpha \restriction A_i$.

The following lemma is now a direct consequence of Lemma 4.1.

Lemma 4.2

Let $A_1, \ldots, A_N$ be compatible safe I/O automata and let $Q_1, \ldots, Q_N$ be temporal formulas over $A_1, \ldots, A_N$, respectively. Furthermore, let $A = A_1 \| \cdots \| A_N$ and $\alpha \in \mathit{exec}(A)$. Then, for all $1 \leq i \leq N$ and all $j \geq 0$,

$$(\alpha \restriction A_i, j) \models Q_i \qquad \textit{iff} \qquad (\alpha, j) \models Q_i$$

Proof

Since α is an execution over $(\mathit{variables}(A), \mathit{acts}(A))$ and each Q_i is a temporal formula over $(\mathit{variables}(A_i), \mathit{acts}(A_i))$ with $\mathit{variables}(A_i) \subseteq \mathit{variables}(A)$ and $\mathit{acts}(A_i) \subseteq \mathit{acts}(A)$, the result follows directly from Lemma 4.1 and the definition of $\alpha \restriction A_i$.

■

Lemma 4.3

Let $A_1, \ldots, A_N$ be compatible safe I/O automata and let $Q_1, \ldots, Q_N$ be stuttering-insensitive temporal formulas over $A_1, \ldots, A_N$, respectively. Let $A = A_1 \| \cdots \| A_N$ and $\alpha \in \mathit{exec}(A)$. Then,

$$\alpha \lceil A_1 \models Q_1 \quad \textit{and} \quad \cdots \quad \textit{and} \quad \alpha \lceil A_N \models Q_N \qquad \textit{iff} \qquad \alpha \models Q_1 \wedge \ldots \wedge Q_N$$

Proof

In Appendix B.

■

The following important result for parallel composition can now be proved.

Proposition 4.4

Let $(A_1, L_1), \ldots, (A_N, L_N)$ be compatible live I/O automata and let $Q_1, \ldots, Q_N$ be stuttering-insensitive temporal formulas over $A_1, \ldots, A_N$, respectively, such that each L_i is induced by Q_i. Let $(A, L) = (A_1, L_1)\| \cdots \|(A_N, L_N)$. Then L is induced by $Q_1 \wedge \ldots \wedge Q_N$.

Proof

In Appendix B.

■

It is important to understand the role that stuttering-insensitivity plays in the proposition. In the execution of a composed system, each step represents activity in a certain subset of the components while all other components do not engage in the step at all. When projecting the execution to any component, such steps where the component does not engage (i.e., stuttering steps) are simply removed. Thus, when specifying the liveness for a component system (A_i, L_i), we might write $Q_i = \Diamond\Box(x^\circ = x + 1)$ and hence specify that in any live execution (of (A_i, L_i)) there must be an infinite suffix where x is incremented by one at each step. Now, in a live execution α of the composed system, even though $\alpha\lceil A_i$ satisfies Q_i, α itself does not necessarily satisfy Q_i since steps performed by other components might result in x being incremented only in, e.g., every other step (but still, of course, incremented in every step where A_i engages). In the proposition we solve the problem by simply ruling out Q_i since it is not stuttering-insensitive. However, in the example we might write the following stuttering-insensitive liveness condition which captures the same idea: $Q'_i = \Box\Diamond\langle acts(A_i)\rangle \wedge \Diamond\Box(\langle acts(A_i)\rangle \Longrightarrow (x^\circ = x + 1))$. Thus, Q'_i describes that there is a suffix, with infinite activity of A_i, such that every time A_i engages, x is incremented.

Attention is now turned to the simpler operations of action hiding and action renaming.

Proposition 4.5

Let (A, L) be a live I/O automaton such that L is induced by the temporal formula Q for A and let $\mathcal{A} \subseteq local(A)$. Then the liveness condition of $(A, L) \setminus \mathcal{A}$ is induced by Q.

Proof

In Appendix B.

■

Proposition 4.6

Let (A, L) be a live I/O automaton such that L is induced by the temporal formula Q for A, and let ρ be an action mapping applicable to (A, L). Define $\rho(Q)$ to be the temporal formula obtained by applying ρ to every action function in Q. Then the liveness condition of $\rho((A, L))$ is induced by $\rho(Q)$.

Proof

In Appendix B.

■

4.1.2.2 Fairness

Fairness is a special form of liveness, where the requirement is that each component of the system be given fair turns. Fairness is important since it in most cases is environment-free, and furthermore fairness is easy to implement on a physical system. Traditionally, two different kinds of fairness are considered: *weak* and *strong* fairness.

Weak fairness to a system component or, as we shall phrase it, to the set of actions representing this component says that actions from the set cannot be enabled indefinitely without being executed infinitely often. Thus, for a safe I/O automaton A and a set $C \subseteq acts(A)$, weak fairness to C can be expressed as the temporal formula

$$WF_A(C) \triangleq \Box\Diamond\langle C\rangle \vee \Box\Diamond\neg enabled_A(C) \tag{4.2}$$

where $enabled_A(C)$ is a state predicate over A that holds in exactly the states of A where an action in C is enabled. As usual we omit the subscript A and write $WF(C)$ and $enabled(C)$ when A is clear.

We have in this work found it useful to use a slight variant of weak fairness in which actions are only forced to occur if they are enabled indefinitely *and* a special *forcing condition* is satisfied indefinitely. This can be formalized as

$$WF(C, P) \triangleq \Box\Diamond\langle C\rangle \vee \Box\Diamond\neg(enabled(C) \wedge P) \tag{4.3}$$

where P is a state predicate (the forcing condition). When using this variant of weak fairness, it is possible to separate the issues of when actions *may* occur (are enabled) and when they *must* occur.

Strong fairness says that actions from a set must be executed infinitely often if actions from the set are enabled infinitely often. In other words, we cannot ignore the actions forever if we are given infinitely many chances to execute them.

$$SF(C) \triangleq \Box\Diamond\langle C\rangle \vee \Diamond\Box\neg enabled(C) \tag{4.4}$$

Again, with a forcing condition this looks like

$$SF(C, P) \triangleq \Box\Diamond\langle C\rangle \vee \Diamond\Box\neg(enabled(C) \wedge P) \tag{4.5}$$

It is easy to see that temporal formulas of the form $WF(C)$, $WF(C,P)$, $SF(C)$, or $SF(C,P)$, where $C \subseteq acts(A)$ and P is a state predicate over A, are liveness formulas for A. But are they environment-free? First of all environment-freedom must require that C consist of only locally-controlled actions since otherwise we could be restricting the environment to perform certain input actions. This condition turns out to be sufficient for weak fairness to be environment-free. However, there is a problem with strong fairness as illustrated by the following example: Let L be induced by the strong fairness formula $SF(C)$ for A, where $C \subseteq local(A)$. Then, for any infinite execution α in L it is the case that if C is enabled in infinitely many states in α, then α contains infinitely many actions from C. Now suppose, in the game between system and environment, that each environment move consists of two input actions: one that is bound to enable C and one that is bound to disable C (thus no g function of a strategy can be defined to avoid that

C is enabled between the input actions and disabled afterwards). In this situation no strategy function f can be defined that can ever execute an action in C during such a game; in other words, every time the system gets a chance to move, it is not possible to execute an action in C since C is not enabled. Thus, any strategy defined on A will, when playing against this villainous environment, generate an outcome in which C is infinitely often enabled (namely between the two input actions of every environment move) but in which only finitely many C actions are executed. Thus the outcome is not live and it follows that $SF(C)$ is not environment-free.

However, strong fairness is environment-free if the safe I/O automaton in question is *C-persistent*, where $C \subseteq local(A)$. Define A to be C-persistent if for each state s of A in which C is enabled and each step (s, a, s') where $a \in in(A)$, C is enabled in s'. Thus, in any execution of A, if C becomes enabled, C will stay enabled at least until a locally-controlled action has been executed.

Lemma 4.7

Let A be a safe I/O automaton and let Q_i, $1 \leq i \leq k$, be temporal formulas over A of the form $WF(C_i)$, $WF(C_i, P_i)$, $SF(C_i)$, or $SF(C_i, P_i)$, where

- *$C_i \subseteq local(A)$,*
- *P_i is a state predicate over A, and*
- *if $Q_i = SF(C_i)$ or $Q_i = SF(C_i, P_i)$, then A is C_i-persistent.*

Then $Q_1 \wedge \cdots \wedge Q_k$ is an environment-free liveness formula for A.

Proof

This proof can be carried out similarly to the proof of Lamport and Abadi's Proposition 4 in [AL92b]. (Note that [GSSL93] argues that Lamport and Abadi's notion of μ-machine-realizability is similar to our notion of environment-freedom. Furthermore, μ-invariance is similar to our notion of C-persistence.)

■

Another important property of the fairness formulas is that they are stuttering-insensitive as expressed by the following lemma.

Lemma 4.8

Any conjunction of temporal formulas of the form $WF(C)$, $WF(C, P)$, $SF(C)$, and $SF(C, P)$ is stuttering-insensitive.

Proof

Directly by the definition of the fairness formulas and Proposition 3.4.

■

4.2 Specifying Timed Systems

We now turn attention to timed systems. As above we first describe how to specify safe timed I/O automata, and then how to use our temporal logic to specify liveness.

4.2.1 Safe Timed I/O Automata

In this work we use two approaches for specifying safe timed I/O automata: *explicit* and *implicit* specification. Both approaches describe state spaces using state variables as in the untimed setting. The definition of safe timed I/O automata (Definition 2.17) describes that the time can be obtained from any state by the $.now$ mapping. Below we assume that

> each safe timed I/O automaton has a special *now* state variable such that the $.now$ mapping simply returns the value of this state variable.

(We will not be able to see if $s.now$ means the value of the *now* state variable in state s or the result of applying the $.now$ mapping to state s, but since, by definition, both interpretations return the same time, this does not give rise to ambiguity.)

We denote by $variables(A)$ the set of state variables (including *now*) of the safe timed I/O automaton A. With this definition we can extend the definition of compatibility for safe timed I/O automata (cf. Definition 2.18) by requiring the state variables of the safe timed I/O automata be almost mutually disjoint. (They sets of state variables must only have *now* in common): Safe timed I/O automata $A_1, \ldots, A_N$ are *syntactically compatible* if for all $1 \leq i, j \leq N$ with $i \neq j$

1. $out(A_i) \cap out(A_j) = \emptyset$
2. $int(A_i) \cap acts(A_j) = \emptyset$
3. $variables(A_i) \cap variables(A_j) = \{now\}$

As in the untimed setting we use, for brevity, the term "compatibility" to refer to syntactical compatibility. The notion of compatibility trivially extends to *live* timed I/O automata (cf. Definition 2.26). As in the untimed setting we can now characterize the state of a composed safe timed I/O automaton $A = A_1 \| \cdots \| A_N$ by a "flat" mapping from $variables(A_1) \cup \cdots \cup variables(A_N)$ (i.e., $variables(A)$) to values such that s is the state of A if $s \restriction variables(A_i)$ is the state the component A_i. This characterization is possible since all components must agree on real time (cf. Definition 2.18).

Explicit Specification

The explicit approach to specifying safe timed I/O automata is similar to our way of specifying safe I/O automata: the state space and initial states are specified by a list of typed state variables with possible initial values (the *now* variable must assume the value 0 initially), the action signature is specified by using action generator functions to list input, output, and internal actions and the special time-passage action ν, and the steps are specified using the precondition-effect style.

Some of the state variables will typically be used to keep track of deadlines etc. Also, when specifying the steps using this explicit approach, the time-passage steps will have to be specified explicitly. The precondition for the time-passage steps will usually state that time is not allowed to pass beyond some deadlines representing times by which some other steps must have been executed.

It must be proved that what we specify is in fact a safe timed I/O automaton (cf. Definition 2.1). The axioms **S1**–**S3** are easy to ensure: **S1** is ensured by initializing *now* to 0, **S2** is ensured by leaving *now* unchanged in the step rules for visible and internal actions, and **S3** is ensured by requiring, in the step rule for ν, that time will increase. **S4** and **S5** are ensured if time-passage steps change the *now* variable only and, from any time, time-passage steps to any future time, possibly less than some deadline, is allowed.

As in the untimed setting it is easy to construct the syntactic description of a safe time I/O automaton from the syntactic description of its components. The only difference compared to the untimed setting is constructing the step-rule for ν when dealing with the parallel composition operator. In this case the preconditions of the step-rules for ν have to be combined so that all components allow the assignment to the (common) *now* variable. This turns out not to be a problem in practice.

In some situations it is possible to avoid dealing explicitly with deadlines and time-passing when specifying safe timed I/O automata. This approach is described next.

Implicit Specification

In [MMT91] and [LA91] alternative models for timed systems are developed. We will refer to these models by "MMT-models" derived from the names of the authors of [MMT91]. As shown in [GSSL93] the model we use is a generalization of the MMT-models.

In the MMT-models the locally-controlled actions are partitioned into classes and each class has associated with it a lower and upper time bound that represent the maximum and minimum delay of the system when executing these actions.

While these models are sufficient for the specification of many timed distributed systems, they are not sufficient for all the examples presented later in this work. However, because the MMT-models handle time implicitly, they tend to be easier to understand.

Instead of developing a theory for MMT-models, we will merely, whenever possible, use the style of these models as a convenient way of specifying our safe timed I/O automata. So below we define a notion of *MMT-specification* and show what such a specification denotes in the model of safe timed I/O automata.

Definition 4.9 (MMT-Specification)

An *MMT-specification* A_{MMT} is a triple where

- $automaton(A_{MMT})$ is a safe I/O automaton,
- $sets(A_{MMT})$ is a collection $C_1, \ldots, C_k$ of disjoint sets of locally-controlled actions of the safe I/O automaton $automaton(A_{MMT})$, and
- $boundmap(A_{MMT})$ is a mapping that to each $C_i \in sets(A_{MMT})$ associates a lower time bound $b_l(C_i) \in \mathsf{T}$ and an upper time bound $b_u(C_i) \in (\mathsf{T} \setminus \{0\}) \cup \{\infty\}$, such that $b_u(C_i) \geq b_l(C_i)$.

■

We let $states(A_{MMT})$, etc., refer to the corresponding components of the underlying safe I/O automaton $automaton(A_{MMT})$.

The intuition behind an MMT-specification is as follows: Let the triple (A, S, b) be an MMT-specification. A itself contains no information about time but we will now "execute" it in a world that has a notion of real time and *now*. Suppose during execution that a set $C_i \in S$ becomes enabled at time t. Then b specifies that if C_i stays enabled, then an action from C_i must be executed in the time interval $[t + b_l(C_i), t + b_u(C_i)]$. Thus, the boundmap specifies the time interval (relative to t) in which an action from C_i must be executed, unless C_i becomes enabled in the meantime. The same has to hold for C_i if it stays enabled after being executed; thus, in this case a new legal interval is calculated based on the current time, $b_l(C_i)$, and $b_u(C_i)$. If C_i becomes disabled, the timing constraints on C_i are removed.

To encode this idea into the model of safe timed I/O automata, we need to add several state variables. For instance we need to add the variable *now* representing real time, and for each of the sets C_i we need to add two variables: $\mathit{first}(C_i)$ and $\mathit{last}(C_i)$ to denote the first and last (absolute) times at which an action from C_i must be executed. In the encoding in our model, the *first* and *last* variables should then be set to the proper interval when the associated set C_i becomes (re-)enabled and reset to "no timing constraints" (i.e., the interval $[0, \infty]$) when C_i becomes disabled. Furthermore, actions in C_i are only allowed to be executed if real time has passed beyond $\mathit{first}(C_i)$. Additional time-passage steps also need to be added. These steps should only change *now* and are not allowed to let time pass beyond any of the *last* bounds. This idea is now formalized.

Definition 4.10

Let A_{MMT} be an MMT-specification. Then $\mathit{time}(A_{MMT})$ is the safe timed I/O automaton A for which

- each state s of $\mathit{states}(A)$ consists of a state $s.\mathit{basic}$, which is a state of A_{MMT}, augmented with a new state variable *now* and, for each set C_i of $\mathit{sets}(A_{MMT})$, two new state variables $\mathit{first}(C_i)$ and $\mathit{last}(C_i)$.
- $\mathit{start}(A)$ consists of states s for which $s.\mathit{basic}$ is a start state of A_{MMT}, $s.\mathit{now} = 0$, and, for each set C_i of $\mathit{sets}(A_{MMT})$, if C_i is enabled in $s.\mathit{basic}$ then $\mathit{first}(C_i) = b_l(C_i)$ and $\mathit{last}(C_i) = b_u(C_i)$; otherwise, $\mathit{first}(C_i) = 0$ and $\mathit{last}(C_i) = \infty$.
- $(\mathit{in}(A), \mathit{out}(A), \mathit{int}(A)) = (\mathit{in}(A_{MMT}), \mathit{out}(A_{MMT}), \mathit{int}(A_{MMT}))$.
- $\mathit{ext}(A) = \mathit{ext}(A_{MMT}) \cup \{\nu\}$.
- $(s, a, s') \in \mathit{steps}(A)$ iff the following conditions hold:
  1. If $a \in \mathit{acts}(A_{MMT})$ then
     (a) $s'.\mathit{now} = s.\mathit{now}$.
     (b) $(s.\mathit{basic}, a, s'.\mathit{basic}) \in \mathit{steps}(A_{MMT})$.
     (c) For each $C_i \in \mathit{sets}(A_{MMT})$:
        i. If $a \in C_i$ then $s.\mathit{first}(C_i) \leq s.\mathit{now}$.
        ii. If C_i is enabled in both $s.\mathit{basic}$ and $s'.\mathit{basic}$, and $a \notin C_i$, then $s'.\mathit{first}(C_i) = s.\mathit{first}(C_i)$ and $s'.\mathit{last}(C_i) = s.\mathit{last}(C_i)$.
        iii. If C_i is enabled in $s'.\mathit{basic}$ and either $a \in C_i$ or C_i is not enabled in $s.\mathit{basic}$, then $s'.\mathit{first}(C_i) = s'.\mathit{now} + b_l(C_i)$ and $s'.\mathit{last}(C_i) = s'.\mathit{now} + b_u(C_i)$.
        iv. If C_i is not enabled in $s'.\mathit{basic}$ then $s'.\mathit{first}(C_i) = 0$ and $s'.\mathit{last}(C_i) = \infty$.
  2. If $a = \nu$ then
     (a) $s'.\mathit{now} > s.\mathit{now}$.
     (b) $s'.\mathit{basic} = s.\mathit{basic}$.

(c) $s'.now \leq s'.last(C_i)$ for all $C_i \in sets(A_{MMT})$.
(d) $s'.first(C_i) = s.first(C_i)$ and $s'.last(C_i) = s.last(C_i)$ for all $C_i \in sets(A_{MMT})$.

■

It is easy to see that $time(A_{MMT})$ is in fact a safe timed I/O automaton (cf. Definition 2.17). Specifically, axiom **S1** is ensured since *now* is initialized to 0, **S2** is ensured since, by explicit construction, *now* does not change in steps labeled by visible or internal actions, **S3** is ensured since time-passage steps are explicitly required to increase time, and finally **S4** and **S5** are easily seen to be ensured since $time(A_{MMT})$ from any time allows time-passage to any future time less than some deadline (expressed by the *last* variables) and time-passage steps do not change the basic part of the state.

When using the implicit approach to specifying safe timed I/O automata, we use the precondition-effect style of Section 4.1.1 to specify the underlying safe I/O automaton, and then use standard notation (cf. Appendix A) to specify the sets of locally-controlled actions and the boundmap. Based on the simple way the new variables (*now* and the *first* and *last* variables) are manipulated, it is easy to construct an explicit description of $time(A_{MMT})$ based on the description of A_{MMT}.

We refer to Chapter 10 for an example of the implicit style of specification.

4.2.2 Live Timed I/O Automata

If we were to follow the lines of the untimed section when specifying the liveness condition for a safe timed I/O automaton, we should devise some temporal logic in which formulas were evaluated over *timed* executions. However, we take a different approach. The idea is that a timed execution can be characterized by a set of (ordinary) executions each of which can be thought of as a *sampling* of the timed execution. Thus, there exists a close relationship between timed executions and (ordinary) executions of a safe timed I/O automaton.

We proceed by defining the notion of sampling. Then we define what constitutes a sampling characterization of a liveness condition, show how the operations on live timed I/O automata are reflected in the syntax describing the liveness of the live timed I/O automata, and finally discuss the notions of weak and strong fairness in the timed setting.

4.2.2.1 Sampling

All definitions and lemmas in this section are taken from [GSSL93] and are similar to those of [LV93b].

Roughly speaking, an (ordinary) execution fragment can be regarded as "sampling" the state information in a timed execution fragment at a countable number of points in time. Formally, we say that an execution fragment $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ of A *samples* a timed execution fragment $\Sigma = \omega_0 b_1 \omega_1 b_2 \omega_2 \cdots$ of A if there is a monotone increasing mapping $f : \mathsf{N} \to \mathsf{N}$ such that the following conditions are satisfied.

1. $f(0) = 0$,
2. $b_i = a_{f(i)}$ for all $i \geq 1$,
3. $a_j = \nu$ for all j not in the range of f,

4. For all $i \geq 0$ such that ω_i is not the last trajectory in Σ,
   (a) $s_j \in rng(\omega_i)$ for all j, $f(i) \leq j < f(i+1)$,
   (b) $s_{f(i)}.now = ftime(\omega_i)$, and
   (c) $s_{f(i+1)-1}.now = ltime(\omega_i)$.
5. If ω_i is the last trajectory in Σ, then
   (a) $s_j \in rng(\omega_i)$ for all j, $f(i) \leq j$,
   (b) $s_{f(i)}.now = ftime(\omega_i)$, and
   (c) $sup\{s_j.now \mid f(i) \leq j\} = ltime(\omega_i)$.

In other words, the function f in this definition maps the (indices of) actions in Σ to corresponding (indices of) actions in α, in such a way that exactly the non-time-passage actions of α are included in the range. Condition 4 is a consistency condition relating the first and last times for each non-final trajectory to the times produced by the appropriate steps of α. Condition 5 gives a similar consistency condition for the first time of the final trajectory (if any); in place of the consistency condition for the last time, there is a "cofinality" condition asserting that the times grow to the same limit in both executions.

The following two straightforward lemmas show the relationship between timed execution fragments and ordinary execution fragments.

Lemma 4.11

Let A be a safe timed I/O automaton. If $\alpha \in frag(A)$, then there is a timed execution fragment $\Sigma \in t\text{-}frag(A)$ such that α samples Σ.

■

Lemma 4.12

Let A be a safe timed I/O automaton. If $\Sigma \in t\text{-}frag(A)$, then there is an execution fragment $\alpha \in frag(A)$ such that α samples Σ.

■

Recall that an execution fragment α is *finite* if it is a finite sequence. Furthermore, in the timed setting, an execution fragment α is defined to be *admissible* if there is no finite upper bound on the $.now$ values of the states in α. Finally, an execution fragment is said to be *Zeno* if it is neither finite nor admissible. We denote by $exec^*(A)$, $exec^\infty(A)$, and $exec^Z(A)$ the sets of finite, admissible, and Zeno executions of a safe timed I/O automaton A.

Lemma 4.13

If α samples Σ then

1. *α is finite iff Σ is finite,*
2. *α is admissible iff Σ is admissible, and*
3. *α is Zeno iff Σ is Zeno.*

■

It is possible to give a sensible definition of the timed trace of an ordinary execution fragment of a safe timed I/O automaton. Namely, suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of a safe timed I/O automaton A. First, define $ltime(\alpha)$ to be the supremum of the $.now$ values of all the states in α. Then let δ be the sequence consisting of the actions in α paired with their times of occurrence:

$$\delta = (a_1, s_1.now)(a_2, s_2.now) \cdots.$$

Then $t\text{-}trace(\alpha)$, the *timed trace* of α, is defined to be the pair

$$t\text{-}trace(\alpha) \triangleq (\delta \upharpoonright (vis(A) \times \mathsf{T}), ltime(\alpha))$$

The following lemma shows that the definitions of timed traces for execution fragments and timed execution fragments are properly related:

Lemma 4.14

If α samples Σ then $t\text{-}trace(\alpha) = t\text{-}trace(\Sigma)$.

■

4.2.2.2 Sampling Characterization of Liveness Conditions

As mentioned above we will characterize liveness conditions for safe timed I/O automata by a set of ordinary executions.

Let A be a safe timed I/O automaton and let $L_s \subseteq exec^\infty(A)$ be a set of admissible (ordinary) executions of A. Then L_s is said to be a *sampling characterization* of the set

$$L = \{\Sigma \in t\text{-}exec^\infty(A) \mid \text{for all } \alpha, \text{ if } \alpha \text{ samples } \Sigma, \text{ then } \alpha \in L_s\} \tag{4.6}$$

That is, L contains all those admissible timed executions of A that have all their samplings in L_s. We say that L is *induced* by the sampling characterization L_s. Note, that the sampling characterization L_s may contain "extra" executions that are not samplings of any timed executions in the set L induced by L_s. (Such an extra execution will be the sampling of some timed execution Σ, but since all samplings of Σ are not in L_s, Σ is not in L.) If L_s coincides with the set of all samplings of all timed executions in the set L induced by L_s, i.e., if L_s does not contain any "extra" executions, then L_s is said to be *minimal*.

If the set L induced by L_s is a liveness condition for A, L_s is said to be a *liveness sampling characterization* for A. Furthermore, if (A, L) is a live timed I/O automaton, i.e., if $(A, L \cup t\text{-}exec^{Zt}(A))$ is environment-free, L_s is said to be *environment-free* for A.

A liveness sampling characterization for some safe timed I/O automaton A can now be specified indirectly in exactly the same way we defined liveness conditions in the untimed setting using temporal formulas. Thus, for any temporal formula Q over A we refer to the set

$$L_s = \{\alpha \in exec^\infty(A) \mid \alpha \models Q\} \tag{4.7}$$

as the *sampling characterization induced by Q*. If L_s is a liveness sampling characterization for A, Q is referred to as *timed liveness formula* for A. Furthermore, if L_s is environment-free or minimal, Q is said to be *environment-free* or *minimal*, respectively. Finally, if L is induced by L_s which, in turn, is induced by Q, we say that L is induced by Q.

4.2.2.3 Operations on Live Timed I/O Automata

As in the untimed setting we now show how the liveness of live timed I/O automata obtained as results of the operators (parallel composition, action hiding, and action renaming) is induced by temporal formulas derived from the temporal formulas inducing the liveness of the live timed I/O automata to which the operators were applied.

We start by looking at parallel composition and for that we need the following result, which expresses the relationship between sampling and projection ($\lceil$). We state the result without proof (except we note that point 3 follows from points 1 and 2).

Lemma 4.15

Let $A_1, \ldots, A_N$ be compatible safe timed I/O automata, $A = A_1 \| \cdots \| A_N$, and $\Sigma \in$ t-exec(A). Then, for all $1 \leq i \leq N$,

1. *if α samples Σ, then $\alpha \lceil A_i$ samples $\Sigma \lceil A_i$,*
2. *if α_i sample $\Sigma \lceil A_i$, then there exists an α such that α samples Σ and $\alpha_i = \alpha \lceil A_i$, and*
3. *$\{\alpha \lceil A_i \mid \alpha \text{ samples } \Sigma\} = \{\alpha_i \mid \alpha_i \text{ samples } \Sigma \lceil A_i\}$.*

■

Lemmas 4.2 and 4.3 above for safe I/O automata are actually valid for safe *timed* I/O automata as well. We restate the timed version of Lemma 4.3.

Lemma 4.16

Let $A_1, \ldots, A_N$ be compatible safe timed I/O automata and $Q_1, \ldots, Q_N$ be stuttering-insensitive temporal formulas over $A_1, \ldots, A_N$, respectively. Let $A = A_1 \| \cdots \| A_N$ and $\alpha \in exec(A)$. Then,

$$\alpha \lceil A_1 \models Q_1 \quad \textit{and} \quad \cdots \quad \textit{and} \quad \alpha \lceil A_N \models Q_N \qquad \textit{iff} \qquad \alpha \models Q_1 \wedge \ldots \wedge Q_N$$

■

The main result for parallel composition of live timed I/O automata can now be stated and proved.

Proposition 4.17

Let $(A_1, L_1), \ldots, (A_N, L_N)$ be compatible live timed I/O automata and $Q_1, \ldots, Q_N$ be stuttering-insensitive temporal formulas over $A_1, \ldots, A_N$, respectively, such that each L_i is induced by Q_i. Let $(A, L) = (A_1, L_1) \| \cdots \| (A_N, L_N)$. Then L is induced by $Q_1 \wedge \ldots \wedge Q_N$.

Proof

In Appendix B.

■

Attention is now turned to the simpler operations of action hiding and action renaming.

Proposition 4.18

Let (A, L) be a live timed I/O automaton such that L is induced by the temporal formula Q for A and let $\mathcal{A} \subseteq local(A)$. Then the liveness condition of $(A, L) \setminus \mathcal{A}$ is induced by Q.

Proof

In Appendix B.

■

Proposition 4.19

Let (A, L) be a live timed I/O automaton such that L is induced by the temporal formula Q for A, and let ρ be an action mapping applicable to (A, L). Define $\rho(Q)$ to be the temporal formula obtained by applying ρ to every action function in Q. Then the liveness condition of $\rho((A, L))$ is induced by $\rho(Q)$.

Proof

In Appendix B.

■

4.2.2.4 Fairness

The fairness formulas (Equations (4.2)–(4.5)) presented in the untimed setting also express fairness requirements in the timed setting. However, fairness in the timed setting is not necessarily environment-free as in the untimed setting.

The problem is that environment-freedom can be jeopardized because the system may collaborate with the environment to generate non-Zeno-tolerant outcomes, as explained in Section 2.2.2, regardless of the fairness formulas. We do not investigate further if weak and strong fairness are environment-free for certain classes of safe timed I/O automata.

4.3 Embedding

In Section 2.3 we introduced the *patient* operator, which takes a safe or live I/O automaton as argument and returns the corresponding safe or live *timed* I/O automaton, respectively, that allows time to pass arbitrarily.

The *patient* operator on safe I/O automata (cf. Definition 2.34) adds an extra state component representing real time. When describing state spaces using state variables, we shall assume that the *patient* operator adds an extra state variable called *now* (as well as it adds the extra time-passage action ν). Thus, we shall assume that *now* is not a state variable of any safe I/O automaton to which we apply *patient*.

In Section 2.3 we described what it means to *untime* a timed execution of a patient safe I/O automaton. A similar definition can be given for ordinary executions: let A be a safe I/O automaton such that $now \notin variables(A)$ and $\nu \notin acts(A)$, and let $A_p = patient(A)$. Then for any $\alpha \in exec(A_p)$, define $untime(\alpha)$ to be the execution of A obtained from α by restricting every state to the state variables of A and removing every time-passage step (which do not change the state variables of A). Formally we have

$$untime(\alpha) \;\triangleq\; \natural(\alpha \restriction (variables(A), acts(A)))$$

The following lemma, which we state without proof, says that the definition of $untime(\alpha)$ is sensible.

Lemma 4.20

Let A be a safe I/O automaton such that $now \notin variables(A)$ and $\nu \notin acts(A)$, and let $A_p = patient(A)$. Then, for any $\Sigma \in t\text{-}exec(A_p)$ and $\alpha \in exec(A_p)$, if α samples Σ, then $untime(\alpha) = untime(\Sigma)$.

■

Lemma 4.21

Let A be a safe I/O automaton and let Q be a stuttering-insensitive temporal formula over A. Furthermore, let $A_p = patient(A)$. Then, for all $\alpha \in exec(A_p)$,

$$untime(\alpha) \models Q \qquad \textit{iff} \qquad \alpha \models Q$$

Proof

In Appendix B.

■

We can now state and prove the main result of this section, namely that stuttering-insensitive temporal formulas carry over as environment-free liveness formulas when applying the *patient* operator.

Proposition 4.22

Let (A, L) be a live I/O automaton with L induced by a stuttering-insensitive temporal formula Q over A. Furthermore, let $(A_p, L_p) = patient(A, L)$. Then, L_p is induced by Q, and Q is minimal.

Proof

In Appendix B.

■

The minimality of Q as implied by the proposition will be important when proving that a live timed I/O automaton correctly implements the patient version of a live I/O automaton. In fact, as we shall see in the next chapter, our proof techniques in the timed setting requires liveness conditions of certain live timed I/O automata to be induced by minimal temporal formulas.

This concludes this chapter. We have described how to specify safe (timed) I/O automata using a precondition-effect language and how to use the temporal logic defined in Chapter 3 to specify liveness. Furthermore, this chapter contains several results which state how operations in the semantic model are reflected in the syntax.

Before we start the protocol verification example in Part II of this report, the next chapter deals with presenting a number of proof techniques for proving correctness.

Chapter 5

Proof Techniques

The previous chapters have defined the general models of timed and untimed systems that we will use in this work, and described our approach to specifying objects of these models. This chapter is devoted to presenting a host of *proof techniques* for proving that one live (timed) I/O automaton *correctly* or *safely* implements another live (timed) I/O automaton.

In Chapter 2 the notions of safe and correct implementation are defined. These notions are, for both untimed and timed systems, based on the (timed) traces that the involved systems can exhibit. For safe implementation, all (timed) traces are considered, whereas correct implementation restricts attention to live (timed) traces. The respective implementation notions are then expressed as the subset relation between the sets of all/live (timed) traces of the involved systems.

For untimed systems, reasoning about implementation directly in terms of trace inclusion is not feasible. First of all, traces are defined implicitly as the traces of the executions, and second, the liveness condition is defined implicitly as the set of executions that satisfy a certain temporal formula. Thus, the sets of traces and live traces are not readily available but are derived from safe I/O automata and temporal formulas. This calls for some *proof techniques* that are based on this available information and that are sound with respect to the safe and correct implementation relations.

The same discussion is valid for timed systems as well. In timed systems there is even an extra level of indirection since the liveness condition of a live timed I/O automaton is usually induced by a sampling characterization which, in turn, is induced by a temporal formula.

We first present, in Section 5.1, the proof techniques used for untimed systems, and then, in Section 5.2, these techniques are extended to timed systems. Most of the techniques are taken from [GSSL93] and are included here to make this report self-contained. We refer to [GSSL93] for details and proofs.

5.1 Untimed Systems

This section presents a number of techniques for proving the safe implementation relation and assist in proving the correct implementation relation for live I/O automata. The techniques are based on *simulations* between safe I/O automata, which are sound with respect to the *safe* implementation relation, i.e., trace inclusion.

However, as shown in [GSSL93], it turns out that a stronger result can be proved for the simulation techniques: that there is a certain correspondence between the *executions* of the

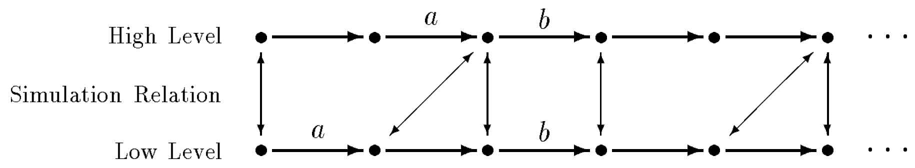


Figure 5.1

Example of a simulation. The actions a and b are external actions. The rest of the transitions are thought of as labeled by internal actions.

involved safe I/O automata and not only between their traces. Since the liveness conditions of live I/O automata are stated in terms of executions and not in terms of traces, this result, which is called the *Execution Correspondence Theorem*, can form the basis for the proof of the *correct* implementation relation, i.e., live trace inclusion.

Thus, when proving correct implementation between two live I/O automata, first a simulation result between the safe I/O automata parts is proved and then this simulation result and the Execution Correspondence Theorem are used to prove live trace inclusion.

We proceed by defining a number of *simulation proof techniques* and stating the Execution Correspondence Theorem. Then we present the proof techniques for proving the safe and correct implementation relations. Finally, we consider the additional proof technique of adding *history variables*.

5.1.1 Simulation Proof Techniques

A *simulation* from A to B, where A and B are safe I/O automata with the same input and output actions, is a relation between the states of A and the states of B such that certain conditions hold. A will be referred to as the *concrete*, *low-level*, or *implementation* safe I/O automaton, and B as the the *abstract*, *high-level*, or *specification* safe I/O automaton.

Exactly what conditions a simulation must satisfy depend on the *kind* of simulation. Below we define notions of, e.g., *forward* and *backward* simulations which differ in few but important respects. Generally, however, two conditions must be satisfied: first, the start states of the two safe I/O automata must be related in a certain way, and, second, each step of the low-level safe I/O automaton must "correspond" to a sequence of steps of the high-level safe I/O automaton.

The second condition is depicted in Figure 5.1. For each step of the low-level safe I/O automaton, i.e., for each low-level step, there must exist a sequence of (high-level) steps of the high-level safe I/O automaton between states related—by the simulation relation—to the pre- and post-state of the low-level step, such that the sequence of high-level steps contains exactly the same external actions as the low-level step. How the sequence of high-level steps is selected depends on what kind of simulation is considered.

Below forward simulations, refinement mappings, and backward simulations are defined. We refer to [GSSL93, LV93a, Jon91] for more details about these simulations.

The simulation techniques use *invariants* of the safe I/O automata to restrict the steps needed to be considered. Define an invariant of a safe I/O automaton A to be any set of states of A that is a superset of the reachable states of A. Equivalently, an invariant can be defined to be a state formula over A that is satisfied by at least all reachable states of A. We will use the two definitions interchangeably.

The following notational convention is used: if R is a relation over $S_1 \times S_2$ and $s_1 \in S_1$, then $R[s_1]$ denotes the set $\{s_2 \in S_2 \mid (s_1, s_2) \in R\}$.

Definition 5.1 (Forward simulation)

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and with invariants I_A and I_B, respectively. A *forward simulation* from A to B, with respect to I_A and I_B, is a relation f over $states(A) \times states(B)$ that satisfies:

1. If $s \in start(A)$ then $f[s] \cap start(B) \neq \emptyset$.
2. If $(s, a, s') \in steps(A)$, $s, s' \in I_A$, and $u \in f[s] \cap I_B$, then there exists an $\alpha \in frag^*(B)$ with $fstate(\alpha) = u$, $lstate(\alpha) \in f[s']$, and $trace(\alpha) = trace(a)$.

We write $A \leq_F B$ if there exists a forward simulation from A to B with respect to some invariants I_A and I_B. If f is a forward simulation from A to B with respect to some invariants I_A and I_B, we write $A \leq_F B$ via f.

■

A refinement mapping is a special case of a forward simulation where the relation is a function. Because of its practical importance (cf. [AL91]) we give an explicit definition.

Definition 5.2 (Refinement mapping)

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and with invariants I_A and I_B, respectively. A *refinement mapping* from A to B, with respect to I_A and I_B, is a function r from $states(A)$ to $states(B)$ that satisfies:

1. If $s \in start(A)$ then $r(s) \in start(B)$.
2. If $(s, a, s') \in steps(A)$, $s, s' \in I_A$, and $r(s) \in I_B$, then there exists an $\alpha \in frag^*(B)$ with $fstate(\alpha) = r(s)$, $lstate(\alpha) = r(s')$, and $trace(\alpha) = trace(a)$.

We write $A \leq_R B$ if there exists a refinement mapping from A to B with respect to some invariants I_A and I_B. If r is a refinement mapping from A to B with respect to some invariants I_A and I_B, we write $A \leq_R B$ via r.

■

In a forward simulation there has to be a sequence of high-level steps starting from *any* of the high-level states related to the low-level pre-state and ending in *some* state related to the low-level post-state. The word "forward" thus refers to the fact that the high-level sequence of steps is constructed from any possible pre-state in a forward direction toward the set of possible post-states.

In a backward simulation, on the other hand, there has to be a sequence of high-level steps *ending* in *any* state related to the low-level post-state and starting in *some* state related to the low-level pre-state. Thus, in a backward simulation the steps are constructed in a backward direction.

This difference between forward and backward simulations implies that they apply to different situations. In some cases a forward simulation is needed whereas other situations might require a backward simulation. We shall see examples of this below.

We need the auxiliary definition of image-finiteness. A relation R over $S_1 \times S_2$ is *image-finite* if for each $s_1 \in S_1$, $R[s_1]$ is a finite set.

Definition 5.3 (Backward simulation)

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and with invariants I_A and I_B, respectively. A *backward simulation* from A to B, with respect to I_A and I_B, is a relation b over $states(A) \times states(B)$ that satisfies:

1. If $s \in I_A$ then $b[s] \cap I_B \neq \emptyset$.
2. If $s \in start(A)$ then $b[s] \cap I_B \subseteq start(B)$.
3. If $(s, a, s') \in steps(A)$, $s, s' \in I_A$, and $u' \in b[s] \cap I_B$, then there exists an $\alpha \in frag^*(B)$ with $lstate(\alpha) = u'$, $fstate(\alpha) \in b[s] \cap I_B$, and $trace(\alpha) = trace(a)$.

We write $A \leq_B B$ if there exists a backward simulation from A to B with respect to some invariants I_A and I_B. If furthermore the backward simulation is image-finite, we write $A \leq_{iB} B$. If b is a backward simulation from A to B with respect to some invariants I_A and I_B, we write $A \leq_B B$ (or $A \leq_{iB} B$ when b is image-finite) via b.

■

In [LV93a] abstract notions of *history variables* [OG76, AL91] and *prophecy variables* [AL91] are given in terms of *history relations* and *prophecy relations*. Below, in Section 5.1.5, we consider history and prophecy variables and show how history variables can be added to a specification.

5.1.2 Execution Correspondence

This subsection introduces the Execution Correspondence Theorem (ECT). The ECT states that if any of the simulations from above has been proven from a low-level safe I/O automaton A to a high-level safe I/O automaton B, then for any execution of A, there exists a "corresponding" execution of B. In order to formalize this notion of correspondence, the notions of *R-relation* and *index mapping* are first introduced.

Definition 5.4 (R-relation and index mappings)

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$. Furthermore, let α and α' be executions of A and B, respectively.

$$\begin{array}{rcl} \alpha & = & s_0 a_1 s_1 a_2 s_2 \cdots \\ \alpha' & = & u_0 b_1 u_1 b_2 u_2 \cdots \end{array}$$

We say that α and α' are *R-related*, written $(\alpha, \alpha') \in R$, if there exists a total, nondecreasing mapping[1] $m : \{0, 1, \ldots, |\alpha|\} \rightarrow \{0, 1, \ldots, |\alpha'|\}$ such that

[1] If, e.g., α is infinite ($|\alpha| = \infty$), then the set $\{0, 1, \ldots, |\alpha|\}$ is supposed to denote the set of natural numbers (not including ∞), and $i \leq |\alpha|$ lets i range over all natural numbers but not ∞.

1. $m(0) = 0$,
2. $(s_i, u_{m(i)}) \in R$ for all $0 \leq i \leq |\alpha|$,
3. $trace(b_{m(i-1)+1} \cdots b_{m(i)}) = trace(a_i)$ for all $0 < i \leq |\alpha|$, and
4. for all j, $0 \leq j \leq |\alpha'|$, there exists an i, $0 \leq i \leq |\alpha|$, such that $m(i) \geq j$.

The mapping m is referred to as an *index mapping* from α to α' with respect to R.

We write $(A, B) \in R$ if for every execution α of A, there exists an execution α' of B such that $(\alpha, \alpha') \in R$.

■

Thus, an index mapping maps indices of states in the low-level execution to indices of states in the high-level execution. Effectively, an index mapping maps low-level states to corresponding high-level states such that the start states correspond (Condition 1), corresponding states are related by R (Condition 2), and the external actions between two consecutive pairs of corresponding states are the same at both the low level and the high level (Condition 3). Condition 4 ensures that the high-level execution (α') is not "too long", i.e., α' must not extend beyond the last state of α' corresponding to a state in α (if such a state exists). (Note, that if α is finite, then α' must also be finite. However, even if α is infinite, α' can be finite if the index mapping is constant for indices above some bound.)

The Execution Correspondence Theorem of [GSSL93] is now stated. The theorem states that if a relation S has been proved to be a forward simulation, refinement mapping, or image-finite backward simulation from A to B, then for any execution of A, there exists an S-related execution of B.

Theorem 5.5 (Execution Correspondence Theorem)

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$. Assume for $X \in \{F, R, iB\}$ that $A \leq_X B$ via S. Then $(A, B) \in S$.

■

5.1.3 Proving Safe Implementation

The simulation proof techniques presented above are sound proof techniques for the safe implementation relation. Before we state this result, we first show two results relating the traces of R-related executions.

Lemma 5.6

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$. Assume that $(\alpha, \alpha') \in R$ and let m be any index mapping from α to α' with respect to R. Then, for all $0 \leq i \leq |\alpha|$, $trace({}_i|\alpha) = trace({}_{m(i)}|\alpha')$.

■

Since for any execution α, ${}_0|\alpha = \alpha$ and any index mapping maps 0 to 0, the following corollary is a direct consequence of Lemma 5.6.

Corollary 5.7

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$. If $(\alpha, \alpha') \in R$, then $trace(\alpha) = trace(\alpha')$.

∎

Using this corollary and ECT, soundness of the simulation techniques can be proved.

Theorem 5.8 (Soundness of simulations w.r.t. safe implementation)

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$. Assume for some $X \in \{F, R, iB\}$ that $A \leq_X B$. Then $A \sqsubseteq_S B$.

∎

5.1.4 Proving Correct Implementation

A proof strategy for proving that a live I/O automaton (A, L) correctly implements another live I/O automaton (B, M) is now described.

Lemma 5.9

Let (A, L) and (B, M) be live I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$. Also, let L and M be induced by the temporal formulas Q_L and Q_M, respectively. Assume for some $X \in \{F, R, iB\}$ that $A \leq_X B$ via S. If, for all $\alpha \in exec(A)$ and $\alpha' \in exec(B)$ with $(\alpha, \alpha') \in S$, $\alpha \models Q_L$ implies $\alpha' \models Q_M$, then $(A, L) \sqsubseteq_L (B, M)$.

Proof

This lemma follows directly from a similar result in [GSSL93] and our definition of a liveness condition being induced by a temporal formula.

∎

Thus, we have the following proof strategy to prove that (A, L) is a correct implementation of (B, M):

1. Prove a simulation S from A to B with respect to some invariants.
2. Assume α and α' are arbitrary executions of A and B, respectively, and assume that $(\alpha, \alpha') \in S$ and α is live (i.e., $\alpha \models Q_L$).
3. Prove that α' is also live (i.e., $\alpha' \models Q_M$).

   This will usually be a proof by contradiction. That is, assume that α' is not live and show that this leads to a contradiction. This strategy gives a nice way of splitting the proof into cases since being live usually means satisfying a conjunction of conditions such that not being live means not satisfying one (at least) of these conditions. Thus, each of the conditions can be considered separately.

It is evident that this proof strategy needs a way to go from temporal formulas satisfied by the high-level execution α' to temporal formulas satisfied by the low-level execution α. For this purpose we have identified the following two basic lemmas which will prove very useful in the verification examples in Part II of this report.

Lemma 5.10

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$. Furthermore, let α and α' be executions of A and B, respectively, such that $(\alpha, \alpha') \in R$. Finally, let C be a set of external actions (from the common set of external actions). Then

$$\alpha \models \Diamond\Box\neg\langle C\rangle \qquad \text{iff} \qquad \alpha' \models \Diamond\Box\neg\langle C\rangle$$

Proof

In Appendix B.

■

Lemma 5.11

Let A and B be safe I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$. Furthermore, let α and α' be executions of A and B, respectively, such that $(\alpha, \alpha') \in R$. Assume P and Q are state formulas over A and B, respectively, such that for all $(s, u) \in R$, if $u \models Q$, then $s \models P$. Then,

$$\text{if} \quad \alpha' \models \Diamond\Box Q \quad \text{then} \quad \alpha \models \Diamond\Box P$$

Proof

In Appendix B.

■

5.1.5 History and Prophecy Variables

In [AL91] *history* and *prophecy* variables (together called *auxiliary* variables) are considered. It is shown that even though it is not possible to find a refinement mapping from A to B, by adding appropriate auxiliary variables to A to obtain A_{aux} it is in most cases possible to find a refinement mapping from A_{aux} to B. Then, since A can be shown to be equivalent to (i.e., to have the same traces as) B, the soundness of refinement mappings implies that A safely implements B.

History variables are only allowed to record the past history of the system. Thus, history variables are allowed in each step to be assigned a value based on all variables in the system, but must not affect the enabledness of actions or the changes made to other (ordinary) variables. As we shall see below, it is easy to syntacticly define how to add a history variable to a system.

Prophecy variables, on the other hand, are much more complicated since they are allowed to constrain the future behavior of the system. It is not possible to give a general syntactic characterization of prophecy variables.

In [GSSL93] and [LV93a] abstract notions of history and prophecy variables are given in terms of *history relations* and *prophecy relations*. A system A_h is then said to be obtained from A by adding history variables if there exists a history relation from A to A_h, and similarly for prophecy variables.

The motivation for adding, e.g., history variables to a specification A to obtain A_h is to ensure that a refinement mapping from A_h to some high-level specification B can be devised. But since the existence of a history relation from A to A_h implies that there exists a forward simulation from A to A_h, it is clear that it is possible to define a forward simulation directly from A to B and thereby avoid mentioning A_h at all. (The forward simulation from A to B would be the composition of the forward simulation from A to A_h and the refinement mapping from A_h to B.)

Similarly, instead of adding prophecy variables to A to get A_p such that a refinement mapping from A_p to B can be devised, it is possible to define a backward simulation directly from A to B.

Now, since history variables can be defined using simple *syntactic* constraints, they are almost free to use, as opposed to prophecy variables. Thus, the approach we take is to use history variables whenever possible (which allows us to use refinement mappings instead of the more complicated notion of forward simulations) but to use backward simulations instead of having to deal with prophecy variables. Whether to use prophecy variables or backward simulations is a matter of taste and probably amounts to the same effort. When using backward simulations the complexity lies in showing that the relation is in fact a backward simulation, and when using prophecy variables the complexity lies in showing that the variables are in fact prophecy variables (which is done in a proof that actually has the flavor of a backward simulation).

Syntactically Adding History Variables

Let there be given a syntactic description of a safe I/O automaton A. Then a history variable h ($\notin$ *variables*(A)) can be added to A to get A_h as follows:

1. To the list of state variables of A, append a line with h, the type of h, and the initial value of h.

2. To each step rule of the form

   name
       Precondition:
           P
       Effect:
           E

   an assignment to h may be added

   name
       Precondition:
           P
       Effect:
           E
           $h := e$

   where e is an expression that may mentions h as well as other variables. Note, that the assignment to h may appear in an if-then-else statement, and that it may be moved anywhere in the effect clause since this does not affect the assignment of values to any of the other variables (but of course could affect the value assigned to h).

   For step rules for input actions, which have no precondition, the assignment to the history variable can be added to the effect clause similarly.

We say that A_h is obtained from safe I/O automaton A by adding the history variable h if the syntactic specification of A_h can be obtained from that of A by 1) and 2). In this case, clearly A_h is a safe I/O automaton and $variables(A_h) = variables(A) \cup \{h\}$. The following simple lemma states the close correspondence between the steps of A and A_h.

Lemma 5.12

Let A_h be obtained from A by adding history variable h. Then,

1. *for each $(s, a, s') \in steps(A)$ and each $s_h \in states(A_h)$ with $s_h \restriction variables(A) = s$, there exists a step $(s_h, a, s'_h) \in steps(A_h)$ such that $s'_h \restriction variables(A) = s'$, and*
2. *for each $(s_h, a, s'_h) \in steps(A_h)$, $(s_h \restriction variables(A), a, s'_h \restriction variables(A)) \in steps(A)$.*

■

Lemma 5.13

Let A_h be obtained from A by adding history variable h. Then,

1. *for each execution $\alpha \in exec(A)$, there exists an execution $\alpha_h \in exec(A_h)$ such that $\alpha_h \restriction A = \alpha$, and*
2. *for each execution $\alpha_h \in exec(A_h)$, $\alpha_h \restriction A \in exec(A)$.*

Proof

In Appendix B.

■

Instead of proving the existence of a history relation from A to A_h we directly prove that A safely implements A_h and vice versa.

Lemma 5.14

Let A_h be obtained from A by adding history variable h. Then $A \sqsubseteq_{\mathrm{S}} A_h$ and $A_h \sqsubseteq_{\mathrm{S}} A$.

Proof

In Appendix B.

■

We now turn attention to live I/O automata. Let (A, L) be a live I/O automaton and let A_h be a safe I/O automaton obtained from A by adding history variable h. Define

$$L_h \triangleq \{\alpha_h \in exec(A_h) \mid \alpha_h \restriction A \in L\}$$

Then (A_h, L_h) is a live I/O automaton since any environment-free strategy (g, f) for (A, L) can be trivially extended to an environment-free strategy (g_h, f_h) for (A_h, L_h) by letting g_h and f_h be like g and f except that they make arbitrary (possible) assignments to the history variable. We say that (A_h, L_h) is a live I/O automaton obtained from (A, L) by adding history variable h.

Lemma 5.15

Let (A_h, L_h) be obtained from (A, L) by adding history variable h. Then $(A, L) \sqsubseteq_{\mathrm{L}} (A_h, L_h)$ and $(A_h, L_h) \sqsubseteq_{\mathrm{L}} (A, L)$.

Proof

In Appendix B.

■

The final lemma of this section deals with liveness formulas.

Lemma 5.16

Let (A_h, L_h) be obtained from (A, L) by adding history variable h, and assume that L is induced by Q. Then L_h is induced by Q.

Proof

In Appendix B.

■

We can now turn attention to similar techniques to be used in the timed setting.

5.2 Timed Systems

The structure of this section is similar to the structure of Section 5.1.

5.2.1 Timed Simulation Proof Techniques

There are only two minor differences between the simulation relations presented here and the simulation relations from the untimed case. First, states related by a simulation relation must have the same time. Second, since the *trace* operator on execution fragments does not adequately abstract from time-passage actions, the simulation techniques below use a notion of *visible trace*. For any timed automaton A and any execution fragment α of A, define the visible trace of α, written $\mathit{vis\text{-}trace}_A(\alpha)$, or just $\mathit{vis\text{-}trace}(\alpha)$ when A is clear from context, to be $\alpha \upharpoonright \mathit{vis}(A)$. Similarly, given any sequence of actions β, define the visible trace of β, written $\mathit{vis\text{-}trace}_A(\beta)$, or just $\mathit{vis\text{-}trace}(\beta)$ if A is clear from context, to be $\beta \upharpoonright \mathit{vis}(A)$.

We now introduce the notions of timed forward simulations, timed refinement mappings, and timed backward simulations.

Definition 5.17 (Timed forward simulation)

Let A and B be safe timed I/O automata with $\mathit{in}(A) = \mathit{in}(B)$ and $\mathit{out}(A) = \mathit{out}(B)$ and with invariants I_A and I_B, respectively. A *timed forward simulation* from A to B, with respect to I_A and I_B, is a relation f over $\mathit{states}(A) \times \mathit{states}(B)$ that satisfies:

1. If $u \in f[s]$ then $u.\mathit{now} = s.\mathit{now}$.

2. If $s \in \mathit{start}(A)$ then $f[s] \cap \mathit{start}(B) \neq \emptyset$.

3. If $(s, a, s') \in \mathit{steps}(A)$, $s, s' \in I_A$, and $u \in f[s] \cap I_B$, then there exists an $\alpha \in \mathit{frag}^*(B)$ with $\mathit{fstate}(\alpha) = u$, $\mathit{lstate}(\alpha) \in f[s']$, and $\mathit{vis\text{-}trace}(\alpha) = \mathit{vis\text{-}trace}(a)$.

Write $A \leq_{tF} B$ if there exists a timed forward simulation from A to B with respect to some invariants I_A and I_B. If f is a timed forward simulation from A to B with respect to some invariants I_A and I_B, we write $A \leq_{tF} B$ via f.

■

Definition 5.18 (Timed refinement mapping)

Let A and B be safe timed I/O automata with $\mathit{in}(A) = \mathit{in}(B)$ and $\mathit{out}(A) = \mathit{out}(B)$ and with invariants I_A and I_B, respectively. A *timed refinement mapping* from A to B, with respect to I_A and I_B, is a function r from $\mathit{states}(A)$ to $\mathit{states}(B)$ that satisfies:

1. $r(s).\mathit{now} = s.\mathit{now}$.
2. If $s \in \mathit{start}(A)$ then $r(s) \in \mathit{start}(B)$.
3. If $(s, a, s') \in \mathit{steps}(A)$, $s, s' \in I_A$, and $r(s) \in I_B$, then there exists an $\alpha \in \mathit{frag}^*(B)$ with $\mathit{fstate}(\alpha) = r(s)$, $\mathit{lstate}(\alpha) = r(s')$, and $\mathit{vis\text{-}trace}(\alpha) = \mathit{vis\text{-}trace}(a)$.

Write $A \leq_{tR} B$ if there exists a timed refinement mapping from A to B with respect to some invariants I_A and I_B. If r is a timed refinement mapping from A to B with respect to some invariants I_A and I_B, we write $A \leq_{tR} B$ via r.

■

Definition 5.19 (Timed backward simulation)

Let A and B be safe timed I/O automata with $\mathit{in}(A) = \mathit{in}(B)$ and $\mathit{out}(A) = \mathit{out}(B)$ and with invariants I_A and I_B, respectively. A *timed backward simulation* from A to B, with respect to I_A and I_B, is a relation b over $\mathit{states}(A) \times \mathit{states}(B)$ that satisfies:

1. If $u \in b[s]$ then $u.\mathit{now} = s.\mathit{now}$.
2. If $s \in I_A$ then $b[s] \cap I_B \neq \emptyset$.
3. If $s \in \mathit{start}(A)$ then $b[s] \cap I_B \subseteq \mathit{start}(B)$.
4. If $(s, a, s') \in \mathit{steps}(A)$, $s, s' \in I_A$, and $u' \in b[s'] \cap I_B$, then there exists an $\alpha \in \mathit{frag}^*(B)$ with $\mathit{lstate}(\alpha) = u'$, $\mathit{fstate}(\alpha) \in b[s] \cap I_B$, and $\mathit{vis\text{-}trace}(\alpha) = \mathit{vis\text{-}trace}(a)$.

Write $A \leq_{tB} B$ if there exists a timed backward simulation from A to B with respect to some invariants I_A and I_B. If furthermore the timed backward simulation is image-finite, write $A \leq_{itB} B$. If b is a timed backward simulation from A to B with respect to some invariants I_A and I_B, we write $A \leq_{tB} B$ (or $A \leq_{itB} B$ when b is image-finite) via b.

■

5.2.2 Execution Correspondence

As in the untimed case, the simulation relations imply a certain correspondence between the ordinary executions of the involved timed automata. The following definition formalizes this correspondence, called *timed R-relation*, and defines a notion of *timed index mapping*. The definition is similar to Definition 5.4 in the untimed model; the only differences are that the R relation must relate states with the same time and that the definition below deals with visible traces as opposed to traces, i.e., the same differences as in the simulations.

Definition 5.20 (Timed R-relation and timed index mappings)

Let A and B be safe timed I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$, and let R be a relation over $states(A) \times states(B)$ such that if $(s, u) \in R$, then $s.now = u.now$. Furthermore, let α and α' be (ordinary) executions of A and B, respectively.

$$\begin{array}{rcl} \alpha & = & s_0 a_1 s_1 a_2 s_2 \cdots \\ \alpha' & = & u_0 b_1 u_1 b_2 u_2 \cdots \end{array}$$

Let α and α' be *timed R-related*, written $(\alpha, \alpha') \in_t R$, if there exists a total, nondecreasing mapping $m : \{0, 1, \ldots, |\alpha|\} \to \{0, 1, \ldots, |\alpha'|\}$ such that

1. $m(0) = 0$,
2. $(s_i, u_{m(i)}) \in R$ for all $0 \leq i \leq |\alpha|$,
3. $\textit{vis-trace}(b_{m(i-1)+1} \cdots b_{m(i)}) = \textit{vis-trace}(a_i)$ for all $0 < i \leq |\alpha|$, and
4. for all j, $0 \leq j \leq |\alpha'|$, there exists an i, $0 \leq i \leq |\alpha|$, such that $m(i) \geq j$.

The mapping m is referred to as a *timed index mapping* from α to α' with respect to R.

Write $(A, B) \in_t R$ if for every execution α of A, there exists an execution α' of B such that $(\alpha, \alpha') \in_t R$.

■

Now the Execution Correspondence Theorem for the timed case [GSSL93] can be stated.

Theorem 5.21 (Execution Correspondence Theorem)

Let A and B be safe timed I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$. Assume for $X \in \{tF, tR, itB\}$ that $A \leq_X B$ via S. Then $(A, B) \in_t S$.

■

5.2.3 Proving Safe Timed Implementation

Due to the fact that timed R-related executions have the same time in related states and have a correspondence between the their visible traces, it is possible to prove that timed R-related executions have the same timed traces.

Lemma 5.22

Let A and B be safe timed I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$ such that if $(s,u) \in R$ then $s.now = u.now$. Then, if $(\alpha, \alpha') \in_t R$, then $t\text{-}trace(\alpha) = t\text{-}trace(\alpha')$.

■

The soundness of the timed simulations with respect to the timed safe preorders can now be stated.

Theorem 5.23 (Soundness of timed simulations w.r.t. safe timed implementation)

Let A and B be safe timed I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$. Assume for some $X \in \{tF, tR, itB\}$ that $A \leq_X B$. Then $A \sqsubseteq_{\mathrm{St}} B$.

■

5.2.4 Proving Correct Timed Implementation

We can prove the following result which is similar to Lemma 5.9 in the untimed setting. This lemma will form the basis of any proof of correct implementation in the timed setting.

Lemma 5.24

Let (A, L) and (B, M) be live timed I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$. Also, let L and M be induced by Q_L and Q_M, respectively, and assume that Q_M is minimal. Assume for some $X \in \{tF, tR, itB\}$ that $A \leq_X B$ via S. If, for all $\alpha \in exec^{\infty}(A)$ and $\alpha' \in exec^{\infty}(B)$ with $(\alpha, \alpha') \in S$, $\alpha \models Q_L$ implies $\alpha' \models Q_M$, then $(A, L) \sqsubseteq_{\mathrm{Lt}} (B, M)$.

■

Proof

This lemma directly follows from a similar result in [GSSL93] and our definition of a sampling characterization being induced by a temporal formula.

■

Lemma 5.24 can be used to prove the correct timed implementation relation between two live timed I/O automata in a manner similar to the way Lemma 5.9 is used in the untimed model. However, one must first prove that the high-level liveness condition is induced by a *minimal* timed liveness formula.

The following lemmas correspond to Lemmas 5.10 and 5.11 above.

Lemma 5.25

Let A and B be safe timed I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$ such that if $(s,u) \in R$, then $s.now = u.now$. Furthermore, let α and α' be executions of A and B, respectively, such that $(\alpha, \alpha') \in R$. Finally, let C be a set of visible actions (from the common set of visible actions). Then

$$\alpha \models \Diamond\Box\neg\langle C\rangle \qquad \text{iff} \qquad \alpha' \models \Diamond\Box\neg\langle C\rangle$$

Proof

Similar to the proof of Lemma 5.10.

■

Lemma 5.26

Let A and B be safe timed I/O automata with $in(A) = in(B)$ and $out(A) = out(B)$ and let R be a relation over $states(A) \times states(B)$ such that if $(s, u) \in R$, then $s.now = u.now$. Furthermore, let α and α' be executions of A and B, respectively, such that $(\alpha, \alpha') \in R$. Assume P and Q are state formulas over A and B, respectively, such that for all $(s, u) \in R$, if $u \models Q$, then $s \models P$. Then,

$$\text{if} \quad \alpha' \models \Diamond\Box Q \quad \text{then} \quad \alpha \models \Diamond\Box P$$

Proof

Similar to the proof of Lemma 5.11.

■

5.2.5 History and Prophecy Variables

As in the untimed setting it is possible to add history variables to safe and live timed I/O automata. As above we only deal with history variables and adhere to timed backwards simulations instead of using prophecy variables.

Syntactically Adding History Variables

The syntactic rules for adding history variables to a safe timed I/O autoamaton are equal to the same rules in the untimed setting. However, in the timed setting, we do not allow history variables to be updated in time-passage steps since otherwise the resulting object would not necessarily be a safe timed I/O automaton (that is, the trajectory axiom **S5** of Definition 2.17 could be violated). Thus, a history variable h ($\notin$ *variables*(A)) can be added to a safe timed I/O automaton A to get A_h by following the two rules in Section 5.1.5 with the restriction that h must not be changed in the step rule for the time-passage action ν. We say that A_h is obtained from A by adding the history variable h. Clearly A_h is a safe timed I/O automaton and *variables*(A_h) = *variables*$(A) \cup \{h\}$.

In previous chapters we have defined how to restrict ordinary executions to subsets of state variables and actions. Below we need a similar result for timed executions, however, we need only deal with restriction to a subset of the state variables. So, let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$ be a timed execution of a safe timed I/O automaton A. Then, for any set $\mathcal{V} \subseteq$ *variables*(A), define $\Sigma \upharpoonright \mathcal{V}$ to be the sequence $\omega'_0 a_1 \omega'_1 a_2 \omega'_2 \cdots$, where for each index i and each $t \in dom(\omega_i)$, $\omega'_i(t) = \omega_i(t) \upharpoonright \mathcal{V}$. Thus, informally $\Sigma \upharpoonright \mathcal{V}$ is obtained from Σ by restricting all states in the range of all trajectories to $\mathcal{V}$. If A_h is obtained from A by adding history variable h and $\Sigma_h \in$ *t-exec*(A_h), we let $\Sigma_h \upharpoonright A$ be a shorthand for $\Sigma_h \upharpoonright$ *variables*(A).

As in the untimed case, we have the following lemmas.

Lemma 5.27

Let A_h be obtained from A by adding history variable h. Then,

1. *for each* $(s, a, s') \in \mathit{steps}(A)$ *and each* $s_h \in \mathit{states}(A_h)$ *with* $s_h \restriction \mathit{variables}(A) = s$*, there exists a step* $(s_h, a, s'_h) \in \mathit{steps}(A_h)$ *such that* $s'_h \restriction \mathit{variables}(A) = s'$*, and*

2. *for each* $(s_h, a, s'_h) \in \mathit{steps}(A_h)$*,* $(s_h \restriction \mathit{variables}(A), a, s'_h \restriction \mathit{variables}(A)) \in \mathit{steps}(A)$*.*

■

Lemma 5.28

Let A_h *be obtained from* A *by adding history variable* h*. Then,*

1. *for each timed execution* $\Sigma \in \textit{t-exec}(A)$*, there exists a timed execution* $\Sigma_h \in \textit{t-exec}(A_h)$ *such that* $\Sigma_h \restriction A = \Sigma$*, and*

2. *for each timed execution* $\Sigma_h \in \textit{t-exec}(A_h)$*,* $\Sigma_h \restriction A \in \textit{t-exec}(A)$*.*

Proof

In Appendix B.

■

These lemmas allow us to prove that a safe timed I/O automaton A is a safe implementation of any safe timed I/O automaton A_h obtained by adding history variable h to A, and vice versa.

Lemma 5.29

Let A_h *be obtained from* A *by adding history variable* h*. Then* $A \sqsubseteq_{\mathrm{St}} A_h$ *and* $A_h \sqsubseteq_{\mathrm{St}} A$*.*

Proof

Similar to the proof of Lemma 5.14 by using Lemma 5.28.

■

Now, let (A, L) be a live timed I/O automaton and let A_h be a safe timed I/O automaton obtained from A by adding history variable h. Define

$$L_h \triangleq \{\Sigma_h \in \textit{t-exec}^{\infty}(A_h) \mid \Sigma_h \restriction A \in L\}$$

Then (A_h, L_h) is a live timed I/O automaton since any environment-free strategy (g, f) for $(A, L \cup \textit{t-exec}^{Zt}(A))$ can be trivially extended to an environment-free strategy (g_h, f_h) for $(A_h, L_h \cup \textit{t-exec}^{Zt}(A_h))$ by letting g_h and f_h be like g and f except that they make arbitrary (possible) assignments to the history variable. We say that (A_h, L_h) is a live timed I/O automaton obtained from (A, L) by adding history variable h.

Lemma 5.30

Let (A_h, L_h) *be obtained from* (A, L) *by adding history variable* h*. Then* $(A, L) \sqsubseteq_{\mathrm{Lt}} (A_h, L_h)$ *and* $(A_h, L_h) \sqsubseteq_{\mathrm{Lt}} (A, L)$*.*

Proof

Similar to the proof of Lemma 5.15 by using Lemma 5.28.

■

Before we can prove the final lemma, which deals with timed liveness formulas, we state the following trivial result without proof.

Lemma 5.31

Let A_h be obtained from A by adding history variable h. Furthermore let α_h and Σ_h range over $exec(A_h)$ and t-$exec(A_h)$, respectively, and let α and Σ range over $exec(A)$ and t-$exec(A)$, respectively. Then,

1. *if α_h samples Σ_h then $\alpha_h \upharpoonright A$ samples $\Sigma_h \upharpoonright A$, and*
2. *if α samples $\Sigma_h \upharpoonright A$, then there exists an α_h such that $\alpha = \alpha_h \upharpoonright A$ and α_h samples Σ_h.*

■

Lemma 5.32

Let (A_h, L_h) be obtained from (A, L) by adding history variable h, and assume that L is induced by Q. Then L_h is induced by Q.

Proof

In Appendix B.

■

This concludes the theoretical part of the report. We now turn attention to the verification example of proving correctness of two solutions to the at-most-once message delivery problem.

Part II

Reliable At-Most-Once Message Delivery Protocols

A Protocol Verification Example

Chapter 6

Specification S

This chapter describes the top-level specification of the "at-most-once message delivery" problem. The specification will be given in terms of a live I/O automaton. The objective of the S level is to give a clear, easy-to-understand specification that can easily be checked to have the desirable behavior.

The *at-most-once message delivery* problem is that of delivering a sequence of messages submitted by a user at one location to another user at another location. Ideally, we would like to insist that all messages be delivered in the order in which they are sent, each exactly once, and that an acknowledgement be returned for each delivered message.[1]

Unfortunately, it is expensive to achieve these goals in the presence of failures (e.g., node crashes). In fact, it is impossible to achieve them at all unless some change is made to the stable state (i.e., the state that survives a crash) for each message. To permit less expensive solutions, we weaken the statement of the problem slightly. We allow some messages to be lost when a node crash occurs; however, no messages should otherwise be lost, and those messages that are delivered should not be reordered or duplicated. (The specification is weakened in this way because message loss is generally considered to be less damaging than duplicate delivery.) Now it is required that the user who sent the message receive either an acknowledgement that the message has been delivered, or in the case of crashes, an indication that the message might have been lost.

Even though our specification S is *centralized* (i.e., has no distributed structure), the external actions of S can be partitioned into actions connected to the user at the sender side and actions connected to the user at the receiver side. This *user interface*, which will be the same for all subsequent implementations, is depicted in Figure 6.1, where the specification S is shown as a "black box".

A user can send a message m to the system by issuing a $send\_msg(m)$ action, and the system can pass a message m to the user at the receiver end by means of a $receive\_msg(m)$ action. Crashes at the sender and receiver sides are modeled as inputs $crash_s$ and $crash_r$, respectively[2], and the corresponding recovery actions are outputs $recover_s$ and $recover_r$. If a $crash_s$ but not yet a $recover_s$ action has occurred, we say the the sender side is *crashed* or equivalently that it is in *recovery phase*. Correspondingly for the receiver side. During a crash messages can be lost. This is in S modeled by a $lose(I)$ actions (not depicted in Figure 6.1 since it is internal).

[1] Our definition of at-most-once message delivery is different from what some people call at-most-once message delivery in that we include acknowledgements and require messages to be delivered in order.

[2] We will use subscripts s and r on actions and state variables to indicate which are related to the sender and receiver sides, respectively.

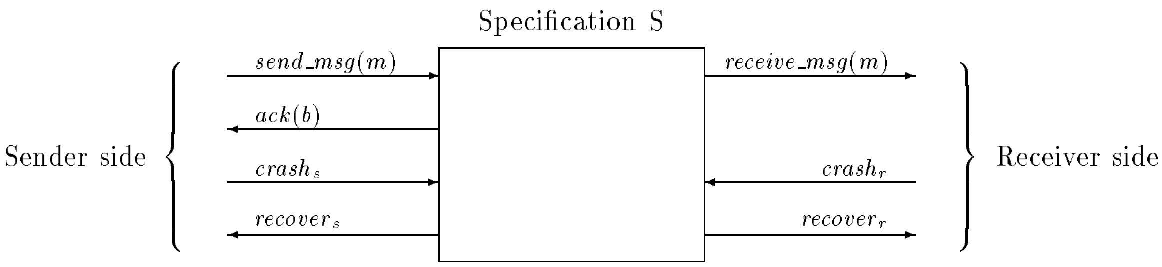


Figure 6.1

The specification S as a "black box"

Finally, there is a simple acknowledgement mechanism incorporated into the specification. An action $ack(b)$, where b is a Boolean, notifies the user at the sender side about the status of the *last* message sent. If acknowledgements are needed for each message, the user must wait for acknowledgement before sending the next message. Our simpler acknowledgement mechanism reflects the way typical low-level protocols work. Thus, if the user sends a sequence of messages $m_1, \ldots, m_n$ without waiting for acknowledgement between each pair of messages, a subsequent acknowledgement will be for message m_n. Ideally, an $ack(\mathit{true})$ should be issued if the last message sent has been successfully delivered to the receiver, and an $ack(\mathit{false})$ should be issued if the last message has been lost during a crash. This is, again, impossible to obtain in a distributed implementation unless some changes are made to the stable state for each message, so we will use a weaker acknowledgement mechanism: if an $ack(\mathit{true})$ is issued, the last message has been successfully receiver. If, on the other hand, an $ack(\mathit{false})$ is issued, the only thing the user can infer is that a crash has occurred. Thus, even in the case of negative acknowledgement, the last message might have been successfully delivered since all messages are not necessarily lost during crashes.

6.1 The Specification of S

We now define the live I/O automaton representing the specification S. We will let S represent both the name of this level of development and the name of the live I/O automaton.

We specify S by defining its components (cf. Definitions 2.1 and 2.8). We refer to the safe I/O automaton part of S by A_{S}, and to the liveness part by L_{S}. Thus, $\mathrm{S} = (A_{\mathrm{S}}, L_{\mathrm{S}})$. L_{S} will be specified implicitly by an environment-free liveness formula Q_{S} for A_{S}.

6.1.1 States and Start States

In S and the lower level protocols we assume that messages are taken from a set *Msg*. We require that $\texttt{nil} \notin \mathit{Msg}$ but assume no other properties of *Msg*.

The state space of S is made up of four state variables as shown in the following table, which furthermore shows the types and initial values of the state variables. The *status* variable ranges over the set

$$Stat \triangleq \mathsf{Bool} \cup \{?\}$$

Variable	Type	Initially	Description
queue	Msg^*	ε	The list of messages sent but not yet delivered.
rec_s	Bool	*false*	*true* iff the sender side has crashed and not yet recovered.
rec_r	Bool	*false*	*true* iff the receiver side has crashed and not yet recovered.
status	*Stat*	*false*	Indicates the status of the last message sent. The special value '?' indicates that the last message sent is still in *queue* and no crashes have occurred since it was sent.

6.1.2 Actions

The set of actions of S consists of the input and output actions from Figure 6.1 plus the internal $lose(I)$ action.

Input:
- $send\_msg(m)$, $m \in Msg$
- $crash_s$
- $crash_r$

Output:
- $receive\_msg(m)$, $m \in Msg$
- $ack(b)$, $b \in \mathsf{Bool}$
- $recover_s$
- $recover_r$

Internal:
- $lose(I)$, $I \subseteq \mathsf{N}$

6.1.3 Steps

The transition relation $steps(A_S)$ will be specified using the precondition-effect style presented in Section 4.1.1.

$send\_msg(m)$
  Effect:
    $queue := queue \frown m$
    $status := ?$

$ack(b)$
  Precondition:
    $status = b$
  Effect:
    none

$receive\_msg(m)$
  Precondition:
    $queue \neq \varepsilon \wedge$
    $head(queue) = m$
  Effect:
    $queue := tail(queue)$
    if $queue = \varepsilon \wedge status = ?$ then
      $status := true$

$crash_s$
  Effect:
    $rec_s := true$

$crash_r$
  Effect:
    $rec_r := true$

$lose(I)$
  Precondition:
    $(rec_s = true \lor rec_r = true) \land I \subseteq dom(queue)$
  Effect:
    if $queue \neq \varepsilon \land maxidx(queue) \in I$
      $status := false$
    else
      optionally $status := false$
    $queue := delete(queue, I)$

$recover_s$
  Precondition:
    $rec_s = true$
  Effect:
    $rec_s := false$

$recover_r$
  Precondition:
    $rec_r = true$
  Effect:
    $rec_r := false$

The function *delete* in the step rule for $lose(I)$ deletes messages with indices in I from *queue*. Formally, for any list q and any set $I \subseteq dom(q)$, define

$$delete(q, I) \triangleq \langle q[i] \mid i \in dom(q) \land i \notin I \rangle$$

The notation to the right of $\triangleq$ is defined in Appendix A.

The handling of *queue*, rec_s, and rec_r in the step rules is self-explanatory. The handling of *status* is a bit more complicated: when a new message m is sent to the system (modeled by $send\_msg(m)$ steps), *status* is changed to ? to indicate that the last message sent is in *queue*. When a message is delivered to the receiver (modeled by $receive\_msg(m)$ steps) and *queue* thereby becomes empty, *status* should be changed to *true*, but only if the message delivered is in fact the last message sent and not another message, which happens to be last on *queue* because the last message sent has been lost in a crash. Thus, at any point a status value of ? indicates that the message at the end of *queue* is actually the last message sent by the sender. This explains the $receive\_msg(m)$ steps. The $lose(I)$ action then records if the message at the end of *queue* is lost by changing *status* to *false*. (If the message at the end of *queue* is not the last message sent, *status* would already be *false*). On the other hand, if the message at the end of *queue* is not deleted, we are still allowed to change *status* to *false* according to our informal description of the acknowledge mechanism given in the introduction to this chapter.

Note, that it is possible for the system to output a positive acknowledgement for a message and then "change its mind" and start issuing negative acknowledgements. However, this change of mind can only happen during a crash. (In such a situation the user knows that the last message has been delivered since she has received a positive acknowledgement.)

Another thing to note is the fact that the $ack(b)$ steps do not disable themselves. Thus, once *status* becomes *true* or *false*, acknowledgements can be sent continuously until a new message is put into *queue* by a $send\_msg(m)$ step. (Actually, with the liveness restrictions we present below, acknowledgements *must* be issued infinitely often if *status* stays *true* or *false*, and no crashes occur.) A remedy to this situation would be to introduce an additional flag, which is set when *status* is changed from ? to a Boolean, and reset when an acknowledgement is issued. Acknowledgements should then only be enabled when this flag is set. We have chosen not to introduce the flag since it would only add few interesting aspects to the implementations.

6.1.4 Liveness

We now present the environment-free liveness formula $Q_{\rm S}$ for $A_{\rm S}$, which induces the liveness condition $L_{\rm S}$. The liveness we specify for S is weak fairness to four sets of locally-controlled actions. Two of these sets have associated forcing conditions. Note, that $lose(I)$ actions are not in any set since we do not want to force the system to lose anything. Informally, the sets and forcing conditions are.

1. $ack(b)$ actions

   Forcing condition: $rec_s = rec_r = false$

2. $receive\_msg(m)$ actions

   Forcing condition: $rec_s = rec_r = false$

3. $recover_s$

4. $recover_r$

With these liveness restrictions we guarantee that in the absence of crashes, messages in *queue* will be delivered and acknowledgements for the last message will be issued unless new messages are sent to the system. Furthermore, both the sender side and the receiver side are guaranteed to recover after a crash. (This requirement on recovery could be removed from all levels of abstraction without affecting other liveness properties. All interesting liveness properties are, in fact, conditioned by the assumption that no new crashes occur.)

The liveness requirements can be formalized in the following way. Let

$$\begin{array}{lcl} C_{\rm S,1} & \triangleq & \{ack(true), ack(false)\} \\ C_{\rm S,2} & \triangleq & \{receive\_msg(m) \mid m \in Msg\} \\ C_{\rm S,3} & \triangleq & \{recover_s\} \\ C_{\rm S,4} & \triangleq & \{recover_r\} \end{array}$$

Then the formalization of $Q_{\rm S}$ is

$$\begin{array}{lcl} Q_{\rm S} & \triangleq & WF(C_{\rm S,1}, rec_s = false \wedge rec_r = false) \wedge \\ & & WF(C_{\rm S,2}, rec_s = false \wedge rec_r = false) \wedge \\ & & WF(C_{\rm S,3}) \wedge \\ & & WF(C_{\rm S,4}) \end{array}$$

By Lemma 4.7, $Q_{\rm S}$ is an environment-free liveness formula for $A_{\rm S}$. Thus, $\mathrm{S} = (A_{\rm S}, L_{\rm S})$ is a live I/O automaton. Furthermore, by Lemma 4.8, $Q_{\rm S}$ is stuttering-insensitive.

This concludes the formal specification of the at-most-once message delivery problem.

Chapter 7

Delayed-Decision Specification D

In our specification S, presented in Chapter 6, we saw that it is allowed to lose any number of messages in the system, but only if either rec_s or rec_r is *true*, i.e., we can only lose messages between crash and recovery. In the low-level protocols we consider, the choice whether or not to lose a message because of a crash may be postponed until *after* recovery and the choice is dependent on certain *race-conditions* on the network channels: a message m traveling on a channel and the receiver have no way of knowing if the sender has crashed, so even if the sender has crashed, the message might still be successfully received by the receiver. But, if the sender recovers and sends a new message on the channel, the reception of this new message before m (our channels are not FIFO) will lead to the discartion of m when it is eventually received (since otherwise messages could be reordered).

This postponing of nondeterministic choices suggests that we at one point have to rely on a *backward* simulation to prove correctness of the low-level protocols. In a first attempt, a timed backward simulation was proved directly from the Clock-Based Protocol C to S (or rather the *patient* version of S). A lot of this work would have had to be repeated in a backward simulation from the Five-Packet Handshake Protocol H to S, so after having designed the Generic Protocol G, we proved a backward simulation from G to S, and could then do with a timed refinement from C to *patient*(G) and a refinement from H to G.

Still, the proof from G to S was very large and comprehensive. It is our experience that backward simulations are generally difficult to deal with, mainly because they are not so intuitive as forward simulations. This observation led us to try to "limit" the backward simulation to a development step as small as possible. Generally, one should always try to find steps of development that are intuitive, and remember that a series of steps (with proofs) are generally easier to comprehend than is one big proof, even though the combined length of the small proofs might exceed the length of the big proof.

So, as an intermediate level between S and G we came up with the Delayed-Decision Specification D, which looks very much like S, but instead of deleting messages between crash and recovery, D *marks* arbitrary messages, and marked messages can then be lost at any point. D also deals with postponing of losing (i.e., changing to *false*) the status as the result of a crash. When we describe the steps of D, we will further explain the differences between S and D.

It should be noted, that even though we postpone the decision about which messages to lose, only messages which were in the system between crash and recovery can be lost. A system that did not satisfy this restriction could not, of course, implement S.

The rest of this chapter is organized as follows. First, in Section 7.1, we present D and then, in

Section 7.2, we prove that D correctly implements S.

7.1 The Specification of D

We specify $\mathrm{D} = (A_{\mathrm{D}}, L_{\mathrm{D}})$ as a live I/O-automaton using the notation introduced in Chapter 4. L_{D} will be specified implicitly by the environment-free liveness formula Q_{D} for A_{D}.

7.1.1 States and Start States

The marks we put on messages and status are taken from the following set:

$$Flag \triangleq \{\texttt{OK}, \texttt{marked}\}$$

Variable	**Type**	**Initially**	**Description**
queue	$(Msg \times Flag)^*$	ε	The list of messages in the system. Each message has an associated *flag*. If the flag value is `marked`, the message might be lost in a subsequent $drop(I)$ action.
rec_s	Bool	*false*	*true* iff the sender has crashed and not yet recovered.
rec_r	Bool	*false*	*true* iff the receiver has crashed and not yet recovered.
status	$Stat \times Flag$	$(false, \texttt{OK})$	Indicates the status of the last message sent. If the associated flag is `marked`, the status might be changed to *false* in a subsequent $drop(I)$ action.

We use the normal record notation to extract components of a value or variable. For instance, *status.stat* and *status.flag* extract the status value and status flag from *status*.

We say that *status* is marked if $status.flag = \texttt{marked}$, and correspondingly an element e of *queue* is marked if $e.flag = \texttt{marked}$. If en element of *queue* or the *status* is not marked, it is said to be OK or "not marked".

7.1.2 Actions

The input and output actions, i.e., the user interface, of A_{D} is, of course, the same as for A_{S}. A_{D} has the internal actions $mark(I)$, $unmark(I)$, and $drop(I)$.

Input:
  $send\_msg(m)$, $m \in Msg$
  $crash_s$
  $crash_r$
Output:
  $receive\_msg(m)$, $m \in Msg$
  $ack(b)$, $b \in$ Bool
  $recover_s$
  $recover_r$
Internal:

$mark(I)$, $I \subseteq \mathsf{N}$
$unmark(I)$, $I \subseteq \mathsf{N}$
$drop(I)$, $I \subseteq \mathsf{N}$

7.1.3 Steps

Here we present the steps of A_{D}. An explanation of the steps is offered below.

$send\_msg(m)$
Effect:
$queue := queue \,\hat{}\, (m, \mathtt{OK})$
$status := (?, \mathtt{OK})$

$ack(b)$
Precondition:
$status.stat = b$
Effect:
$status.flag = \mathtt{OK}$

$receive\_msg(m)$
Precondition:
$queue \neq \varepsilon \wedge$
$(head(queue)).msg = m$
Effect:
$queue := tail(queue)$
if $queue = \varepsilon \wedge status.stat = ?$ then
$status.stat := true$

$crash_s$
Effect:
$rec_s := true$

$crash_r$
Effect:
$rec_r := true$

$mark(I)$
Precondition:
$(rec_s = true \vee rec_r = true) \wedge I \subseteq dom(queue)$
Effect:
$queue := mark(queue, I)$
optionally $status.flag := \mathtt{marked}$

$recover_s$
Precondition:
$rec_s = true$
Effect:
$rec_s := false$

$recover_r$
Precondition:
$rec_r = true$
Effect:
$rec_r := false$

$unmark(I)$
Precondition:
$I \subseteq dom(queue)$
Effect:
$queue := unmark(queue, I)$
optionally $status.flag := \mathtt{OK}$

$drop(I)$
Precondition:
$I \subseteq \{i \mid i \in dom(queue) \wedge queue[i].flag = \mathtt{marked}\}$
Effect:
if $queue \neq \varepsilon \wedge maxidx(queue) \in I$ then
$status := (false, \mathtt{OK})$
else if $status.flag = \mathtt{marked}$ then
optionally $status := (false, \mathtt{OK})$
$queue := delete(queue, I)$

In the step rule for *drop* we use the function *delete*, which was defined in Chapter 6 and used in the definition of $lose(I)$ at the S level. The precondition of $drop(I)$ guarantees that only marked messages are deleted. The step rule for *mark* uses a function *mark*, which is intended to mark messages with indices in I. Formally, for any queue $q \in (Msg \times Flag)^*$ and any set $I \subseteq dom(q)$, define

$$mark(q, I) \triangleq \langle(\text{if } i \in I \text{ then } (q[i].msg, \texttt{marked}) \text{ else } q[i]) \mid i \in dom(q)\rangle$$

Similarly, the step rule for *unmark* uses the function *unmark* defined as

$$unmark(q, I) \triangleq \langle(\text{if } i \in I \text{ then } (q[i].msg, \texttt{OK}) \text{ else } q[i]) \mid i \in dom(q)\rangle$$

Furthermore, note that when a new message is put into *queue* (by *send_msg*(m)), the message and *status* get the flag `OK` to indicate that they cannot be lost (yet). In the definition of the *receive_msg*(m) steps it is seen that a message might be successfully delivered to the receiver even though it is marked. This is because a marked message only has the *possibility* of being deleted.

Recall from the definition of S that there are two ways in which *status* can be lost (i.e., get a status value of *false*), and both ways are described in the definition of *lose*(I) in A_{S}: 1) if the element at the end of the queue is deleted, then the status is *required* to be lost, and 2) in any *lose*(I) step the status *may* be lost.

In A_{D} a status flag of `marked` corresponds to point 2), i.e., that *status* may be lost. In the *mark*(I) steps of A_{D} permission is given to lose some messages and maybe *status*. Then in *drop*(I) steps of A_{D}, which does the actual deleting performed by *lose*(I) in A_{S}, *status* is required to be lost if the element at the end of *queue* is deleted, even though *status* is OK. This corresponds to point 1) above, where *status* is required to be lost. Steps labeled by *drop*(I) is, of course, always allowed to lose a marked *status*.

The effect clause in the definition of the *ack*(b) steps is explained as follows: suppose *status.stat* $=$? and that *status.flag* has been changed to `marked` during a crash (by *mark*(I)). In a subsequent *receive_msg*(m) step that empties *queue*, *status.stat* is changed to *true* which enables an *ack*$(true)$ action. After the *receive_msg*(m) step, *status* $= (true, \texttt{marked})$, so there is still a possibility of losing *status*. However, once a positive acknowledgement has been issued, the system must not lose *status* and start issuing negative acknowledgements. Remember from the S level that the system is only allowed to change its mind in this respect during a crash. Thus, by changing *status.flag* to `OK` in the *ack* steps, we disallow this change of mind. Note, that it would be too restrictive to change *status* to $(true, \texttt{OK})$ in *receive_msg*(m) since we want A_{D} to be as nondeterministic as possible, to allow as many implementations as possible.

Another point where we have made A_{D} very nondeterministic is in the way messages (and *status*) are marked and deleted. In a *mark*(I) step *some* messages are marked and in an *unmark*(I) step, which can happen at any time, *some* of the marked messages can be made OK again, and finally in a *drop*(I) step, *some* of the marked messages are deleted.

Here, again, the point is that we want A_{D} to be as nondeterministic as possible. Of course the effect of marking some elements could be obtained by a "deterministic" *mark* that marks everything followed by *unmark*(I). However, when performing simulation proofs from lower levels of abstraction, it is desirable, for clarity, to have as nondeterministic actions of A_{D} as possible. Thus, by removing nondeterminism from A_{D}, which could not jeopardize its correctness with respect to A_{S}, we might rule out some implementations and make the correctness proofs of other implementations more cumbersome.

7.1.4 Liveness

As at the S level, we specify liveness in terms of fairness. Specifically, the liveness condition L_{D} at the D level will be specified implicitly as an environment-free liveness formula Q_{D} for A_{D}. Q_{D} will be stated as a conjunction of four weak fairness formulas, two of which have associated forcing conditions. We do not require fairness on the actions $mark(I)$, $unmark(I)$, and $drop(I)$. Informally, we have the four weak fairness conjuncts:

1. $ack(b)$ actions

   Forcing condition: $rec_s = rec_r = false$

2. $receive\_msg(m)$ actions

   Forcing condition: $rec_s = rec_r = false$

3. $recover_s$

4. $recover_r$

This ensures the same liveness as at the S level. Formally, let

$$
\begin{aligned}
C_{\mathrm{D},1} &\triangleq \{ack(true), ack(false)\} \\
C_{\mathrm{D},2} &\triangleq \{receive\_msg(m) \mid m \in Msg\} \\
C_{\mathrm{D},3} &\triangleq \{recover_s\} \\
C_{\mathrm{D},4} &\triangleq \{recover_r\}
\end{aligned}
$$

Then the formalization of Q_{D} is

$$
\begin{aligned}
Q_{\mathrm{D}} \triangleq\; & WF(C_{\mathrm{D},1}, rec_s = false \wedge rec_r = false) \wedge \\
& WF(C_{\mathrm{D},2}, rec_s = false \wedge rec_r = false) \wedge \\
& WF(C_{\mathrm{D},3}) \wedge \\
& WF(C_{\mathrm{D},4})
\end{aligned}
$$

By Lemma 4.7, Q_{D} is an environment-free liveness formula for A_{D}. Thus, $\mathrm{D} = (A_{\mathrm{D}}, L_{\mathrm{D}})$ is a live I/O automaton. Furthermore, by Lemma 4.8, Q_{D} is stuttering-insensitive.

This concludes the Delayed-Decision Specification of the at-most-once message delivery problem and attention is now turned towards proving that D correctly implements S.

7.2 Correctness of D

In this section we prove that $\mathrm{D} = (A_{\mathrm{D}}, L_{\mathrm{D}})$ is a correct implementation of our specification $\mathrm{S} = (A_{\mathrm{S}}, L_{\mathrm{S}})$. First we give some invariants of A_{D}. Then we prove, by means of an image-finite backward simulation, that A_{D} safely implements A_{S}, and finally we use this simulation result to prove that D correctly implements S.

7.2.1 Invariants

We only need one invariant in the proof. The invariant should be understood as the conjunction of the two parts.

Invariant 7.1

1. if $status.stat = ?$ then $queue \neq \varepsilon$
2. if $status.stat = true$ then $queue = \varepsilon$

Proof

By a simple inductive argument, it is easily proven that all reachable states of A_{D} satisfy the two parts of the invariant, so we omit the proof here. At the lower levels of abstraction we will give examples of proofs of more interesting invariants.

■

Below, we refer to this invariant by I_{D}.

7.2.2 Safety

To show that A_{D} safely implements A_{S}, we show the existence of an image-finite backward simulation from A_{D} to A_{S} with respect to some invariants. However, before we can do this we need a few preliminary definitions and lemmas.

Below we let q_{D} be a queue at the D level, i.e., $q_{\mathrm{D}} \in (Msg \times Flag)^*$, and let q_{S} be a queue at the S level, i.e., $q_{\mathrm{S}} \in Msg^*$.

Definition 7.2 (Explanation)

Define an *explanation* from q_{S} to q_{D} to be any mapping $f : dom(q_{\mathrm{S}}) \rightarrow dom(q_{\mathrm{D}})$ that satisfies the following four conditions

1. f is total
2. f is strictly increasing
3. $\forall i \in dom(q_{\mathrm{D}}) \setminus rng(f) : q_{\mathrm{D}}[i].flag = \mathtt{marked}$
4. $\forall i \in dom(q_{\mathrm{S}}) : q_{\mathrm{D}}[f(i)].msg = q_{\mathrm{S}}[i]$

■

Basically, if there exists an explanation from q_{S} to q_{D}, this means that q_{S} can be obtained from q_{D} by first deleting some of the marked elements of q_{D} and then removing the flags from the remaining elements.

Lemma 7.3

Let f be an explanation from q_{S} to q_{D}. Then $|q_{\mathrm{S}}| \leq |q_{\mathrm{D}}|$.

Proof

Suppose $|q_{\mathrm{S}}| > |q_{\mathrm{D}}|$. Then it is impossible to find a mapping from $dom(q_{\mathrm{S}})$ to $dom(q_{\mathrm{D}})$ that is total and strictly increasing, thus Conditions 1 and 2 of Definition 7.2 are violated. Hence, we can conclude $|q_{\mathrm{S}}| \leq |q_{\mathrm{D}}|$.

■

Now, define $\#_{\mathtt{OK}}(q_{\mathrm{D}})$ to be the number of elements e of q_{D} with $e.\mathit{flag} = \mathtt{OK}$. Thus, formally

$$\#_{\mathtt{OK}}(q_{\mathrm{D}}) \triangleq |q_D \upharpoonright (Msg \times \{\mathtt{OK}\})|$$

Lemma 7.4

Let f be an explanation from q_{S} to q_{D}. Then $|q_{\mathrm{S}}| \geq \#_{\mathtt{OK}}(q_{\mathrm{D}})$.

Proof

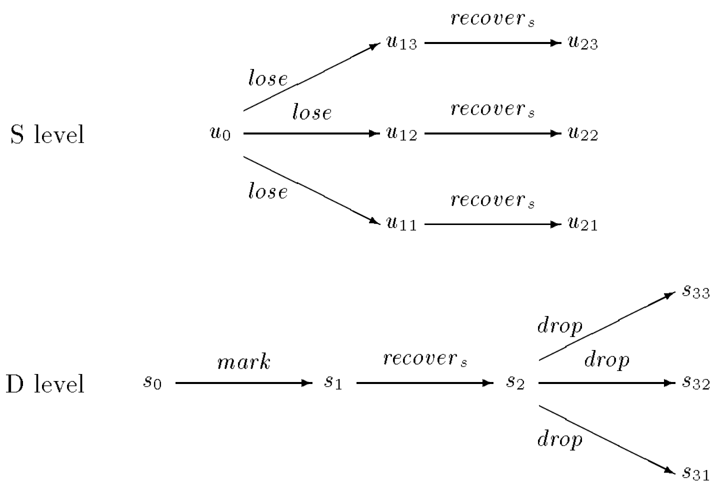
Suppose $|q_{\mathrm{S}}| < \#_{\mathtt{OK}}(q_{\mathrm{D}})$. Then Conditions 1 and 2 of Definition 7.2 give us that $|rng(f)| = |q_{\mathrm{S}}|(< \#_{\mathtt{OK}}(q_{\mathrm{D}}))$, so there must exist indices i in q_{D} such that $q_{\mathrm{D}}[i].\mathit{flag} = \mathtt{OK}$ and $i \notin rng(f)$. But this contradicts Condition 3 of Definition 7.2. Hence, we can conclude $|q_{\mathrm{S}}| \geq \#_{\mathtt{OK}}(q_{\mathrm{D}})$.

■

We are now ready to define a relation B_{DS} over $states(A_{\mathrm{D}}) \times states(A_{\mathrm{S}})$. In Lemma 7.11 below we prove that B_{DS} is an image-finite backward simulation from A_{D} to A_{S}.

However, before we give the actual definition of B_{DS}, it might be appropriate to discuss how to define a backward simulation in general. What states should be related? Let us give some guide-lines in terms of A_{D} and A_{S} in this example.

Recall that a backward simulation is needed when an implementation postpones some nondeterminism of the specification. The deletion of messages during a crash in A_{S} can in A_{D} be postponed until after recovery, which indicates that we need a backward simulation from A_{D} to A_{S}. (It is impossible to find a forward simulation from A_{D} to A_{S}. See, e.g., [LV92] for details.) This situation is shown—in a simplified way—in the following picture.



The *mark* step of A_{D} marks some messages, and after recovery some of the marked messages can be deleted by the nondeterministic *drop* steps. In this simplified example we assume that there are three ways of deleting messages, leading to states s_{31}, s_{32}, and s_{33}.[1] In A_{S} this scenario corresponds to *lose* having the "same" three ways of deleting messages, leading to states u_{11}, u_{12}, and u_{13}, followed by recovery.

[1] When dealing with two levels of abstraction, we always let s range over the states of the concrete level and u over the states of the abstract level.

It seems fairly intuitive that B_{DS} should relate s_{3i} to u_{2i} for $1 \leq i \leq 3$. But what about s_2? Well, s_2 is the state right after A_{D} has recovered, so it should be related to states after A_{S} has recovered. Thus, we are down to u_{21}, u_{22}, and u_{23}. Now the point is that s_2 actually corresponds to *all* of these states. In some sense B_{DS} offers an *explanation* of the nondeterminism occurring after s_2 by saying that this nondeterminism corresponds to some previous nondeterminism of A_{S}, which has led to one of the states u_{21}, u_{22}, or u_{23}.

To check that B_{DS} is a backward simulation from A_{D} to A_{S} we have, among other things, to verify that each step of A_{D} corresponds to a sequence of steps of A_{S} with the same trace. More specifically, consider, e.g., the step $(s_2, \mathit{drop}, s_{32})$ of A_{D}. According to Condition 3 of Definition 5.3, we have to verify that for each state of A_{S} that is related to s_{32}, here only u_{22}, there *exists* a state u of A_{S} such that there is a sequence of steps from u to u_{22} with an empty trace (since *drop* is internal). But here we can just choose u to be u_{22}. This makes the sequence of steps in A_{S} empty which certainly has an empty trace.

For s_1 we can use similar arguments and find that s_1 should be related to all of the states u_{11}, u_{12}, and u_{13}. Now, consider the step $(s_1, \mathit{recover}_s, s_2)$ of A_{D}. Again, we have to consider every state that is related to s_2. Let this state be u_{2i} for some arbitrary $1 \leq i \leq 3$. We then have to find some state u related to s_1 such that there is a sequence of steps from u to u_{2i} with the trace $\mathit{recover}_s$. But here we just choose $u = u_{1i}$, and since, for all $1 \leq i \leq 3$, $(u_{1i}, \mathit{recover}_s, u_{2i})$ is a step of A_{S}, we are done.

Finally, of course, s_0 should be related to u_0.

The above example offers some guide-lines when defining backward simulations, and even though the real B_{DS} from A_{D} to A_{S} is more complicated—mainly because of the nondeterminism involved with the *status* and the connection between *queue* and *status*—the recipe is the same:

> To any state s of A_{D}, we have to relate all states u of A_{S} that could have resulted from some nondeterminism of A_{S} that "corresponds" to nondeterminism that may happen after state s of A_{D}.

Of course, one has to use ones intuition about the safe I/O automata in question in order to identify the "corresponding" nondeterminism.

B_{DS} can now be defined and motivated.

Definition 7.5 (Image-Finite Backward Simulation from A_{D} to A_{S})

If $s \in \mathit{states}(A_{\mathrm{D}})$ and $u \in \mathit{states}(A_{\mathrm{S}})$, then define that $(s, u) \in B_{\mathrm{DS}}$ if there exists an explanation f from $u.\mathit{queue}$ to $s.\mathit{queue}$ such that the following conditions hold:

1. $u.\mathit{rec}_s = s.\mathit{rec}_s$ and $u.\mathit{rec}_r = s.\mathit{rec}_r$
2. $u.\mathit{status} \in$
   if $s.\mathit{status}.\mathit{flag} = \mathtt{OK} \wedge (s.\mathit{queue} = \varepsilon \vee (\mathit{last}(s.\mathit{queue})).\mathit{flag} = \mathtt{OK})$ then $\{s.\mathit{status}.\mathit{stat}\}$
   else $\{s.\mathit{status}.\mathit{stat}, \mathit{false}\}$
3. if $u.\mathit{status} = ? \wedge s.\mathit{queue} \neq \varepsilon$ then $\mathit{maxidx}(s.\mathit{queue}) \in \mathit{rng}(f)$

We say that an explanation from $u.\mathit{queue}$ to $s.\mathit{queue}$ is a *valid explanation* from u to s provided that Conditions 1–3 are satisfied.

■

Note, that $(s, u) \in B_{\mathrm{DS}}$ iff there exists a valid explanation from u to s.

The requirement that there has to be an explanation from $u.queue$ to $s.queue$ in order for $(s, u) \in B_{\mathrm{DS}}$ is a generalization of the example above. Thus, all states u related to s have queues that can be obtained by deleting some marked messages from $s.queue$ and removing the flags from the remaining elements.

Condition 1 gives the straight-forward correspondence between the *rec* flags of A_{D} and A_{S}.

Condition 2 deals with the status. In A_{D} there are two ways of losing *status* (i.e., changing *status.stat* to *false*), and both situations are described in the specification of the *drop* steps of A_{D}: either the element at the end of *queue* gets deleted, in which case *status must* be lost, or *status* is marked, in which case *status may* be lost. Alternatively, we can say that if $status.flag = \mathtt{OK}$ and either *queue* is empty or its last element is `OK`, the *status* cannot be changed by a *drop* step. Thus, in this case we are not in a situation where A_{D} is "waiting" to perform some nondeterminism on *status*, which has already been performed by A_{S}. If, on the other hand, *status* is marked or the last element on *queue* is marked, *drop* may lead to loss of *status*, and this corresponds to a loss at the S level, which has already occurred in a *lose* step of S. Thus, in this situation B_{DS} should allow the corresponding state at the S level to have $status = false$. This explains Condition 2.

Finally, Condition 3 in the definition of B_{DS} is a consistency condition between the explanation f and the value chosen for $u.status$. The condition should intuitively ensure that whenever the last element of $s.queue$ is not in the range of f, i.e., when f states that u describes a situation where the last element of *queue* has been lost, then $u.status$ must reflect this by having the value *false*. Thus, the condition should limit the number of legal combinations of $u.queue$ and $u.status$ due to the fact that these values are not always independent. The condition could initially be written as

$$\text{if } s.queue \neq \varepsilon \wedge maxidx(s.queue) \notin rng(f) \text{ then } u.status = false$$

Taking the contrapositive of this condition gives us

$$\text{if } u.status \neq false \text{ then } s.queue = \varepsilon \vee maxidx(s.queue) \in rng(f)$$

Now, if $u.status = true$ then Condition 2 gives us that also $s.status.stat = true$. Invariant 7.1 Part 2 then implies that $s.queue$ is empty. Thus, if $u.status = true$, the condition is trivially satisfied. So we only need to deal with the case where $u.status = ?$ and this is exactly Condition 3 of the definition in a slightly rewritten form.

Note, that in defining B_{DS} we have used our intuition about A_{S} and A_{D}. It is not at all sure that a first attempt to define a simulation relation is correct. However, any errors in the definition will be caught in the subsequent simulation proof and lead to a revised definition, and so on. For instance, the consistency condition (Condition 3) in the definition of B_{DS} was added during a proof attempt that failed. In Lemma 7.11 below we prove that B_{DS} is in fact an image-finite backward simulation from A_{D} to A_{S}.

The following lemmas make the main simulation proof shorter.

Lemma 7.6

Let $s \in states(A_{\mathrm{D}})$ and $q \in Msg^*$ such that there exists an explanation from q to $s.queue$. Then there exists a state $u \in states(A_{\mathrm{S}})$ with $u.rec_s = s.rec_s$, $u.rec_r = s.rec_r$, $u.queue = q$, and $(s, u) \in B_{\mathrm{DS}}$.

Proof

Let f be an arbitrary explanation from q to $s.queue$ and let $u.rec_s = s.rec_s$, $u.rec_r = s.rec_r$, and $u.queue = q$. We must show that we can define $u.status$ such that Conditions 1–3 of Definition 7.5 are satisfied.

Condition 1 is trivially satisfied.

We now consider cases, in each case defining $u.status$ and showing that Conditions 2 and 3 are satisfied.

1. $s.queue = \varepsilon$

   Define $u.status = s.status.stat$. Then Conditions 2 and 3 are vacuously satisfied.

2. $s.queue \neq \varepsilon$

   (a) $(last(s.queue)).flag = \texttt{marked}$
   Define $u.status = false$. This satisfies Conditions 2 and 3, the latter vacuously.

   (b) $(last(s.queue)).flag = \texttt{OK}$
   Define $u.status = s.status.stat$. Then Condition 2 is vacuously satisfied.
   Now, assume that $maxidx(s.queue) \notin rng(f)$. Then Condition 3 of Definition 7.2 of an explanation says that $s.queue[maxidx(s.queue)].flag = \texttt{marked}$ which is the same as $(last(s.queue)).flag = \texttt{marked}$, but this contradicts the assumptions in this subcase. Hence we have that $maxidx(s.queue) \in rng(f)$. Thus Condition 3 is satisfied.

■

Now, define the total function $maxqueue : (Msg \times Flag)^* \rightarrow Msg^*$ such that for any queue q_D in the domain, $maxqueue(q_D)$ is defined to be the queue q_S obtained by removing all flag components from q_D. Formally, we have

$$q_S = maxqueue(q_D) \quad \text{iff} \quad |q_S| = |q_D| \quad \text{and} \quad \forall i \in dom(q_D) : q_S[i] = q_D[i].msg$$

Lemma 7.7

The identity mapping f from $dom(q_D)$ to $dom(q_D)$ is an explanation from $maxqueue(q_D)$ to q_D.

Proof

We check Conditions 1–4 of Definition 7.2 of an explanation. Since the identity mapping is both total and strictly increasing Conditions 1 and 2 are satisfied. Condition 3 is vacuously satisfied since $rng(f) = dom(q_D)$. From the definition of $maxqueue$ we directly see that also Condition 4 is satisfied.

■

Lemma 7.8

Let $s \in states(A_D)$. Then there exists a state $u \in states(A_S)$ with $u.rec_s = s.rec_s$, $u.rec_r = s.rec_r$, and $u.queue = maxqueue(s.queue)$, such that $(s, u) \in B_{DS}$.

Proof

Let $q_S = maxqueue(s.queue)$. Then by Lemma 7.7 there exists an explanation (namely the identity mapping) from q_S to $s.queue$. Lemma 7.6 then gives us the existence of a state u with $u.rec_s = s.rec_r$, $u.rec_r = s.rec_r$, and $u.queue = q_S$ such that $(s, u) \in B_{DS}$. That suffices.

■

Corollary 7.9

Let $s \in states(A_D)$. Then there exists a state $u \in states(A_S)$ such that $(s, u) \in B_{DS}$.

Proof

Immediate from Lemma 7.8.

■

We state the following trivial lemma without proof.

Lemma 7.10

Let q_D be an element of $(Msg \times Flag)^*$. Then, any element q_S of Msg^*, such that there exists an explanation from q_S to q_D, can be obtained from $maxqueue(q_D)$ by deleting some elements.

■

We can now state and prove the main result of this section, namely that the relation B_{DS} defined in Definition 7.5 is an image-finite backward simulation from A_D to A_S (with respect to I_D (Invariant 7.1) and *true*). The style of the proof is careful mathematical reasoning.

Lemma 7.11

$A_D \leq_{iB} A_S$ via B_{DS}.

Proof

We prove that B_{DS} is an image-finite backward simulation from A_D to A_S with respect to I_D and *true*. We first show that B_{DS} is image-finite and then check the three conditions (which we call nonemptiness, base case, and inductive case, respectively) of Definition 5.3.

Image-Finiteness

Let s be an arbitrary state of A_D. We must show that there exists only finitely many states u of A_S such that $(s, u) \in B_{DS}$. Since rec_s, rec_r, and *status* can only take on finitely many values in A_S these variables cannot give rise to problems. It now remains to be shown that for a fixed but arbitrary s also *queue* (in S) can only take on finitely many values. For (s, u) to be in B_{DS} there must exist an explanation from $u.queue$ to $s.queue$. Lemma 7.3 gives us that $|u.queue| \leq |s.queue|$, thus there are only a finite number of lengths to choose from (since $s.queue$ is a finite queue). Also, there exists only a finite number of mappings (explanations) between two finite domains. Condition 4 of Definition 7.2 finally gives us that the elements of the possible $u.queue$ values are uniquely determined by $s.queue$ and the (finitely many) explanations. Hence, $u.queue$ can only take on finitely many values given s. That suffices.

Nonemptiness

Corollary 7.9 immediately gives the result.

Base Case

Let s_0 be the (unique) start state of $A_{\rm D}$. Then if $(s, u) \in B_{\rm DS}$, then $u.rec_s = s.rec_s = \mathit{false}$, $u.rec_r = s.rec_r = \mathit{false}$, $u.status = s.status.stat = \mathit{false}$ (since $s.status.\mathit{flag} = \mathtt{OK}$ and $s.queue = \varepsilon$), and $u.queue = \varepsilon$ (since the existence of an explanation from $u.queue$ to $s.queue$ and the fact that $s.queue = \varepsilon$ implies that $u.queue = \varepsilon$.) Thus, u is the unique start state of $A_{\rm S}$. That suffices.

Inductive Case

Assume $(s, a, s') \in steps(A_{\rm D})$ such that s and s' satisfy $I_{\rm D}$ (Invariant 7.1), and let u' be an arbitrary state of $A_{\rm S}$ such that $(s', u') \in B_{\rm DS}$. Below we consider cases based on a (and sometimes sub-cases of each case) and for each (sub)case we define a finite execution fragment α of $A_{\rm S}$ with $lstate(\alpha) = u'$, $(s, fstate(\alpha)) \in B_{\rm DS}$, and $trace(\alpha) = trace(a)$. In this particular proof all execution fragments will be of length zero or one. Thus, in each (sub)case we will either

- define an action $b \in acts(A_{\rm S})$ and a state $u \in states(A_{\rm S})$, such that $(u, b, u') \in steps(A_{\rm S})$, $(s, u) \in B_{\rm DS}$, and $trace(b) = trace(a)$, or
- show that $(s, u') \in B_{\rm DS}$ and a is internal.

In the former case, we show that $(u, b, u') \in steps(A_{\rm S})$ by showing that all four state variables of $A_{\rm S}$ are related in u and u' according to the definition of the b steps of $A_{\rm S}$.

In the proof, when we refer to Conditions 1–3, we mean Conditions 1–3 of Definition 7.5 of $B_{\rm DS}$ unless otherwise specified.

$\underline{a = send\_msg(m)}$

In this case we show that we can define u such that $(u, send\_msg(m), u') \in steps(A_{\rm S})$ and $(s, u) \in B_{\rm DS}$. Clearly the step has the right trace.

We have $s'.queue = s.queue \,\hat{}\, (m, \mathtt{OK})$ and $s'.status = (?, \mathtt{OK})$. Lemma 7.4 implies $u'.queue \neq \varepsilon$.

$$\begin{aligned} \text{Define } u.rec_s &= s.rec_s \\ u.rec_r &= s.rec_r \\ u.queue &= init(u'.queue) \end{aligned}$$

First we find an explanation from $u.queue$ to $s.queue$. Let f' be a valid explanation from u' to s'. (Such a valid explanation exists since $(s', u') \in B_{\rm DS}$). Since $last(s'.queue).\mathit{flag} = \mathtt{OK}$, we have from Lemma 7.4 and Conditions 1–3 of Definition 7.2 of an explanation that $f'(maxidx(u'.queue)) = maxidx(s'.queue)$. Then $f = f' \restriction dom(u.queue)$ is clearly an explanation from $u.queue$ to $s.queue$.

Now, by Lemma 7.6, define $u.status$ such that $(s, u) \in B_{\rm DS}$.

It remains to show that $(u, send\_msg(m), u') \in steps(A_{\rm S})$:

rec_s and rec_r:

From the definition of the $send\_msg(m)$ steps of $A_{\rm D}$, the definition of u, and the fact that $(s', u') \in B_{\rm DS}$, we have that $u'.rec_s = s'.rec_s = s.rec_s = u.rec_s$ and correspondingly for rec_r. This is as required by the definition of the $send\_msg(m)$ steps of $A_{\rm S}$.

status:

Since $(s', u') \in B_{\mathrm{DS}}$, Condition 2 implies that $u'.status = ?$. No matter what the value of $u.status$ is, this is as required by the definition of the $send\_msg(m)$ steps of A_{S}.

queue:

We have $u'.queue \neq \varepsilon$ (by Lemma 7.4) and $last(u'.queue) = m$ (by use of Definition 7.2 of an explanation). Then, by definition, we have $u'.queue = init(u'.queue) \mathbin{\hat{}} last(u'.queue) = u.queue \mathbin{\hat{}} m$. Again, this is as required by the definition of the $send\_msg(m)$ steps of A_{S}.

$\underline{a = crash_s}$

Define
$$\begin{array}{lcl} u.rec_s & = & s.rec_s \\ u.rec_r & = & u'.rec_r \\ u.status & = & u'.status \\ u.queue & = & u'.queue \end{array}$$

Then it is easy to see that $(s, u) \in B_{\mathrm{DS}}$ (any valid explanation from u' to s' is also a valid explanation from u to s) and that $(u, crash_s, u') \in steps(A_{\mathrm{S}})$.

$\underline{a = crash_r}$

Similar to the case $a = crash_s$.

$\underline{a = receive\_msg(m)}$

In this case we define u such that $(u, receive\_msg(m), u') \in steps(A_{\mathrm{S}})$ and $(s, u) \in B_{\mathrm{DS}}$. Clearly the step has the right trace.

From the definition of the $receive\_msg(m)$ steps of A_{D} we have that $s.rec_r = s'.rec_r$, $s.rec_s = s'.rec_s$, $s.queue \neq \varepsilon$ with $(head(s.queue)).msg = m$ and $s'.queue = tail(s.queue)$.

Define
$$\begin{array}{lcl} u.rec_s & = & s.rec_s \\ u.rec_r & = & s.rec_r \\ u.queue & = & m \mathbin{\hat{}} u'.queue \end{array}$$

We first find an explanation from $u.queue$ to $s.queue$. Let f' be any valid explanation from u' to s' (we know it exists), and define f in the following way:

$$f = [(i+1) \mapsto (f'(i)+1) \mid i \in dom(f')] \cup [0 \mapsto 0]$$

Intuitively f relates the same elements in $u.queue$ and $s.queue$ that were related by f' in $u'.queue$ and $s'.queue$ (these elements all have their indices increased by one because of the new elements at the head of the queues), and relates these new messages m. Based on the fact that f' is an explanation from $u'.queue$ to $s'.queue$, it is easy to check that f is an explanation from $u.queue$ to $s.queue$.

We consider cases, in each case defining $u.status$, showing $(s, u) \in B_{\mathrm{DS}}$ by showing that Conditions 2–3 hold (Condition 1 clearly holds) and showing that $(u, receive\_msg(m), u') \in steps(A_{\mathrm{S}})$. For the latter part it is easy to see that a $receive\_msg(m)$ step is enabled in u and that rec_s, rec_r and $queue$ are handled correctly. So all we need to do is to show that also $status$ is handled correctly in the $receive\_msg(m)$ step of A_{S}.

1. $s.status.stat = true$
   Invariant 7.1 Part 2 implies that this situation cannot occur.
2. $s.status.stat = false$
   Define $u.status = false$.
   Then clearly $(s, u) \in B_{\mathrm{DS}}$ (Conditions 2 and 3 are vacuously satisfied)
   status:
   Since $s.status.stat = false$, we have $s'.status = s.status$, so $u'.status = false$. Leaving $status = false$ unchanged is permitted by the definition of the $receive\_msg(m)$ steps in A_{S}.
3. $s.status.stat = ?$
   (a) $u'.queue \neq \varepsilon$
   Then also $s'.queue \neq \varepsilon$ (by Lemma 7.3) so from the definition of $receive\_msg(m)$ in A_{D} we have $s'.status = s.status$.
   Define $u.status = u'.status$.
   Condition 2:
   Since (s', u') satisfies Condition 2, also (s, u) satisfies that condition. (Neither the emptiness of *queue*, *status.flag*, nor the flag of the last element in *queue* are changed in the step in A_{D}).
   Condition 3:
   Assume that $u.status (= u'.status) = ?$. Since $s.queue \neq \varepsilon$, we must show that $maxidx(s.queue) \in rng(f)$. Since $s'.queue \neq \varepsilon$, and (s', u') and f' satisfy Condition 3, we have $maxidx(s'.queue) \in rng(f')$, so from the construction of f, it is easy to see that $maxidx(s.queue) \in rng(f)$.
   status:
   Leaving *status* unchanged is as required by the definition of $receive\_msg(m)$ in A_{S} since we assume that $u'.queue \neq \varepsilon$.
   (b) $u'.queue = \varepsilon$
   i. $s'.queue = \varepsilon$
   Then the definition of $receive\_msg(m)$ in A_{D} implies that $s'.status.stat = true$ and $s'.status.flag = s.status.flag$. Then, by definition of B_{DS}, $u'.status$ is either *true* or *false*. We consider cases.
   A. $s'.status.flag = \mathtt{OK}$ or ($s'.status.flag = \mathtt{marked}$ and $u'.status = true$)
   If $s'.status.flag = \mathtt{OK}$, then by Condition 2 we also have $u'.status = true$ since $s'.status.stat = true$.
   Define $u.status = ?$ ($= s.status.stat$).
   Condition 2:
   Vacuously satisfied by (s, u).
   Condition 3:
   Since $s'.queue = \varepsilon$, we have $|s.queue| = 1$. Now, since $f(0) = 0$, we have $maxidx(s.queue) \in rng(f)$ as required.
   status:
   Changing *status* from ? to *true* when $u'.queue = \varepsilon$ is as required by the definition of $receive\_msg(m)$ in A_{S}.
   B. $s'.status.flag = \mathtt{marked}$ and $u'.status = false$
   Define $u.status = false$.
   Condition 2:

Since $s.status.flag = s'.status.flag = false$, we have that (s, u) satisfies Condition 2.

Condition 3:

Vacuously satisfied.

status:

Leaving $status = false$ unchanged is allowed by $receive\_msg(m)$ in A_S.

ii. $s'.queue \neq \varepsilon$

The definition of $receive\_msg(m)$ in D implies $s'.status.stat = s.status.stat = ?$ and $s'.status.flag = s.status.flag$. Since $u'.queue = \varepsilon$, $s'.queue \neq \varepsilon$, and (s', u') and f' satisfy Condition 3, we get that $u'.status \neq ?$ (f' must be empty). Note, that this is one of the two places in the entire proof where we need the consistency condition (Condition 3). Condition 2 now gives us that $u'.status = false$ and that either $s'.status.flag =$ `marked` or $(last(s'.queue)).flag =$ `marked`.

Define $u'.status = false$.

Condition 2:

Since $s.status.flag = s'.status.flag$, $(last(s'.queue)).flag = (last(s.queue)).flag$, and one of these flag values is `marked`, we see that (s, u) satisfies Condition 2.

Condition 3:

Vacuously satisfied.

status:

Leaving $status = false$ unchanged is allowed by the definition of $receive\_msg(m)$ in A_S.

$\underline{a = ack(b)}$

In this case we define u such that $(u, ack(b), u') \in steps(A_S)$ and $(s, u) \in B_{DS}$. Clearly the step has the right trace.

From the definition of $ack(b)$ in A_D, we have that $s.status.stat = b$ and that $s' = s$ except that s' and s may differ on the value of $status.flag$, which is set to `OK` in the step.

We consider cases based on the value of b.

1. $b = false$

   Then $u'.status = false$.

   Define $u = u'$.

   It is now easy to see that $(s, u) \in B_{DS}$. (The fact that s and s' may differ on the value of $status.flag$ could only cause troubles in Condition 2 but this is seen not to be the case since $s.status.stat = false$ implies that the only choice for $u.status$ is $false$ as we have defined it to be.)

   Now, since $u' = u$, we have $u.status = false$, Thus, an $ack(b)$ step is enabled in u. Again since $u = u'$, we now see that $(u, ack(b), u')$ is a step of A_S as required.

2. $b = true$

   Since $s.status.stat = s'.status.stat = true$, Invariant 7.1 Part 2 gives us that $s'.queue = \varepsilon$ and $s.queue = \varepsilon$. Furthermore, since $s'.status.flag =$ `OK`, we get from Condition 2 that $u'.status = true$.

   Define $u = u'$.

   As in the previous case clearly $(s, u) \in B_{DS}$ and $(u, ack(b), u') \in steps(A_S)$.

$a = recover_s$

$$\begin{aligned} \text{Define } u.rec_s &= \mathit{false} \\ u.rec_r &= u'.rec_r \\ u.status &= u'.status \\ u.queue &= u'.queue \end{aligned}$$

Since $u.rec_s = s.rec_s = \mathit{false}$, it is easy to see that $(s, u) \in B_{\mathrm{DS}}$ (any valid explanation from u' to s' is also a valid explanation from u to s) and that $(u, recover_s, u') \in steps(A_{\mathrm{S}})$ (and clearly has the right trace).

$a = recover_r$

Similar to the case $a = recover_s$.

$a = mark(I)$

In this case we define u and I' such that $(u, lose(I'), u') \in steps(A_{\mathrm{S}})$ and $(s, u) \in B_{\mathrm{DS}}$. Clearly the step has the right trace (the empty trace).

From the definition of the *mark* steps in A_{D} we have $s'.rec_s = s.rec_s$, $s'.rec_r = s.rec_r$, and either $s.rec_s = true$ or $s.rec_r = true$.

$$\begin{aligned} \text{Define } u.rec_s &= s.rec_s \\ u.rec_r &= s.rec_r \\ u.queue &= maxqueue(s.queue) \\ u.status &= s.status.stat \end{aligned}$$

By Lemma 7.7 the identity mapping f is an explanation from $u.queue$ to $s.queue$, and it is easy to show that f is a valid explanation from u to s. Thus, $(s, u) \in B_{\mathrm{DS}}$.

To show that $(u, lose(I'), u') \in steps(A_{\mathrm{S}})$, we first observe that since $(s, u) \in B_{\mathrm{DS}}$ we have $u.rec_s = true$ or $u.rec_r = true$, so a $lose(I')$ step is enabled in u.

rec_s and *rec_r*:
  $u'.rec_s = s'.rec_s = s.rec_s = u.rec_s$ and similarly for rec_r. This is as required by the definition of $lose(I')$ in A_{S}.

queue:
  First observe that $maxqueue(s.queue) = maxqueue(s'.queue)$. Then, since by definition we have $u.queue = maxqueue(s.queue)$, Lemma 7.10 implies that $u'.queue$ can be obtained from $u.queue$ by deleting some (possibly zero) elements. Thus, we can define I' accordingly, and this is as required by the definition of $lose(I')$ in A_{S}.

status:
  First note that since we might have $s'.status.flag = \texttt{marked}$, we also might have $u'.status = \mathit{false}$ by Condition 2, but since $lose(I')$ can always change *status* to *false* in A_{S}, this situation does not cause troubles.

  The situation that could cause troubles is if $u'.status \neq \mathit{false}$ but the $lose(I')$ step is required to change *status* to *false* because the element at the end of $u.queue$ must be deleted in order to treat *queue* correctly. We must show that this situation is impossible.

  Assume that $u'.status \neq \mathit{false}$. Then Condition 2 and the definition of $mark(I)$ in A_{D} give $u'.status = s'.status.stat = s.status.stat \neq \mathit{false}$. We consider cases.

1. $u'.status = s'.status.stat = s.status.stat = true$.
   Invariant 7.1 Part 2 implies $s.queue = s'.queue = \varepsilon$. Then Lemma 7.3 implies that $u.queue = u'.queue = \varepsilon$. Thus $I' = \emptyset$. That suffices.
2. $u'.status = s'.status.stat = s.status.stat = ?$.
   (a) $s.queue = \varepsilon$
   Similar to Case 1.
   (b) $s.queue \neq \varepsilon$
   Then Condition 3 and Definition 7.2 imply $f(maxidx(u.queue)) = maxidx(s.queue)$. It is now easy to see that $u'.queue$ can be obtained by deleting some elements, but not the element at the end, from $u.queue$. That suffices.

$a = unmark(I)$

In this case we show that $unmark(I)$ in A_{D} corresponds to an empty step in A_{S} (remember that $unmark(I)$ is internal). Thus, we show that $(s, u') \in B_{\mathrm{DS}}$.

From the definition of the $unmark(I)$ steps of A_{D}, we have that $s'.queue$ is obtained from $s.queue$ by changing some (maybe zero) flag values from `marked` to `OK`. Now, let f' be a valid explanation from u' to s'. Then by Definition 7.2 it is easy to see that f' is also an explanation from $u'.queue$ to $s.queue$. (The only interesting case is Condition 3 of Definition 7.2 but since messages that are marked in $s'.queue$ cannot be `OK` in $s.queue$, this case is easily checked).

We show that f' is a valid explanation from u' to s by checking Conditions 1–3.

Condition 1:
This condition is satisfied since the $unmark(I)$ step does not change rec_s and rec_r.
Condition 2:
The unmarking of $status$ and message flags might lead to the requirement that $u'.status = s'.status.stat$ (by Condition 2). But then obviously also (s, u') satisfies Condition 2 since both the "then" and the "else" in this condition allow $u'.status = s.status.stat (= s'.status.stat)$. The important thing to note here is that $unmark(I)$ cannot lead from a situation where the "then" clause must be chosen to a situation where the "else" clause must be chosen.
Condition 3:
Since Condition 3 does not mention any flag values, it is seen that (s, u') and f' satisfy this condition.

$a = drop(I)$

In this case we show that $drop$ corresponds to an empty step of A_{S}, i.e., that $(s, u') \in B_{\mathrm{DS}}$ (recall that $drop(I)$ is internal).

Let f' be an arbitrary valid explanation from u' to s'. We now construct an explanation f from $u'.queue$ to $s.queue$: I contains the indices of the elements of $s.queue$ that were deleted in the $drop(I)$ step. Then $|dom(s'.queue)| = |dom(s.queue) \setminus I|$. Now, let g be the (unique) bijective, strictly increasing mapping from $dom(s'.queue)$ to $dom(s.queue) \setminus I$. Informally g maps indices of elements in $s'.queue$ to the indices the same elements had in $s.queue$.

Define $f = g \circ f'$. To check that f is an explanation from $u'.queue$ to $s.queue$, we check Conditions 1–4 of Definition 7.2:

Conditions 1–2 of Definition 7.2:

Since f' is total and strictly increasing from $dom(u'.queue)$ to $dom(s'.queue)$ and g is total and strictly increasing from $dom(s'.queue)$ to $dom(s.queue) \setminus I$, f is total and strictly increasing from $dom(u'.queue)$ to $dom(s.queue)$.

Condition 3 of Definition 7.2:

We have that $dom(s.queue) \setminus rng(g \circ f') = I \cup g^{-1}(dom(s'.queue) \setminus rng(f'))$. This informally states if an element of $s.queue$ is not "hit" by f then this is because either the element is one of the elements that are deleted in the $drop(I)$ step or because the "corresponding" (by g) element in $s'.queue$ is not "hit" by f'. Now, all elements in $s.queue$ with indices in I are marked (by the precondition of $drop(I)$). Since f' is an explanation, all elements of $s'.queue$ with indices in $dom(s'.queue) \setminus rng(f')$ are marked, and since g and then also g^{-1} maps the index of an element to the index of the same element, we have that all elements of $s.queue$ with indices in $g^{-1}(dom(s'.queue \setminus rng(f')))$ are marked. That suffices.

Condition 4 of Definition 7.2:

By the fact that f' is an explanation (and therefore satisfies Condition 4) and the fact that g maps the index of an element to the index of the same element, it directly follows that f satisfies Condition 4 of Definition 7.2.

Thus, f is an explanation from $u'.queue$ to $s.queue$.

It now remains to show that f is a *valid* explanation from u' to s, i.e., we must check Conditions 1–3.

Condition 1:

Condition 1 is clearly satisfied (since neither rec_s nor rec_r are changed in the $drop(I)$ step and $(s', u') \in B_{\mathrm{DS}}$).

Condition 2:

We consider the ways *status* can change in the if-statement in the definition of the $drop(I)$ step.

Assume that the element at the end of $s.queue$ is deleted in the $drop(I)$ step. Then $s'.status = (\mathit{false}, \mathtt{OK})$ which implies $u'.status = \mathit{false}$. But in order to be able to delete the element at the end of $s.queue$ we have that $s.queue \neq \varepsilon$ and $(last(s.queue)).flag = \mathtt{marked}$, so (s, u') satisfies Condition 2.

Then assume that the element at the end of $s.queue$ is not deleted but that $u'.queue$ is changed to $(\mathit{false}, \mathtt{OK})$ since $s.status.flag = \mathtt{marked}$. Again we have $u'.status = \mathit{false}$, and since $s.status.flag = \mathtt{marked}$, we have that (s, u') satisfies Condition 2.

The last possibility is that *status* is not changed at all in the $drop(I)$ step, but then obviously (s, u') satisfies Condition 2 since (s', u') satisfies it.

Condition 3:

Assume $u'.status = ?$ and $s.queue \neq \varepsilon$. Since $u'.status = ?$ we must have $s'.status.stat = ?$ and then from the definition of the $drop(I)$ step we infer $s.status = s'.status$.

Then the element at the end of $s.queue$ is not deleted in the $drop(I)$ step (i.e., $maxidx(s.queue) \notin I$) since otherwise $s'.status = (\mathit{false}, \mathtt{OK})$. Thus, also $s'.queue \neq \varepsilon$. Since f' is a valid explanation from u' to s', Condition 3 gives us $maxidx(s'.queue) \in rng(f')$, and since $maxidx(s.queue) \notin I$ we must have $g(maxidx(s'.queue)) = maxidx(s.queue)$ since otherwise g could not be bijective and strictly increasing. All in all we get $maxidx(s.queue) \in rng(f)$, as required.

This concludes the simulation proof.

■

We can now prove that A_{D} safely implements A_{S}.

Theorem 7.12 (A_{D} **safely implements** A_{S})

$A_{\mathrm{D}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{S}}$

Proof

Directly by Lemma 7.11 and the soundness of image-finite backward simulations with respect to the safe implementation relation (Lemma 5.8).

■

7.2.3 Correctness

Before we can prove the main theorem of this chapter — that D is a correct implementation of S — we need to prove some basic lemmas about S and D. In the remainder of this chapter we use the following abbreviations.

$$\begin{array}{lcl} SM & = & \{send\_msg(m) \mid m \in Msg\} \\ RM & = & \{receive\_msg(m) \mid m \in Msg\} \end{array}$$

From the safe I/O automata A_{S} and A_{D} we get the following lemmas.

Lemma 7.13

$$A_{\mathrm{S}} \models \Box(\Box(status \in \mathsf{Bool}) \Longrightarrow \Box\neg\langle SM\rangle)$$

Proof

Immediate from the definition of A_{S} since any $send\_msg(m)$ step would change $status$ to ?.

■

Lemma 7.14

1. $A_{\mathrm{D}} \models \Box(\Box\neg\langle SM\rangle \Longrightarrow \Box(|queue^{\circ}| \leq |queue|))$
2. $A_{\mathrm{D}} \models \Box(\langle RM\rangle \Longrightarrow |queue^{\circ}| = |queue| - 1)$

Proof

Immediate from the definition of A_{D} since only $send\_msg(m)$ steps can add elements to $queue$, and $receive\_msg(m)$ steps remove one element from $queue$.

■

The following two lemmas prove properties of live executions of D. The lemmas deal with live executions where, from some point on, no $send\_msg(m)$ actions occur and neither the sender nor the receiver is in recovery phase. Then, in the first lemma, we prove that eventually elements will be removed from $queue$, which, in the second lemma, is used to prove that $queue$ is eventually emptied.

The proofs of the lemmas introduce the way we write structured proofs of temporal properties of our systems. The proof style is due to Lamport. The following description is taken from [AL92b]:

We use hierarchically structured proofs. The theorem to be proved is statement $\langle 0\rangle 1$. The proof of statement $\langle i\rangle j$ is either an ordinary paragraph-style proof or the sequence of statements $\langle i+1\rangle 1$, $\langle i+1\rangle 2$, ... and their proofs. ... . Within a proof, $\langle k\rangle l$ denotes the most recent statement with that number. A statement has the form

ASSUME: *Assump* PROVE: *Goal*

which is abbreviated to *Goal* if there is no assumption. The assertion Q.E.D. in statement number $\langle i{+}1\rangle k$ of the proof of statement $\langle i\rangle j$ denotes the goal of statement $\langle i\rangle j$. The statement

CASE: *Assump*

is an abbreviation for

ASSUME: *Assump* PROVE: Q.E.D.

Within the proof of statement $\langle i\rangle j$, Assumption $\langle i\rangle$ denotes that statement's assumption, and Assumption $\langle i\rangle .k$ denotes the assumption's k^{th} item.

Lemma 7.15

$$L_{\mathrm{D}} \models \forall k : \Box(\Box(\neg\langle SM\rangle \wedge rec_s = \mathit{false} \wedge rec_r = \mathit{false}) \Longrightarrow ((|queue| = k \wedge k > 0) \leadsto |queue| < k))$$

Proof

ASSUME: $\alpha \in L_{\mathrm{D}}$
PROVE: $\alpha \models \forall k : \Box(\Box(\neg\langle SM\rangle \wedge rec_s = \mathit{false} \wedge rec_r = \mathit{false}) \Longrightarrow ((|queue| = k \wedge k > 0) \leadsto |queue| < k))$

$\langle 1\rangle 1$. ASSUME: $k \geq 0$
PROVE: $\alpha \models \Box(\Box(\neg\langle SM\rangle \wedge rec_s = \mathit{false} \wedge rec_r = \mathit{false}) \Longrightarrow ((|queue| = k \wedge k > 0) \leadsto |queue| < k))$

$\langle 2\rangle 1$. ASSUME: α_1 is an arbitrary suffix of α
PROVE: $\alpha_1 \models \Box(\neg\langle SM\rangle \wedge rec_s = \mathit{false} \wedge rec_r = \mathit{false}) \Longrightarrow ((|queue| = k \wedge k > 0) \leadsto |queue| < k)$

$\langle 3\rangle 1$. ASSUME: $\alpha_1 \models \Box(\neg\langle SM\rangle \wedge rec_s = \mathit{false} \wedge rec_r = \mathit{false})$
PROVE: $\alpha_1 \models (|queue| = k \wedge k > 0) \leadsto |queue| < k$

$\langle 4\rangle 1$. $\alpha_1 \models \Box\neg\langle SM\rangle \Longrightarrow \Box(|queue^{\circ}| \leq |queue|)$

PROOF: By Lemma 7.14 Part 1, Lemma 3.5 Part 1 and Rule **Par**.

$\langle 4\rangle 2$. $\alpha_1 \models \Box(|queue^{\circ}| \leq |queue|)$

PROOF: By $\langle 4\rangle 1$, Assumption $\langle 3\rangle$, and Rule **MP**.

$\langle 4\rangle 3$. $\alpha_1 \models \Box((|queue| = k \wedge k > 0) \Longrightarrow (|queue| = k \;\mathcal{W}\; |queue| < k))$

PROOF: By $\langle 4\rangle 2$.

$\langle 4\rangle 4$. $\alpha \models \mathit{WF}(RM, rec_s = \mathit{false} \wedge rec_r = \mathit{false})$

PROOF: By proof assumption ($\alpha \in L_{\mathrm{D}}$) and definition of Q_{D}, which induces L_{D}.

$\langle 4\rangle 5.$ $\alpha \models \Box\Diamond\neg(rec_s = false \wedge rec_r = false \wedge |queue| > 0) \vee \Box\Diamond\langle RM\rangle$

PROOF: By $\langle 4\rangle 4$, the definition of WF, and noting that $enabled(RM) = (|queue| \geq 0)$.

$\langle 4\rangle 6.$ $\alpha_1 \models \Box\Diamond\neg(rec_s = false \wedge rec_r = false \wedge |queue| > 0) \vee \Box\Diamond\langle RM\rangle$

PROOF: By $\langle 4\rangle 5$, Lemma 3.5 Part 1, and definition of disjunction.

$\langle 4\rangle 7.$ $\alpha_1 \models \Diamond\neg(rec_s = false \wedge rec_r = false \wedge |queue| > 0) \vee \Diamond\langle RM\rangle$

PROOF: By $\langle 4\rangle 6$, Rule **Par**, and the definition of disjunction.

$\langle 4\rangle 8.$ $\alpha_1 \models \Box(rec_s = false \wedge rec_r = false \wedge |queue| > 0) \Longrightarrow \Diamond\langle RM\rangle$

PROOF: By rewriting $\langle 4\rangle 7$.

$\langle 4\rangle 9.$ $\alpha_1 \models \Box(|queue| > 0) \Longrightarrow \Diamond\langle RM\rangle$

PROOF: By Assumption $\langle 3\rangle$, $\langle 4\rangle 8$, and Rule **MP**.

$\langle 4\rangle 10.$ $\alpha_1 \models (|queue| = k \wedge \langle RM\rangle) \leadsto |queue| < k$

PROOF: Implied by Lemma 7.14 Part 2.

$\langle 4\rangle 11.$ Q.E.D.

PROOF: By $\langle 4\rangle 3$, $\langle 4\rangle 9$, $\langle 4\rangle 10$, and Rule **Pro2**.

$\langle 3\rangle 2.$ Q.E.D.

PROOF: By $\langle 3\rangle 1$ and the definition of implication.

$\langle 2\rangle 2.$ Q.E.D.

By $\langle 2\rangle 1$ and Lemma 3.5 Part 2.

$\langle 1\rangle 2.$ Q.E.D.

PROOF: By $\langle 1\rangle 1$ and Lemma 3.5 Part 5.

■

Lemma 7.16

$$L_{\mathrm{D}} \models \Box(\Box(\neg\langle SM\rangle \wedge rec_s = false \wedge rec_r = false) \Longrightarrow \Diamond\Box(queue = \varepsilon))$$

Proof

ASSUME: $\alpha \in L_{\mathrm{D}}$
PROVE: $\alpha \models \Box(\Box(\neg\langle SM\rangle \wedge rec_s = false \wedge rec_r = false) \Longrightarrow \Diamond\Box(queue = \varepsilon))$

$\langle 1\rangle 1.$ ASSUME: α_1 is an arbitrary suffix of α
PROVE: $\alpha_1 \models \Box(\neg\langle SM\rangle \wedge rec_s = false \wedge rec_r = false) \Longrightarrow \Diamond\Box(queue = \varepsilon)$

$\langle 2\rangle 1.$ ASSUME: $\alpha_1 \models \Box(\neg\langle SM\rangle \wedge rec_s = false \wedge rec_r = false)$
PROVE: $\alpha_1 \models \Diamond\Box(queue = \varepsilon)$

$\langle 3\rangle 1.$ $\alpha_1 \models \forall k : ((|queue| = k \wedge k > 0) \leadsto |queue| < k)$

PROOF: By Lemma 7.15, Lemma 3.5 Parts 1, 5, and 6, and Rules **Par** and **MP**.

$\langle 3\rangle 2.$ $\alpha_1 \models \forall k : (k > 0 \Longrightarrow \exists k' : (k' < k \land (|queue| = k \leadsto |queue| = k')))$

PROOF: By $\langle 3\rangle 1$ and Lemma 3.5 Part 7.

$\langle 3\rangle 3.$ $\alpha_1 \models \Diamond(|queue| = 0)$

PROOF: By $\langle 3\rangle 2$ and Rule **Pro1**.

$\langle 3\rangle 4.$ $\alpha_1 \models \Box\neg\langle SM\rangle \Longrightarrow \Box(|queue^{\circ}| \leq |queue|)$

PROOF: By Lemma 7.14 Part 1, Lemma 3.5 Part 1 and Rule **Par**.

$\langle 3\rangle 5.$ $\alpha_1 \models \Box(|queue^{\circ}| \leq |queue|)$

PROOF: By $\langle 3\rangle 4$, Assumption $\langle 2\rangle$, and Rule **MP**.

$\langle 3\rangle 6.$ $\alpha_1 \models \forall k : \Box(|queue| = k \Longrightarrow (|queue| = k \;\mathcal{W}\; |queue| < k))$

PROOF: By $\langle 3\rangle 5$.

$\langle 3\rangle 7.$ $\alpha_1 \models \Box(|queue| = 0 \Longrightarrow (|queue| = 0 \;\mathcal{W}\; |queue| < 0))$

PROOF: By $\langle 3\rangle 6$ and Lemma 3.5 Part 6.

$\langle 3\rangle 8.$ $\alpha_1 \models \Box(|queue| = 0 \Longrightarrow \Box(|queue| = 0))$

PROOF: By $\langle 3\rangle 7$, the fact that $|queue| < 0$ is always *false*, and the definition of $\Box$.

$\langle 3\rangle 9.$ $\alpha_1 \models \Diamond\Box(|queue| = 0)$

PROOF: By $\langle 3\rangle 3$, $\langle 3\rangle 8$, and Rule **MP**1.

$\langle 3\rangle 10.$ Q.E.D.

PROOF: Directly by $\langle 3\rangle 9$.

$\langle 2\rangle 2.$ Q.E.D.

PROOF: By $\langle 2\rangle 1$ and definition of implication.

$\langle 1\rangle 2.$ Q.E.D.

PROOF: By $\langle 1\rangle 1$ and Lemma 3.5 Part 2.

■

An important advantage of this way of writing structured proofs of temporal properties is that at a first reading, one can concentrate on the first outermost levels of the proof. Once that has been understood, the details at lower levels can be considered.

The next lemma contains the main part of the proof that D correctly implements S. It states that for any B_{DS}-related executions of A_{D} and A_{S}, if the execution of A_{D} satisfies Q_{D} (the temporal formula which induces the liveness condition L_{D}), then the execution of A_{S} satisfies Q_{S} (the temporal formula which induces the liveness condition L_{S}). The proof will be a proof by cases based on a proof by contradiction: if we assume the execution of A_{S} is not live, this means that the execution does not satisfy one of the weak fairness formulas in the definition of Q_{S}. By considering the weak fairness formulas one by one and deriving a contradiction in each case, the result follows.

Lemma 7.17

Let $\alpha \in exec(A_{\mathrm{D}})$ and $\alpha' \in exec(A_{\mathrm{S}})$ be arbitrary executions of A_{D} and A_{S}, respectively, with $(\alpha, \alpha') \in B_{\mathrm{DS}}$. Assume $\alpha \models Q_{\mathrm{D}}$. Then $\alpha' \models Q_{\mathrm{S}}$.

Proof

We prove the conjecture by contradiction. Thus,

ASSUME: $\alpha' \not\models Q_{\mathrm{S}}$
PROVE: False

$\langle 1\rangle 1.$ $\alpha' \models \neg WF(C_{\mathrm{S},1}, rec_s = false \wedge rec_r = false) \vee \neg WF(C_{\mathrm{S},2}, rec_s = false \wedge rec_r = false) \vee \neg WF(C_{\mathrm{S},3}) \vee \neg WF(C_{\mathrm{S},4})$

PROOF: Immediate by the Assumption, definition of Q_{S}, and the Boolean operators.

$\langle 1\rangle 2.$ CASE: $\alpha' \models \neg WF(C_{\mathrm{S},1}, rec_s = false \wedge rec_r = false)$

$\langle 2\rangle 1.$ $\alpha' \models \Diamond\Box\neg\langle C_{\mathrm{S},1}\rangle \wedge \Diamond\Box(rec_s = false \wedge rec_r = false \wedge status \in \mathsf{Bool})$

PROOF: From Case Hypothesis $\langle 1\rangle$ by expanding WF and noting the fact that $enabled_{A_{\mathrm{S}}}(C_{\mathrm{S},1}) = (status \in \mathsf{Bool})$.

$\langle 2\rangle 2.$ $\alpha' \models \Diamond\Box\neg\langle C_{\mathrm{S},1}\rangle \wedge \Diamond\Box\neg\langle SM\rangle \wedge \Diamond\Box(rec_s = false \wedge rec_r = false \wedge status \in \mathsf{Bool})$

PROOF: By $\langle 2\rangle 1$, Lemma 7.13, and **MP1**.

$\langle 2\rangle 3.$ $\alpha \models \Diamond\Box\neg\langle C_{\mathrm{S},1}\rangle \wedge \Diamond\Box\neg\langle SM\rangle \wedge \Diamond\Box(rec_s = false \wedge rec_r = false)$

PROOF: By Lemmas 5.10 and 5.11 since $C_{\mathrm{S},1}$ consists of external actions and Definition 7.5 of B_{DS} implies that for all $(s, u) \in B_{\mathrm{DS}}$, if $u \models (rec_s = false \wedge rec_r = false)$ then $s \models (rec_s = false \wedge rec_r = false)$.

$\langle 2\rangle 4.$ $\alpha \models \Diamond\Box\neg\langle C_{\mathrm{S},1}\rangle \wedge \Diamond\Box(rec_s = false \wedge rec_r = false \wedge queue = \varepsilon)$

PROOF: By $\langle 2\rangle 3$, Lemma 7.16, and **MP1**.

$\langle 2\rangle 5.$ $\alpha \models \Diamond\Box\neg\langle C_{\mathrm{S},1}\rangle \wedge \Diamond\Box(rec_s = false \wedge rec_r = false \wedge status \in \mathsf{Bool})$

PROOF: By $\langle 2\rangle 4$ and Invariant 7.1 Part 1.

$\langle 2\rangle 6.$ $\alpha \models \neg WF(C_{\mathrm{D},1}, rec_s = false \wedge rec_r = false)$

PROOF: By $\langle 2\rangle 5$, the definition of WF, the fact that $C_{\mathrm{S},1} = C_{\mathrm{D},1}$ and the fact that $enabled_{A_{\mathrm{D}}}(C_{\mathrm{D},1}) = (status \in \mathsf{Bool})$.

$\langle 2\rangle 7.$ Q.E.D.

PROOF: By $\langle 2\rangle 6$, the assumption that $\alpha \models Q_{\mathrm{D}}$, and the definition of Q_{D}.

$\langle 1\rangle 3.$ CASE: $\alpha' \models \neg WF(C_{\mathrm{S},2}, rec_s = false \wedge rec_r = false)$

$\langle 2\rangle 1.$ $\alpha' \models (\Diamond\Box\neg\langle C_{\mathrm{S},2}\rangle \wedge \Diamond\Box(rec_s = false \wedge rec_r = false \wedge queue \neq \varepsilon))$

PROOF: By expanding WF and noting that $enabled_{A_{\mathrm{S}}}(C_{\mathrm{S},2}) = (queue \neq \varepsilon)$.

$\langle 2\rangle 2.$ $\alpha \models \Diamond\Box\neg\langle C_{\mathrm{S},2}\rangle \wedge \Diamond\Box(rec_s = false \wedge rec_r = false \wedge queue \neq \varepsilon)$

PROOF: By Lemmas 5.10 and 5.11 since $C_{\mathrm{S},2}$ consists of external actions and Definition 7.5 of B_{DS} and Lemma 7.3 imply that for all $(s, u) \in B_{\mathrm{DS}}$, if $u \models (rec_s = false \wedge$

$rec_r = \mathit{false} \wedge queue \neq \varepsilon$) then $s \models (rec_s = \mathit{false} \wedge rec_r = \mathit{false} \wedge queue \neq \varepsilon)$.

⟨2⟩3. $\alpha \models \neg WF(C_{\mathrm{D},2}, rec_s = rec_r = \mathit{false})$

PROOF: By ⟨2⟩2, the definition of WF, the fact that $C_{\mathrm{S},2} = C_{\mathrm{D},2}$ and the fact that $enabled_{A_\mathrm{D}}(C_{\mathrm{D},2}) = (queue \neq \varepsilon)$.

⟨2⟩4. Q.E.D.

PROOF: By ⟨2⟩3, the assumption that $\alpha \models Q_\mathrm{D}$, and the definition of Q_D.

⟨1⟩4. CASE: $\alpha' \models \neg WF(C_{\mathrm{S},3})$

⟨2⟩1. Q.E.D.

PROOF: Similar to Case ⟨1⟩3.

⟨1⟩5. CASE: $\alpha' \models \neg WF(C_{\mathrm{S},4})$

⟨2⟩1. Q.E.D.

PROOF: Similar to Case ⟨1⟩3.

⟨1⟩6. Q.E.D.

PROOF: By ⟨1⟩1 and the exhaustive cases ⟨1⟩2–⟨1⟩5.

■

Finally, we can prove that D correctly implements S.

Theorem 7.18

$\mathrm{D} \sqsubseteq_\mathrm{L} \mathrm{S}$

Proof

Immediate from Lemmas 7.11, 7.17, and 5.9.

■

The total proof of correctness of D has been partitioned into three parts. First, some invariants were proved. Then, a relation was defined and proved to be an image-finite backward simulation from A_D to A_S. Note, that it is usually during the simulation proof that one realizes which invariants are needed. Thus, when performing the proof there is usually not this clear distinction between defining invariants and proving the simulation result, but for presentation purposes, we make the split.

The third and final part of the proof is the liveness proof which, in conjunction with the simulation proof, allows us to conclude correctness. In the proofs at lower levels of abstraction, the same partition into three parts is found.

The Generic Protocol G is defined and proved correct in the next chapter.

Chapter 8

The Generic Protocol G

We can now start to introduce a more distributed view of the system. Both low-level protocols H and C consist of several parallel components: a sender, a receiver, two channels connecting the sender and receiver, and, for C, a clock subsystem. The G level consists of three parallel processes: a sender/receiver process and two channels. This is depicted in Figure 8.1. The sender/receiver process of G can intuitively be viewed as "partly" distributed. It contains state variables which are intuitively manipulated by a sender part of the sender/receiver process and state variables which are intuitively manipulated by a receiver part. However, some state variables are manipulated by *both* the sender part and the receiver part of the sender/receiver process. These "centralized" variables describe aspects which will be implemented differently by H (using handshakes) and C (using timing assumptions). The "distributed" variables, on the other hand, will basically reoccur in both H and C, and will be manipulated similarly in G, H, and C.

Thus, we have developed G to be as distributed as possible according to H and C, and to contain an abstract handling of the crucial aspects of choosing good identifiers, where H and C use different methods. By looking a little bit forward at H and C, we can make the following more detailed introduction to G:

As mentioned in Chapter 1, solutions to the at-most-once message delivery problem work by tagging each message with a unique identifier and sending it repeatedly over the channel. The receiver will only accept messages which are marked with "good" identifiers.

Thus, the two protocols H and C both go through three major phases during normal operation.

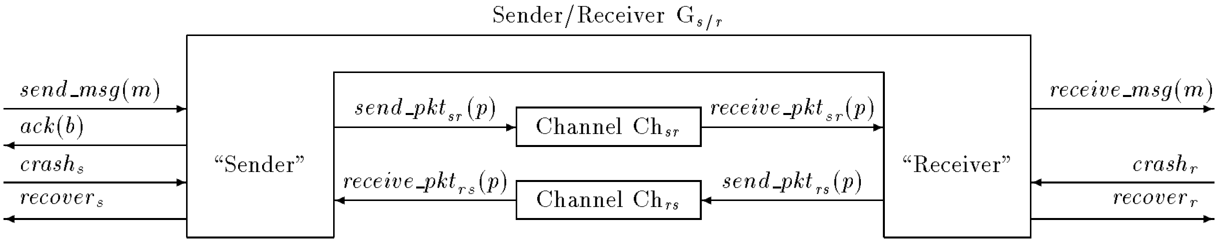


Figure 8.1

The Generic Protocol G.

Choosing a message identifier The sender picks an identifier *id* that is within the set of identifiers that the receiver is willing to accept. In C time bounds are used to choose a good identifier; in H an initial handshake between the sender and the receiver is used.

Sending the message and getting acknowledgement This phase is similar in both H and C. The sender (re)transmits the current message with the chosen *id*, until it receives an acknowledgement packet for that *id*.

Cleaning up Here again, C uses time bounds (in particular timeouts) whereas H uses a handshake to determine when some "old" information may be discarded.

Our Generic Protocol G is designed to capture these three phases in an abstract way that both H and C implement. The key abstractions incorporated into the protocol G are two "centralized" variables, $good_s$ and $good_r$. The variable $good_s$ represents the identifiers that the sender might shortly assign to messages, and $good_r$ represents the identifiers that the receiver is willing to accept. Four actions of G deal with "growing" and "shrinking" $good_s$ and $good_r$, respectively.

The preconditions of the grow and shrink actions are designed to preserve certain key invariants. We actually allow more freedom in these actions than is actually needed by H and C. This leaves open the possibility that other low-level protocols, other than H and C, can be proved to be correct implementations of G.

The rest of this chapter is organized as follows. Section 8.1 introduces the set of message identifiers. Section 8.2 then formally defines the channels in G. Then, in Section 8.3, we present the sender/receiver process, and in Section 8.4 we show how G is obtained from the subprocesses. Finally, in Section 8.5 we consider the proof that G correctly implements D.

8.1 Message Identifiers

In G and the lower level protocols we need a set of identifiers in order to label the messages communicated over the channels. In C the identifiers are timestamps ranging over the non-negative reals; in H the identifiers are just taken from some infinite set of elements. In G we use a set *ID* on which we place some constraints. When proving correct implementation for a lower-level protocol, *ID* is then instantiated with the set used at that lower level, and this set must satisfy the constraints on *ID*. Thus, G can be seen to be parameterized with *ID*. G correctly implements S for any proper value of *ID*; the low-level protocols correctly implement G for particular proper values of *ID*. The constraints on *ID* are:

1. *ID* is infinite.

2. `nil` $\notin$ *ID*. We need `nil` as a special value.

8.2 The Channels

As depicted in Figure 8.1, the G level contains two channels: a channel Ch_{sr} intuitively for sending packets[1] from the sender part to the receiver part of the sender/receiver process, and a channel Ch_{rs} in the other direction (for acknowledgements).

[1] Here and elsewhere, we use the term "packet" to denote objects sent over the channels; we reserve the term "message" for the "higher-level", user-meaningful messages that appear, e.g., in the specification.

Below we specify the Ch_{sr} channel as a live I/O automaton $(A_{\mathrm{Ch},sr}, L_{\mathrm{Ch},sr})$. The $\mathrm{Ch}_{rs} = (A_{\mathrm{Ch},rs}, L_{\mathrm{Ch},rs})$ channel is similar and can be obtained from the definition of Ch_{sr} by replacing the state variable sr by rs and actions $send\_pkt_{sr}(p)$ and $receive\_pkt_{sr}(p)$ by $send\_pkt_{rs}(p)$ and $receive\_pkt_{rs}(p)$.

8.2.1 States and Start States

Ch_{sr} has only one state variable which contains the packets (including duplicates) currently in the channel. We let Ch_{sr} be parameterized with a set P of possible packets.

Variable	Type	Initially	Description
sr	$\mathcal{B}(P)$	$\emptyset$	The packets (including duplicates) in the channel.

8.2.2 Actions

The channel only has two types of actions: $send\_pkt_{sr}(p)$, which represents the input of packet p from the environment, and $receive\_pkt_{sr}(p)$ which represents the output of packet p from the channel.

Input:
 $send\_pkt_{sr}(p)$, $p \in P$
Output:
 $receive\_pkt_{sr}(p)$, $p \in P$
Internal:
 none

8.2.3 Steps

The channel is not reliable. This means that it may remove or duplicate packets. We have chosen to model this unreliability at the time of a $send\_pkt_{sr}(p)$ step.

$send\_pkt_{sr}(p)$
 Effect:
  add a finite number of p to sr

$receive\_pkt_{sr}(p)$
 Precondition:
  $p \in sr$
 Effect:
  $sr := sr \setminus \{p\}$ (* remove one copy *)

In the specification, "a finite number" could mean 0. Note, that we could have modeled the unreliability of the channel by having internal *lose* and *duplicate* actions which could remove or duplicate packets at any time. However, such a channel can be shown to be equivalent to our channel, so by our substitutivity results, we will be able to substitute the channels for each other.

8.2.4 Liveness

The $receive\_pkt_{sr}(p)$ steps of $A_{\mathrm{Ch},sr}$ allow all received packets to be lost. With such a channel we cannot, of course, guarantee any liveness of the composed system, so we shall require that if we keep sending the same packet to the channel, then infinitely many will get through. Thus, if a packet is sent infinitely often, then it is also received infinitely often. Furthermore we impose the natural requirement that if a packet has succeeded in being put into the channel, then eventually it will be delivered.

Then the liveness condition $L_{\mathrm{Ch},sr}$ for the channel is induced by the following liveness formula:

$$\begin{aligned} Q_{\mathrm{Ch},sr} \quad &\triangleq \quad \forall p : \Box\Diamond\langle \mathit{send\_pkt}_{sr}(p)\rangle \Longrightarrow \Box\Diamond\langle \mathit{receive\_pkt}_{sr}(p)\rangle \wedge \\ & \qquad \forall p : \mathit{WF}(\mathit{receive\_pkt}_{sr}(p)) \end{aligned}$$

We do not prove formally that $Q_{\mathrm{Ch},sr}$ is an environment-free liveness formula for $A_{\mathrm{Ch},sr}$. However, we provide some intuition by informally describing an environment-free strategy (g, f) for Ch_{sr} (cf. Definitions 2.5 and 2.7): the g function of the strategy should on every input $\mathit{send\_pkt}_{sr}(p)$ add one copy of p to sr. This means that when we are playing the game against the environment, whenever a $\mathit{send\_pkt}_{sr}(p)$ input arrives, $\mathit{receive\_pkt}_{sr}(p)$ will stay enabled at least until it is executed.

The f function of the strategy, i.e., the function that determines the moves of the channel, should then work as follows: when the game commences after some finite execution, there are only finitely many packets in sr. The strategy can order these and use its first moves on outputting the packets. In the meantime $\mathit{send\_pkt}_{sr}(p)$ actions occur. When the strategy has finished outputting initial packets it should start matching each $\mathit{send\_pkt}_{sr}(p)$ action with a $\mathit{receive\_pkt}_{sr}(p)$ action. Since f has access to the history of the game so far, it should simply at its first move after having output initial packets perform $\mathit{receive\_pkt}_{sr}(p_1)$ if the first input action of the game was $\mathit{send\_pkt}_{sr}(p_1)$, and generally at its nth move perform $\mathit{receive\_pkt}_{sr}(p_n)$ if the nth input action of the game was $\mathit{send\_pkt}_{sr}(p_n)$. Even though the environment may provide several (but only a finite number of) input actions at each move and, thus, might be "faster" than the channel, at any point in time the channel only has finitely many "unmatched" inputs which it will eventually have matched. The point is that the environment can never have sent infinitely many copies of the same packet without the channel having output infinitely many copies of the same packet, and all packets put into the channel will eventually be output. If f has matched all inputs, it should simply return the empty move $\perp$ since in this case the channel is empty.

Note that, by Proposition 3.4, $Q_{\mathrm{Ch},sr}$ is stuttering-insensitive.

8.3 The Sender/Receiver Process

We specify the sender/receiver process as a live I/O automaton $\mathrm{G}_{s/r} = (A_{\mathrm{G},s/r}, L_{\mathrm{G},s/r})$.

8.3.1 States and Start States

As mentioned in the introduction to this chapter, $A_{\mathrm{G},s/r}$ intuitively consists of a sender part and a receiver part such that some state variables are only manipulated by the sender part, some state variables are only manipulated by the receiver part, and some state variables are manipulated by both parts. Thus, the state variables of $A_{\mathrm{G},s/r}$ are consequently grouped into the following three classes. (When we write "sender" below, we refer to the sender part of the sender/receiver process. Similarly for "receiver".)

Variable	**Type**	**Initially**	**Description**
$mode_s$	{`idle`, `needid`, `send`, `rec`}	`idle`	The mode of the sender. Mode `idle` indicates that the sender is not in the process of sending a packet over the channel, `needid` indicates that the sender is ready to choose an identifier for the current message, and `send` indicates that the sender is sending (repeatedly) the current packet (consisting of current message with identifier) over the channel. Mode `rec` denotes that the sender is in recovery phase.
buf_s	Msg^*	ε	The list of messages at the sender side.
$used_s$	ID^*	ε	A list containing all identifiers assigned to messages in the past. These identifiers will never be used again. The list induces a partial order on identifiers (see below).
$current\text{-}msg_s$	$Msg \cup \{\texttt{nil}\}$	`nil`	When $mode_s \in \{\texttt{needid}, \texttt{send}\}$, this variable contains the "current" message, i.e., the message about to be or being sent. In the other modes $current\text{-}msg_s$ is not used and is set to `nil`.
$last_s$	$ID \cup \{\texttt{nil}\}$	Any value	When $mode_s = \texttt{send}$ this variable contains the identifier chosen for the current message. In all other modes its value is not used. Due to requirements in low-level protocols (where $last_s$ could, e.g., be a timestamp), $last_s$ is allowed to assume arbitrary values when it is not used.
$current\text{-}ack_s$	Bool	*false*	Acknowledgement from the receiver.

$mode_r$	{`idle`, `rcvd`, `ack`, `rec`}	`idle`	The mode of the receiver. Mode `idle` indicates that the receiver has delivered all received messages to the user, `rcvd` indicates that messages have been accepted but not yet delivered to the user, `ack` indicates that the receiver is sending positive acknowledgements for the last message accepted to the sender. Mode `rec` denotes that the receiver is in recovery phase.
buf_r	Msg^*	ε	The list of messages accepted by the receiver but not yet delivered.
$last_r$	$ID \cup \{$`nil`$\}$	`nil`	Contains the identifier of the last message accepted. When its value is not used, it is assigned the special value `nil`.
$issued_r$	$\mathcal{P}(ID)$	Any superset of $good_r$ such that $\lvert ID \setminus issued_r \rvert = \infty$	Includes everything that was ever acceptable by the receiver, i.e., in $good_r$. Thus, $issued_r$ is used to guarantee that "old" identifiers do not show up in $good_r$ again, which could otherwise lead to duplicate delivery.
$nack\text{-}buf_r$	ID^*	ε	A list of identifiers for which a negative acknowledgement will be issued.

$good_s$	$\mathcal{P}(ID)$	Any set	When $mode_s =$ `needid` this set contains all the identifiers that the sender might choose for the current message. In all other modes its value is not used.
$good_r$	$\mathcal{P}(ID)$	Any set	At any time this set contains the identifiers the receiver will accept from the channel.
$current\text{-}ok$	Bool	*false*	If $current\text{-}ok = true$ the identifier chosen for the current message is considered good by the receiver, but the current message has not been accepted by the receiver yet.

8.3.2 Partial Order of Identifiers

In the G protocol we need an ordering of all the identifiers used as ids on messages sent by the sender. As we shall see below, an identifier id is chosen in a $choose\_id(id)$ step, so if a $choose\_id(id)$ step has occurred before a $choose\_id(id')$ step, we will require that id is less than id' in this ordering. Since we collect—as we shall see—all the ids used by the sender in $used_s$, we use the following partial order derived from the state of G:

If $used_s$ contains distinct elements and id precedes id' in $used_s$, then $id <_u id'$

In arbitrary states of G the same identifier might occur several times in $used_s$; however, below we shall prove an invariant (Invariant 8.2 Part 2 on Page 125), which states that the elements of $used_s$ are all distinct, which then implies that all identifiers ever used by the sender during execution are related by $<_u$. Since identifiers of ID can be tested for equivalence ($=$), the definition of $<_u$ trivially extends to $\leq_u$.

8.3.3 Actions

Input:
 $send\_msg(m)$, $m \in Msg$
 $receive\_pkt_{sr}(m, id)$, $m \in Msg$, $id \in ID$
 $receive\_pkt_{rs}(id, b)$, $id \in ID$, $b \in \mathsf{Bool}$
 $crash_s$
 $crash_r$
Output:
 $receive\_msg(m)$, $m \in Msg$
 $ack(b)$, $b \in \mathsf{Bool}$
 $send\_pkt_{sr}(m, id)$, $m \in Msg$, $id \in ID$
 $send\_pkt_{rs}(id, b)$, $id \in ID$, $b \in \mathsf{Bool}$
 $recover_s$
 $recover_r$
Internal:
 $prepare$
 $choose\_id(id)$, $id \in ID$
 $shrink\_good_s(ids)$, $ids \subseteq ID$
 $shrink\_good_r(ids)$, $ids \subseteq ID$
 $grow\_good_s(ids)$, $ids \subseteq ID$
 $grow\_good_r(ids)$, $ids \subseteq ID$
 $cleanup_r$

8.3.4 Steps

Before we formally define $steps(A_{G,s/r})$ we provide some intuition. During normal operation the sender goes through the cycle `idle`–`needid`–`send`–`idle` of modes. When the sender is in mode `idle` and buf_s is non-empty, a *prepare* step moves to mode `needid` and makes the message at the head of buf_s the current message. Now "good" identifiers must be put into $good_s$. Exactly how this is done will be discussed below. An identifier id for the current message is chosen from $good_s$ in a $choose\_id(id)$ step. In such a step the sender enters `send` mode in which it repeatedly sends the current message m with associated current identifier id in $send\_pkt_{sr}(m, id)$ steps. The sender will stay in this mode until it receives a positive ($b = true$) or negative ($b = false$) acknowledgement $receive\_pkt_{rs}(id, b)$ for the current identifier. In this case the sender moves to mode `idle` again from where acknowledgements $ack(b)$ can be issued to the user (but only of buf_s is empty since otherwise the sender is not acknowledging the last message sent, as required).

If the receiver receives a packet (m, id) in a $receive\_pkt_{sr}(m, id)$ step, it checks to see whether id is in $good_r$. If this is the case it accepts[2] the message m, adds it to the end of buf_r and enters mode `rcvd` (if it was not there already). Mode `rcvd` indicates that the receiver has messages in buf_r and is in the process of delivering these messages to the user. Once the last message in buf_r has been delivered in a $receive\_msg(m)$ step, the receiver enters `ack` mode in which it will issue positive acknowledgements in $send\_pkt_{rs}(id, true)$ steps for the identifier id of the last message accepted from the sender (and thus the last message delivered to the user). These positive acknowledgements will be issued repeatedly to overcome the unreliability of the channel.

[2] We say that a packet (or the associated message) is "successfully received" or "accepted" when the associated identifier is in $good_r$ at the time of receipt.

The above discussion has focused on the normal modes of operation of the sender and receiver, where no crashes have occurred. After the formal definition of $steps(A_{G,s/r})$, we explain what can happen when sender or receiver crashes occur.

We now look at the manipulation of the *good* sets. When a *prepare* step is performed, the $good_s$ set is emptied. The sender is now in `needid` mode, waiting to perform a $choose\_id(id)$ step. Since *id* must be taken from $good_s$, this set must be "grown" with identifiers. Two types of steps can change $good_s$: $shrink\_good_s(ids)$ removes identifiers from $good_s$ and $grow\_good_s(ids)$ adds identifiers to $good_s$. When the receiver has not been in recovery phase "recently", i.e., after the *prepare* step was performed, the sender and receiver should be in agreement about which identifiers are considered good. This situation is indicated by the special flag *current-ok* being *true*. In this situation $grow\_good_s(ids)$ can only add elements from $good_r$ to $good_s$, and the $shrink\_good_r(ids)$ steps, which can remove elements from $good_r$, must not remove elements which are already in $good_s$. In this way we preserve the key invariant that if $current\text{-}ok = true$, then $good_s \subseteq good_r$, and, thus, the current packet is guaranteed to be accepted by the receiver (unless new crashes occur). A detail is that identifiers put into $good_s$ might immediately be "shrunk" away by a $shrink\_good_s(ids)$ step that empties $good_s$. (If we look forward at C, only the value of the local sender clock is considered a good identifier. Thus, whenever the clock ticks, this corresponds, in G, to the old clock value being removed from $good_s$, and the new value being added to $good_s$.) When we deal with liveness below, we show how to guarantee that the sender will not grow and shrink $good_s$ forever but will eventually choose an identifier in a $choose\_id(id)$ step.

If crashes occur, the low-level implementations H and C have no way of keeping $good_s$ a subset of $good_r$. This must at the G level be reflected in the *grow* and *shrink* steps. We have designed these steps such that they preserve certain key invariants presented below. The steps actually allow more freedom than is needed by the implementations H and C, but in this way we have the possibility that *other* low-level implementations implement G. If, for instance, $current\text{-}ok = false$, it turns out to be necessary to allow $shrink\_good_r$ to remove elements from $good_r$ which are already in $good_s$. If, furthermore, $mode_r = \mathtt{rec}$, $good_s$ can be grown fairly arbitrarily. It is in this situation possible to add elements to $good_s$ which have never been issued by the receiver. This may give rise to a situation where the current identifier is not in $good_r$ when the current packet is sent, but is added to $good_r$ during transmission over the channel. (For this reason we shall, in the proofs below, introduce a derived variable *good-ids* containing identifiers from $good_r$ and identifiers not issued yet. Packets with identifiers in *good-ids* have a chance of being accepted by the receiver.)

Other preconditions on the *grow* and *shrink* steps deal with guaranteeing that the sender and receiver do not reuse identifiers in their *good* sets. In particuler, the set $issued_r$, which "survives" a crash (and thus has to be implemented in *stable* storage in the implementations), contains all identifiers that were ever in $good_r$. No identifiers in $issued_r$ can ever be put in $good_r$. In this way it is guaranteed that the receiver will never—not even in the case of crashes—accept the same packet twice. Similarly, the sender will never choose an identifier which is in $used_s$.

We now define $steps(A_{G,s/r})$. To increase readability we keep the definition of the steps of the sender in the left column and the definition of the steps of the receiver in the right column. Furthermore, we align the definition of the *send-pkt* steps with the definition of the corresponding *receiver-pkt* steps.

$send\_msg(m)$
  Effect:
    if $mode_s \neq \texttt{rec}$ then
      $buf_s := buf_s \,\hat{}\, m$

$prepare$
  Precondition:
    $mode_s = \texttt{idle} \wedge buf_s \neq \varepsilon$
  Effect:
    $mode_s := \texttt{needid}$
    $good_s := \emptyset$
    $current\text{-}msg_s := head(buf_s)$
    $buf_s := tail(buf_s)$
    if $mode_r \neq \texttt{rec}$ then
      $current\text{-}ok := true$

$choose\_id(id)$
  Precondition:
    $mode_s = \texttt{needid} \wedge id \in good_s$
  Effect:
    $mode_s := \texttt{send}$
    $last_s := id$
    $used_s := used_s \,\hat{}\, id$

$send\_pkt_{sr}(m, id)$
  Precondition:
    $mode_s = \texttt{send} \wedge last_s = id \wedge$
    $current\text{-}msg_s = m$
  Effect:
    none

$receive\_pkt_{sr}(m, id)$
  Effect:
    if $mode_r \neq \texttt{rec}$ then
      if $id \in good_r$ then
        $mode_r := \texttt{rcvd}$
        $buf_r := buf_r \,\hat{}\, m$
        $last_r := id$
        $good_r := good_r \setminus \{id' \mid id' \leq_u id\}$
        if $id = last_s \wedge mode_s = \texttt{send}$ then
          $current\text{-}ok := false$
      else if $id \neq last_r$ then
        if $mode_s = \texttt{send} \wedge id = id_s$ then
          $nack\text{-}buf_r := nack\text{-}buf_r \,\hat{}\, id$
        else
          optionally $nack\text{-}buf_r := nack\text{-}buf_r \,\hat{}\, id$
      else if $mode_r = \texttt{idle}$ then
        $mode_r := \texttt{ack}$

$receive\_msg(m)$
  Precondition:
    $mode_r = \texttt{rcvd} \wedge buf_r \neq \varepsilon \wedge head(buf_r) = m$
  Effect:
    $buf_r := tail(buf_r)$
    if $buf_r = \varepsilon$ then
      $mode_r := \texttt{ack}$

$receive\_pkt_{rs}(id, b)$
Effect:
if $mode_s = \mathtt{send} \wedge last_s = id$ then
$mode_s := \mathtt{idle}$
$current\text{-}ack_s := b$
$last_s :=$ arbitrary value
$current\text{-}msg_s := \mathtt{nil}$

$ack(b)$
Precondition:
$mode_s = \mathtt{idle} \wedge buf_s = \varepsilon \wedge$
$current\text{-}ack_s = b$
Effect:
none

$crash_s$
Effect:
$mode_s = \mathtt{rec}$
$current\text{-}ok := false$

$recover_s$
Precondition:
$mode_s = \mathtt{rec}$
Effect:
$mode_s := \mathtt{idle}$
$last_s :=$ arbitrary value
$buf_s := \varepsilon$
$current\text{-}msg_s := \mathtt{nil}$
$current\text{-}ack_s := false$

$grow\_good_s(ids)$
Precondition:
$mode_s \neq \mathtt{needid} \vee$
$((mode_r \neq \mathtt{rec} \Longrightarrow ids \subseteq issued_r) \wedge$
$(current\text{-}ok = true \Longrightarrow ids \subseteq good_r) \wedge$
$(ids \cap used_s = \emptyset))$
Effect:
$good_s := good_s \cup ids$

$shrink\_good_s(ids)$
Precondition:
none
Effect:
$good_s := good_s \setminus ids$

$send\_pkt_{rs}(id, true)$
Precondition:
$mode_r = \mathtt{ack} \wedge last_r = id$
Effect:
optionally $mode_r := \mathtt{idle}$

$send\_pkt_{rs}(id, false)$
Precondition:
$mode_r \neq \mathtt{rec} \wedge nack\text{-}buf_r \neq \varepsilon \wedge$
$head(nack\text{-}buf_r) = id$
Effect:
$nack\text{-}buf_r := tail(nack\text{-}buf_r)$

$crash_r$
Effect:
$mode_r = \mathtt{rec}$
$current\text{-}ok := false$

$recover_r$
Precondition:
$mode_r = \mathtt{rec}$
Effect:
$mode_r := \mathtt{idle}$
$last_r := \mathtt{nil}$
$buf_r := \varepsilon$
$nack\text{-}buf_r := \varepsilon$
$issued_r :=$ any superset of
$issued_r \cup used_s \cup good_s$
such that afterwards
$|ID \setminus issued_r| = \infty$

$grow\_good_r(ids)$
Precondition:
$ids \cap issued_r = \emptyset \wedge$
$|ID \setminus (ids \cup issued_r)| = \infty$
Effect:
$good_r := good_r \cup ids$
$issued_r := issued_r \cup ids$

$shrink\_good_r(ids)$
Precondition:
$current\text{-}ok = false \vee$
$((mode_s = \mathtt{needid} \Longrightarrow ids \cap good_s = \emptyset) \wedge$
$(mode_s = \mathtt{send} \Longrightarrow last_s \notin ids))$
Effect:
$good_r := good_r \setminus ids$

$cleanup_r$
Precondition:
$mode_r \in \{\mathtt{idle}, \mathtt{ack}\} \wedge$
$(mode_s = \mathtt{send} \Longrightarrow last_s \neq last_r)$
Effect:
$mode_r := \mathtt{idle}$
$last_r := \mathtt{nil}$

Note that most locally-controlled steps of the sender and receiver are conditioned by $mode_s$ and $mode_r$, respectively, not being `rec`. Also, inputs (except $crash_s$ and $crash_r$) do not lead to state changes when the side at which they occur is crashed. Thus, G is "dead" when it is crashed. Furthermore, crashes and subsequent recoveries have the effect of resetting all state variables (except $issued_r$ and $used_s$) at the side at which they occur. For instance, even if the sender is about to issue a positive acknowledgement to the user when a sender crash occurs, the sender has forgotten about this when it recovers. These choices about the way G behaves with respect to crashes are motivated by the low-level protocols H and C.

We now discuss certain special situations that can arise mainly due to crashes or recoveries. Assume that the sender is in `send` mode with (m_1, id_1) as the current packet. If a $crash_s$ occurs, the sender forgets, among other things, everything about (m_1, id_1). However, before it crashed, the sender might have succeeded in placing (m_1, id_1) in the channel. Since we do not assume any time bounds on channel delays, (m_1, id_1) might travel very slowly on the channel. In the meantime the sender recovers, receives a new message m_2 in a $send\_msg(m_2)$ step, assigns the identifier id_2 to m_2, and starts sending (m_2, id_2) to the channel. Now both (m_1, id_1) and (m_2, id_2) are traveling on the channel, and both id_1 and id_2 might be in $good_r$. (The receiver has no way of knowing that the sender has been crashed.) In general, if crashes have occurred, several packets $(m_1, id_1), \ldots, (m_k, id_k)$ with identifiers in $good_r$ might be traveling on the channel. This gives rise to a *race condition* between the packets. Assume (m_i, id_i) is the first packet that reaches the receiver and gets accepted. Then the receiver is not allowed subsequently to accept *any* of the packets $(m_1, id_1), \ldots, (m_i, id_i)$ since then either the receiver would accept the same message twice or it would reorder messages (since $m_1, \ldots, m_{i-1}$ were sent *before* m_i). The messages $m_1, \ldots, m_{i-1}$ are thus effectively lost, but since they were in the system during crashes, this is allowed by the Delayed-Decision Specification D (and consequently by the specification S). This explains the manipulation of $good_r$ in the definition of the $receive\_pkt_{sr}(m, id)$ steps. If the sender crashes in `needid` mode, the same kind of race condition does not arise since the current packet has not been placed in the channel yet. However, messages get lost but, again, this is allowed by D.

If the receiver receives a packet (m, id) and id is not in $good_r$ it will not accept the packet. Now, two situations must be considered (which correspond to the two "else-if" cases in the definition of $receive\_pkt_{sr}(m, id)$ above).

1. If $id \neq last_r$, we are not just receiving another copy of the last packet accepted.

   - if $mode_s$ = `send` and $id = last_s$, we are, due to crashes, in a situation where the sender is in `send` mode with a "bad" identifier. The receiver must inform the sender about this situation since otherwise the sender would be stuck forever. Thus, the receiver adds id to $nack\text{-}buf_r$ which will lead to a $send\_pkt_{rs}(id, false)$ step. Note, that since only one $send\_pkt_{rs}(id, false)$ will be performed, there is no guarantee that the packet will actually be put into the channel (which is unreliable). However, the sender continues to send (m, id), so packets will continue to get through (due to channel liveness) to the receiver. Every time this happens, the receiver will add id to $nack\text{-}buf_r$, so $(id, false)$ will continue to be issued. By channel liveness in the other direction the sender will eventually receive $(id, false)$ and thereby be dislodged.
   - if $mode_s \neq$ `send` or $id \neq last_s$, the received packet (m, id) is not the current packet of the sender but instead some old packet from the channel. The low-level protocols we consider cannot always identify this situation—mainly because the receiver in a

distributed implementation does not have access to $mode_s$ and $last_s$. The C protocol can in some situations make some safe guesses, but generally a low-level protocol has to assume the worst case and thus add id to $nack\text{-}buf_r$. The G protocol leaves this possibility open.

2. If $id = last_r$, we are receiving a new copy of the last packet accepted. In this situation $mode_r$ could be `idle`, in which case it should be changed to `ack`. The situation is explained as follows.

   Due to requirements in the low-level implementations, a $send\_pkt_{rs}(id, true)$ step must have the possibility of changing $mode_r$ to `idle`, which disables further $send\_pkt_{rs}(id, true)$ steps. Thus, due to the unreliability of the channels, we are not sure that $(id, true)$ actually arrives to inform the sender that the current packet was successfully received. But the sender will then continue to send (m, id) packets, and the (inevitable) receipt of some of these by the receiver will lead to mode change to `ack`, which, in turn, leads to $send\_pkt_{rs}(id, true)$ steps. As above, channel liveness ensures that a $receive\_pkt_{rs}(id, true)$ step will eventually occur as required.

Some of this discussion has dealt with liveness. We now turn to the formal definition of the liveness condition for $\mathrm{G}_{s/r}$.

8.3.5 Liveness

Let

$$
\begin{aligned}
C_{\mathrm{G},s/r1} &\triangleq \{prepare, ack(true), ack(false), recover_s\} \cup \\
&\ \{send\_pkt_{sr}(m, id) \mid m \in Msg \wedge id \in ID\} \\
C_{\mathrm{G},s/r2} &\triangleq \{choose\_id(id) \mid id \in ID\} \\
C_{\mathrm{G},s/r3} &\triangleq \{recover_r\} \cup \\
&\ \{receive\_msg(m) \mid m \in Msg\} \cup \\
&\ \{send\_pkt_{rs}(id, true) \mid id \in ID\} \\
C_{\mathrm{G},s/r4} &\triangleq \{send\_pkt_{rs}(id, false) \mid id \in ID\}
\end{aligned}
$$

The liveness condition $L_{\mathrm{G},s/r}$ for $A_{\mathrm{G},s/r}$ is now induced by the following temporal formula.

$$
\begin{aligned}
Q_{\mathrm{G},s/r} \triangleq\ & WF(C_{\mathrm{G},s/r1}) \wedge \\
& \Box(\Box(mode_s = \mathtt{needid} \wedge mode_r \neq \mathtt{rec}) \Longrightarrow \Diamond\langle C_{\mathrm{G},s/r2}\rangle) \wedge \\
& WF(C_{\mathrm{G},s/r3}) \wedge \\
& WF(C_{\mathrm{G},s/r4})
\end{aligned}
$$

The first, third, and fourth conjunct express normal weak fairness to some locally-controlled actions of the sender and receiver, respectively.

The second conjunct looks more complicated but simply states that it is always the case that if the sender stays in mode `needid` and the receiver does not crash, then eventually a $choose\_id(id)$ step occurs. Thus, infinite growing and shrinking of the *good* sets are avoided. Note, that this kind of liveness condition is more high-level than, e.g., weak fairness, but it exactly captures the intuitive requirement to the execution of the system, and the general model of live I/O automata allows such general liveness requirements.

As for the liveness formula for the channel Ch_{sr} above, we do not formally prove that $Q_{\mathrm{G},s/r}$ is an environment-free liveness formula for $A_{\mathrm{G},s/r}$ but instead provide some intuition as to how an environment-free strategy (g, f) could be defined: on inputs, the g function can choose arbitrarily between nondeterministic choices. The f function should deal with the four conjuncts of $Q_{\mathrm{G},s/r}$ in a round-robin fashion: if it dealt with the first conjunct last time, it should deal with the second conjunct now, and so on. If it is time to deal with one of the weak-fairness formulas, f simply performs some step from the appropriate set if possible. The second conjunct needs more attention. Here f should do the following if $mode_s = \mathtt{needid}$ and $mode_r \neq \mathtt{rec}$, and do nothing otherwise:

1. If $good_s \neq \emptyset$, then perform a $choose\_id(id)$ step.
2. Else, if $good_r \neq \emptyset$, perform a $grow\_good_s(ids)$ step (with ids nonempty). Such a step is always possible when $good_r \neq \emptyset$.
3. Else, perform a $grow\_good_r(ids)$ step with ids nonempty. Such a step is always possible since it is true that there are always infinitely many unused identifiers left.

If Part 3 was performed, then Part 2 will be performed next time the second conjunct of $Q_{\mathrm{G},s/r}$ is dealt with. If Part 2 was performed, then Part 1 will be chosen next time. This is under the assumption that the sender stays in mode `needid` and the receiver does not crash in the meantime, but if this is not satisfied, then the second conjunct does not restrict the execution at all.

Another thing to note is that, by Lemma 4.8 and Proposition 3.4, $Q_{\mathrm{G},s/r}$ is stuttering-insensitive.

8.4 The Specification of G

As depicted in Figure 8.1, G consists of the sender/receiver process and the two channels. So, first define $\mathrm{G}' = (A'_{\mathrm{G}}, L'_{\mathrm{G}})$ to be the following live I/O automaton

$$\mathrm{G}' \quad \triangleq \quad \mathrm{G}_{s/r} \| \mathrm{Ch}_{sr} \| \mathrm{Ch}_{rs}$$

where the set P of possible packets of the channels is instantiated with the packets that $\mathrm{G}_{s/r}$ can send and receive, i.e., packets of the form (m, id) and (id, b). Thus, G' is the parallel composition of the sender/receiver process and the channels. Since $Q_{\mathrm{G},s/r}$, $Q_{\mathrm{Ch},sr}$, and $Q_{\mathrm{Ch},rs}$ are all stuttering-insensitive, Proposition 4.4 implies that L'_{G} is induced by

$$Q_{\mathrm{G}} \quad \triangleq \quad Q_{\mathrm{G},s/r} \wedge Q_{\mathrm{Ch},sr} \wedge Q_{\mathrm{Ch},rs}$$

By Definition 2.2 the channel actions $send\_pkt_{sr}(m, id)$, $receive\_pkt_{sr}(m, id)$, $send\_pkt_{rs}(id, b)$, and $receive\_pkt_{rs}(id, b)$ are output actions of G'. Thus, to get $\mathrm{G} = (A_{\mathrm{G}}, L_{\mathrm{G}})$ we hide these actions. Let

$$\begin{aligned} \mathcal{A}_{\mathrm{G}} \quad \triangleq \quad & \{send\_pkt_{sr}(m, id) \mid m \in Msg \wedge id \in ID\} \cup \\ & \{receive\_pkt_{sr}(m, id) \mid m \in Msg \wedge id \in ID\} \cup \\ & \{send\_pkt_{rs}(id, b) \mid id \in ID \wedge b \in \mathsf{Bool}\} \cup \\ & \{receive\_pkt_{rs}(id, b) \mid id \in ID \wedge b \in \mathsf{Bool}\} \end{aligned}$$

Then, define

$$\mathrm{G} \quad \triangleq \quad \mathrm{G}' \setminus \mathcal{A}_{\mathrm{G}}$$

By Proposition 4.5, L_{G} is induced by Q_{G}.

We can now turn attention to proving that G correctly implements D.

8.5 Correctness of G

In this section we consider the proof that $\mathrm{G} = (A_{\mathrm{G}}, L_{\mathrm{G}})$ correctly implements $\mathrm{D} = (A_{\mathrm{D}}, L_{\mathrm{D}})$. This will be done in terms of a refinement mapping from A_{G} to A_{D} and a subsequent liveness proof. We perform the refinement proof in all detail, but only sketch the liveness proof. We refer to the formal liveness proof at the H level for a similar—but formal—liveness proof.

First, we state some invariants of A_{G}.

8.5.1 Invariants

As mentioned in Chapter 7, during the process of performing a simulation proof, it usually becomes clear that certain invariants are needed: some situation in the proof is impossible to solve but it turns out that the state in which the situation occurs is not reachable. Thus, an invariant that avoids these "bad" states is found. In this section we present the invariants we need in the refinement mapping proof from A_{G} to A_{D}. The proofs of the invariants are deferred to Appendix C, where we furthermore consider the general way to prove invariants of safe (timed) I/O automata.

In the invariants we use a derived variable *good-ids* defined as follows: in any state s of A_{G}, define

$$s.\mathit{good\text{-}ids} \;\triangleq\; s.\mathit{good}_r \cup \overline{s.\mathit{issued}_r}$$

where $\overline{s.\mathit{issued}_r}$ is the complement of $s.\mathit{issued}_r$ with respect to ID. A message assigned an *id* in $s.\mathit{good\text{-}ids}$ might still be received successfully, i.e, accepted by the receiver.

The first invariant has two parts which state simple properties of the state when the sender is in `send` mode. (Recall from Appendix A that $\mathit{last}_s \in \mathit{used}_s$ is shorthand notation for $\mathit{last}_s \in \mathit{elems}(\mathit{used}_s)$. Similar notation will be used below.)

Invariant 8.1

1. If $\mathit{mode}_s = \texttt{send}$ then $\mathit{last}_s \in \mathit{used}_s$
2. If $\mathit{mode}_s = \texttt{send}$ then $\mathit{last}_s \neq \texttt{nil}$

■

When the sender is in `needid` mode, it can never choose among identifiers that have been used before (since such identifiers cannot be put into good_s again). As a consequence used_s contains distinct elements.

Invariant 8.2

1. If $\mathit{mode}_s = \texttt{needid}$ then $\mathit{used}_s \cap \mathit{good}_s = \emptyset$

2. All elements of $used_s$ are distinct

■

As expected a receiver mode of `rcvd` indicates that there are some messages in the receiver buffer which have not yet been delivered to the user.

Invariant 8.3

1. If $mode_r = \texttt{rcvd}$ then $buf_r \neq \varepsilon$

■

The following invariant is a key invariant. It states relationships between and properties of the different sets of identifiers in A_G.

In this invariant and other invariants below, we use the following definition: define in any state s of A_G $ids(sr)$ to be the set of id components of the packets in the sr channel. Formally, we have

$$ids(sr) \triangleq \{id \mid m \in Msg \wedge (m, id) \in sr\}$$

Similarly,

$$ids(rs) \triangleq \{id \mid b \in \mathsf{Bool} \wedge (id, b) \in rs\}$$

Invariant 8.4

1. $issued_r \supseteq good_s$ if $mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec}$
2. $issued_r \supseteq good_r$
3. $issued_r \supseteq used_s$ if $mode_r \neq \texttt{rec}$
4. $used_s \supseteq ids(sr) \cup (\text{if } mode_s = \texttt{send} \text{ then } \{last_s\} \text{ else } \emptyset)$
5. $used_s \supseteq nack\text{-}buf_r$
6. $used_s \supseteq ids(rs)$
7. $last_r \notin good\text{-}ids$
8. If $last_r \neq \texttt{nil}$ then $last_r \in used_s$

■

The following invariant states the fact that for any two packets in sr (possibly including the current packet), if the packets have the same identifier, then the packets are equal (and thus represent two copies of the same packet).

Invariant 8.5

1. Let $pkts = sr \cup (\text{if } mode_s = \texttt{send} \text{ then } \{(current\text{-}msg_s, last_s)\} \text{ else } \emptyset)$, and let $(m, id) \in pkts$ and $(m', id') \in pkts$. Then If $id = id'$ then $m = m'$

■

The next invariant states properties of reachable states where $\textit{current-ok} = \textit{true}$. Recall that *current-ok* intuitively is a flag which is *true* whenever the sender is in the process of sending the next message (packet), the receiver has not been in recovery phase since the last *prepare* action, and the current packet has not been received yet. Thus, $\textit{current-ok} = \textit{true}$ indicates that the sender and receiver are synchronized and in agreement about which identifiers to use.

Invariant 8.6

1. If $\textit{current-ok} = \textit{true}$ then $\textit{mode}_s \in \{\texttt{needid}, \texttt{send}\}$
2. If $\textit{current-ok} = \textit{true}$ then $\textit{mode}_r \neq \texttt{rec}$
3. If $\textit{current-ok} = \textit{true} \wedge \textit{mode}_s = \texttt{send}$ then $\textit{last}_s \neq \textit{last}_r$
4. If $\textit{current-ok} = \textit{true} \wedge \textit{mode}_s = \texttt{send}$ then $(\textit{last}_s, b) \notin \textit{rs}$
5. If $\textit{current-ok} = \textit{true} \wedge \textit{mode}_s = \texttt{needid}$ then $\textit{good}_s \subseteq \textit{good}_r$
6. If $\textit{current-ok} = \textit{true} \wedge \textit{mode}_s = \texttt{send}$ then $\textit{last}_s \in \textit{good}_r$
7. If $\textit{current-ok} = \textit{true} \wedge \textit{mode}_s = \texttt{send}$ then $\textit{last}_s \notin \textit{nack-buf}_r$

■

In certain situations *current-ok* is guaranteed to be *false*. For instance, if the sender is in $\texttt{send}$ mode and the current packet has been accepted by the receiver (indicated by either $\textit{last}_s = \textit{last}_r$ or the fact that an acknowledgement for $\textit{last}_s$ is in *rs*).

Invariant 8.7

1. If $\textit{mode}_s = \texttt{send} \wedge \textit{last}_s = \textit{last}_r$ then $\textit{current-ok} = \textit{false}$
2. If $\textit{mode}_s = \texttt{send} \wedge (\textit{last}_s, b) \in \textit{rs}$ then $\textit{current-ok} = \textit{false}$

■

We now state properties of the identifiers in *sr*. Part 1 states that each identifier in *sr* has been chosen before (or is equal to) the current identifier when $\textit{mode}_s = \texttt{send}$. This is expressed using the ordering $<_u$ induced by $\textit{used}_s$. Parts 2–4 state that if either (2) the current packet has been accepted by the receiver, (3) the receiver has sent positive acknowledgement for the current packet to *rs*, or (4) the sender has received the positive acknowledgement, then none of the identifiers in *sr* (possibly including the current identifier $\textit{last}_s$) can never become "good", i.e., can never reappear in $\textit{good}_r$. (These invariants among other things guarantee that A_{G} can never reorder messages or accept the same packet twice.)

Invariant 8.8

1. If $\textit{mode}_s = \texttt{send} \wedge \textit{id} \in \textit{ids}(\textit{sr})$ then $\textit{last}_s \geq_u \textit{id}$
2. If $\textit{mode}_s = \texttt{send} \wedge \textit{last}_s = \textit{last}_r$ then $(\{\textit{last}_s\} \cup \textit{ids}(\textit{sr})) \cap \textit{good-ids} = \emptyset$
3. If $\textit{mode}_s = \texttt{send} \wedge (\textit{last}_s, \textit{true}) \in \textit{rs}$ then $(\{\textit{last}_s\} \cup \textit{ids}(\textit{sr})) \cap \textit{good-ids} = \emptyset$

4. If $mode_s = \texttt{idle} \wedge current\text{-}ack_s = true$ then $ids(sr) \cap good\text{-}ids = \emptyset$

■

In certain situations buf_r is guaranteed to be empty. Part 1 of the following invariant states that if $mode_r = \texttt{idle}$ then buf_r is empty. This situation occurs if the receiver has just sent acknowledgement after having delivered the last message to the user, or if the receiver has just recovered. Parts 2–4 deal with the situation where the current message is being acknowledged over rs. Either (2) the receiver is sending positive acknowledgements for the last message received (and passed on to the user), (3) the receiver has succeeded in placing the positive acknowledgement in rs, or (4) the sender has already received the positive acknowledgement.

Invariant 8.9

1. If $mode_r = \texttt{idle}$ then $buf_r = \varepsilon$
2. If $mode_r = \texttt{ack}$ then $buf_r = \varepsilon$
3. If $mode_s = \texttt{send} \wedge (last_s, true) \in rs$ then $buf_r = \varepsilon$
4. If $mode_s = \texttt{idle} \wedge current\text{-}ack_s = true$ then $buf_r = \varepsilon$.

■

The following invariant states that identifiers for which the receiver will or has sent negative acknowledgements can never (again) be considered "good" by the receiver.

Invariant 8.10

1. $nack\text{-}buf_r \cap good\text{-}ids = \emptyset$
2. $ids(rs) \cap good\text{-}ids = \emptyset$

■

Furthermore, the receiver can never issue negative acknowledgements for the current identifier if it has accepted the current packet (unless new crashes have occurred).

Invariant 8.11

1. If $mode_s = \texttt{send} \wedge last_s \in nack\text{-}buf_r$ then $last_s \neq last_r$.
2. If $mode_s = \texttt{send} \wedge (last_s, false) \in rs$ then $last_s \neq last_r$.

■

Our final invariant states that there are always "enough" (read: infinitely many) identifiers left that have not been issued. This is an important invariant since it ensures that a message to be sent can always be associated with an identifier. The invariant will not be used in the safety proof since not being able to choose an identifier does not violate any safety requirement. Instead the invariant is essential for the system to guarantee any liveness requirements.

Invariant 8.12

1. $|ID \setminus issued_r| = \infty$

■

The conjunction of all invariants above (which is itself an invariant) will be referred to by I_{G}.

8.5.2 Safety

In this section we show the existence of a refinement mapping from A_{G} to A_{D}. However, first we need some preliminary definitions.

Let s be any state of A_{G} which satisfies I_{G}. Define the *possible pairs* in s in the following way

$$s.pos\text{-}pairs \;\triangleq\; \{(m, id) \in s.sr \mid id \in s.good\text{-}ids \wedge (s.mode_s = \mathtt{send} \Longrightarrow id \neq s.last_s)\}$$

The pairs in *s.pos-pairs* represent the "old" packets in *sr* that still have a chance of being successfully received by the receiver. Note, that we do not count $(s.current\text{-}msg_s, s.last_s)$ as a possible pair when $s.mode_s = \mathtt{send}$. Thus, the set of possible pairs in a state consists of packets for which the sender never stayed around to receive acknowledgement because of sender crashes. If no crashes have ever occurred the set is empty.

We want to order the possible pairs of a state into a list reflecting the order in which the pairs were sent. For this reason we—for any state s of A_{G} which satisfies I_{G}—define a total order on the packets in $s.sr$ based on the partial order on ids imposed by $s.used_s$ (see Section 8.3.2):

$$(m', id') <_u (m'', id'') \quad \text{if} \quad id' <_u id''$$

Invariant 8.4 Part 4 and Invariant 8.5 Part 1 imply that the order is indeed total on all packets in $s.sr$ for reachable states s of A_{G}.

Now, for any state s of A_{G} which satisfies I_{G}, define the *possible list*, written *s.pos-list*, to be the list obtained by ordering the elements of *s.pos-pairs* according to the ordering just introduced. (The closer to the head of the list the smaller the value according to the ordering). Thus, *s.pos-list* is the list of those packets (excluding the current packet) that still might be successfully received, and is ordered according to the order in which the packets were sent, with older packets occurring towards the head of the list. For all states s of A_{G} not satisfying I_{G}, define *s.pos-list* to be ε.

Define the function *messages* to extract the list of messages from a list of packets of *sr*. Thus, if $l = \langle (m_1, id_1), \ldots, (m_n, id_n) \rangle$ then $messages(l) \;\triangleq\; \langle m_1, \ldots, m_n \rangle$.

When the mode of the sender is either `needid` or `send`, the value of $current\text{-}msg_s$ is the message to be sent to the receiver. (This message has already been removed from buf_s). Now, the destiny of this message might be unknown if there has been a crash, because then the *id* that has been (or is to be) assigned to the message might not be in *good-ids* or it might be removed from *good-ids* before the message is received. The variable *current-ok* in A_{G} is precisely what we need to state this uncertainty. So, the flag (`OK` or `marked`) to be associated with the current message in the refinement mapping below is then derived from *current-ok* in state s in the following way:

$$s.current\text{-}flag \;\triangleq\; (\text{if } s.current\text{-}ok \text{ then } \mathtt{OK} \text{ else } \mathtt{marked})$$

We now define the *current queue*, i.e., the part of the queue at the D level that corresponds to the current message at the G level, as follows

$$s.\mathit{current\text{-}queue} \triangleq \begin{array}{l} \text{if } s.\mathit{mode}_s = \texttt{needid} \lor (s.\mathit{mode}_s = \texttt{send} \land s.\mathit{last}_s \in s.\mathit{good\text{-}ids}) \\ \text{then } \langle (s.\mathit{current\text{-}msg}_s, s.\mathit{current\text{-}flag}) \rangle \\ \text{else } \varepsilon \end{array}$$

When the mode of the sender is `send` and $\mathit{last}_s \in \mathit{good\text{-}ids}$ we denote by *current pair* the set containing the pair $(\mathit{current\text{-}msg}_s, \mathit{last}_s)$. In all other states this set is empty. Thus

$$s.\mathit{current\text{-}pair} \triangleq \begin{array}{l} \text{if } s.\mathit{mode}_s = \texttt{send} \land s.\mathit{last}_s \in s.\mathit{good\text{-}ids} \\ \text{then } \{(s.\mathit{current\text{-}msg}_s, s.\mathit{last}_s)\} \\ \text{else } \emptyset \end{array}$$

We define a function R_{GD} from $\mathit{states}(A_{\mathrm{G}})$ to $\mathit{states}(A_{\mathrm{D}})$. This function will in Lemma 8.14 be proved to be a refinement mapping from A_{G} to A_{D} with respect to I_{G} and I_{D}. In the definition, when we write e.g. "buf_r paired with `OK`", we mean the element of $(\mathit{Msg} \times \mathit{Flag})^*$ obtained from buf_r by pairing every message with `OK`.

Definition 8.13 (Refinement Mapping From A_{G} to A_{D})

If $s \in \mathit{states}(A_{\mathrm{G}})$ then define $R_{\mathrm{GD}}(s)$ to be the state $u \in \mathit{states}(A_{\mathrm{D}})$ such that

1. $u.\mathit{rec}_s = (s.\mathit{mode}_s = \texttt{rec})$

   $u.\mathit{rec}_r = (s.\mathit{mode}_r = \texttt{rec})$

2. $u.\mathit{queue}$ is the concatenation of

   - $s.\mathit{buf}_r$ paired with `OK`
   - $\mathit{messages}(s.\mathit{pos\text{-}list})$ paired with `marked`
   - $s.\mathit{current\text{-}queue}$
   - $s.\mathit{buf}_s$ paired with `OK`

3. $u.\mathit{status} =$

$$\begin{array}{lllr}
 & (\mathit{false}, \texttt{OK}) & \text{if } s.\mathit{mode}_s = \texttt{rec} & \text{A} \\
\text{else} & (?, \texttt{OK}) & \text{if } s.\mathit{buf}_s \neq \varepsilon & \text{B} \\
\text{else} & (?, s.\mathit{current\text{-}flag}) & \text{if } s.\mathit{mode}_s = \texttt{needid} & \text{C(i)} \\
 & (?, s.\mathit{current\text{-}flag}) & \text{if } s.\mathit{mode}_s = \texttt{send} \land s.\mathit{last}_s \in s.\mathit{good\text{-}ids} & \text{C(ii)} \\
 & (?, \texttt{OK}) & \text{if } s.\mathit{mode}_s = \texttt{send} \land s.\mathit{last}_s = s.\mathit{last}_r \land s.\mathit{buf}_r \neq \varepsilon & \text{C(iii)} \\
 & (\mathit{true}, \texttt{OK}) & \text{if } s.\mathit{mode}_s = \texttt{send} \land s.\mathit{last}_s = s.\mathit{last}_r \land s.\mathit{buf}_r = \varepsilon & \text{C(iv)} \\
 & (\mathit{true}, \texttt{marked}) & \text{if } s.\mathit{mode}_s = \texttt{send} \land s.\mathit{last}_s \neq s.\mathit{last}_r \land & \\
 & & \qquad (s.\mathit{last}_s, \mathit{true}) \in s.\mathit{rs} & \text{C(v)} \\
 & (\mathit{false}, \texttt{OK}) & \text{if } s.\mathit{mode}_s = \texttt{send} \land s.\mathit{last}_s \notin s.\mathit{good\text{-}ids} \land & \\
 & & \qquad s.\mathit{last}_s \neq s.\mathit{last}_r \land & \\
 & & \qquad (s.\mathit{last}_s, \mathit{true}) \notin s.\mathit{rs} & \text{C(vi)} \\
 & (s.\mathit{current\text{-}ack}_s, \texttt{OK}) & \text{if } s.\mathit{mode}_s = \texttt{idle} & \text{C(vii)}
\end{array}$$

■

It is easy to see that the cases in Part 3 of the definition are exhaustive. However, the cases C(ii)–C(vi) are overlapping in some non-reachable states (where $s.last_s \in s.good\text{-}ids \wedge (s.last_s = s.last_r \vee (s.last_s, true) \in s.rs)$, cf. Invariants 8.4 Part 7 and 8.10 Part 2). Since we shall only be interested in the image of states satisfying the invariants, this is not a problem in practice. However, to make R_{GD} a mapping from all states of A_{G} to states of A_{D}, we adopt the convention that in cases C(ii)–C(vi) the first case (from top to bottom) that is satisfied by a given state is chosen.

The intuition behind R_{GD} is as follows: When either the sender or receiver in A_{G} is in mode `rec` this, of course, corresponds to A_{D} having either rec_s or rec_r set to *true*, respectively. This is captured in Part 1.

Part 2 associates flags with the messages between the sender and the receiver. The messages in buf_s and buf_r all get paired with the flag `OK`. That is because these messages are "safe" as long as no new crashes occur. If a crash occurs at, e.g., the sender side, then of course the elements in buf_s will be deleted, but this corresponds in A_{D} to marking these elements and dropping them. So, the flag associated with a message (or the *status* below) should indicate the situation for that message (or *status*) here and now.

The messages in *pos-list* are all paired with `marked`. As explained above, when *pos-list* was defined, all elements of *pos-list* are "old" packets that still might be successfully received. However, elements of *pos-list* lose this possibility (i.e., are removed from *pos-list*) if a packet with higher *id* is successfully received by the receiver (since otherwise A_{G} could rearrange messages). Thus, messages in *pos-list* might be lost without any crashes occurring. For this reason these messages are paired with `marked` in R_{GD}.

In *current-queue* the flag is *current-flag*. If the receiver has not been in `rec` mode (which in this situation implies *current-ok* = *true*) since the last *prepare* action, we know that the *id* assigned (or to be assigned) to the current message is in $good_r$ (cf. Invariant 8.6 Parts 5 and 6). Unless crashes occur this will be the case until the current message is successfully received. (Note, that the successful receipt of a message from *pos-list* cannot cause the *id* of the current message to be removed from $good_r$ since all messages in *pos-list* have *id*s less that this *id*). So, in this situation *current-flag* = `OK`. On the other hand, if a crash has occurred the current message might still be successfully received but it could be lost. In this case *current-flag* = `marked` as required.

Part 3 deals with the *status*. First, recall that in A_{D} *status* records the status of the *last message sent* to the system.

Case A deals with the situation where the sender has crashed. In this situation the last message sent can only cause a negative acknowledgement to the user. Therefore *status* = (*false*, `OK`).

In Case B, $mode_s \neq$ `rec` and $buf_s \neq \varepsilon$. Thus, the last element sent is, for now, sitting safely in buf_s. For this reason we have *status* = (?, `OK`).

C(i) and C(ii) describe to the situation where the last element sent is in *current-queue*. Here *status* = (?, *current-flag*), where *current-flag* = `marked` is there has been a crash so that it is permitted to "lose" *status* (i.e., change it to (*false*, `OK`)).

In C(iii) the last message sent has been received by the receiver and is sitting safely in buf_r.

In C(iv) this message has been passed on to the user and the receiver is in the process of sending positive acknowledgements to the sender. This is a sure positive status, thus, *status* = (*true*, `OK`).

Case C(v) then describes the situation where a positive acknowledgement has been sent by the receiver, but where the receiver subsequently has crashed. In this situation the positive acknowledgement might eventually be successfully received by the sender, but, since the sender keeps on sending its current packet until it receives an acknowledgement, the receiver might issue *negative* acknowledgements for the current message and these negative acknowledgements could pass the positive acknowledgements in rs such that the sender receives a negative acknowledgement for the current message. The latter situation corresponds in A_{D} to *status* being lost. This explains why $status = (true, \texttt{marked})$ in case C(v). Note, that in the situation just explained, the current message *has* been successfully delivered to the user, but a subsequent crash could cause *status* to be lost anyway (recall that this is allowed by the specification).

Case C(vi) actually describes two situations: (a) the *id* assigned to the current message is such that the current message can never be successfully received by the receiver. Thus, the receiver can only issue negative acknowledgements for this message. The other situation is: (b) the current message *has* been successfully received, but the receiver crashed before successfully placing a positive acknowledgement on the channel rs. Again, only negative acknowledgements can be received by the sender. This explains $status = (false, \texttt{OK})$.

Finally, case C(vii) reflects the acknowledgement received by the sender for the (last) current message.

After having used our knowledge and intuition about A_{G} and A_{D} to define R_{GD}, we still need to verify that R_{GD} is in fact a refinement mapping from A_{G} to A_{D} (with respect to I_{G} and I_{D}). The following lemma states that this is the case.

Lemma 8.14

$A_{\mathrm{G}} \leq_R A_{\mathrm{D}}$ *via* R_{GD}.

Proof

We prove that R_{GD} is a refinement mapping from A_{G} to A_{D} with respect to I_{G} and I_{D}. We check the two conditions (which we call base case and inductive case, respectively) of Definition 5.2.

Base Case

It is easy to see that for any start state s of A_{G}, $R_{\mathrm{GD}}(s)$ is a start state of A_{D}.

Inductive Case

Assume $(s, a, s') \in steps(A_{\mathrm{G}})$ such that s and s' satisfy I_{G} and $R_{\mathrm{GD}}(s)$ satisfies I_{D} (Invariant 7.1). Below we consider cases based on a (and sometimes subcases of each case) and for each (sub)case we define a finite execution fragment α of A_{D} of the form $(R_{\mathrm{GD}}(s), a', u'', a'', u''', \ldots, R_{\mathrm{GD}}(s'))$ with $trace(\alpha) = trace(a)$. For brevity we let u denote $R_{\mathrm{GD}}(s)$ and u' denote $R_{\mathrm{GD}}(s')$.

Unless otherwise stated we let Part 1–3 refer to the three parts of Definition 8.13.

$\underline{a = send\_msg(m)}$

We consider cases based on $s.mode_s$.

1. $s.mode_s \neq \texttt{rec}$

   Then, it is easy to see that $(u, send\_msg(m), u')$ is a step of A_{D} and thus a finite execution fragment with the right trace.

2. $s.mode_s = \texttt{rec}$

   Then $s' = s$, so also $u' = u$.

   We show that $(u, send\_msg(m), u'', mark(I), u''', drop(I), u')$, where u'', u''', and I are defined below, is a finite execution fragment of A_{D} by showing that $(u, send\_msg(m), u'')$, $(u'', mark(I), u''')$, and $(u''', drop(I), u')$ are steps of A_{D}. Clearly the execution fragment has the right trace.

   Define
   $$\begin{aligned} u''.rec_s &= u.rec_s \\ u''.rec_r &= u.rec_r \\ u''.queue &= u.queue \mathbin{\hat{}} (m, \texttt{OK}) \\ u''.status &= (?, \texttt{OK}) \end{aligned}$$
   Then obviously $(u, send\_msg(m), u'') \in steps(A_{\mathrm{D}})$.

   Define
   $$\begin{aligned} u'''.rec_s &= u.rec_s\,(= true) \\ u'''.rec_r &= u.rec_r \\ u'''.queue &= u.queue \mathbin{\hat{}} (m, \texttt{marked}) \\ u'''.status &= u''.status \end{aligned}$$
   Thus the only difference between u'' and u''' is that the element at the end of *queue* is `marked` in u'''. Define $I = \{maxidx(u''.queue)\}$. Then, since $u'''.rec_s = true$, obviously $(u'', mark(I), u''') \in steps(A_{\mathrm{D}})$.

   Finally, we have to show that $(u''', drop(I), u') \in steps(A_{\mathrm{D}})$. First note that *drop* is enabled in u''' since I contains the index of the last element of $u'''.queue$ and this element is marked by explicit construction. It now suffices that the four state variables of A_{D} are handled correctly.

   rec_s and rec_r:

   We have (by construction and the fact that $u' = u$) $u'''.rec_s = u'.rec_s$ and $u'''.rec_r = u'.rec_r$ as required by the definition of $drop(I)$ in A_{D}.

   queue:

   We have (again by construction and the fact that $u' = u$) $u'''.queue = u'.queue \mathbin{\hat{}} (m, \texttt{marked})$. Thus, since $drop(I)$ requires the last element of *queue* to be deleted, *queue* is handled correctly.

   status:

   Since the element at the end of *queue* is deleted, the definition of $drop(I)$ requires that $u'.status = (false, \texttt{OK})$, but this is the case since $u.status = (false, \texttt{OK})$ (from the definition of R_{GD}) and $u' = u$.

$a = receive\_msg(m)$

We show that $(u, receive\_msg(m), u') \in steps(A_{\mathrm{D}})$. The step clearly has the right trace.

From the precondition of the $receive\_msg(m)$ steps in A_{G} we have that $s.mode_r = \texttt{rcvd}$, $s.buf_r \neq \varepsilon$, and $head(s.buf_r) = m$. The definition of R_{GD} then implies that $u.queue \neq \varepsilon$ and $head(u.queue) = (m, \texttt{OK})$. Thus, from the definition of the $receive\_msg(m)$ steps in A_{D} we see that $receive\_msg(m)$ is enabled in u. It now suffices to show that the four state variables of A_{D} are handled correctly.

rec_s, rec_r, and *queue*:

It is easy to see that $u'.rec_s = u.rec_s$, $u'.rec_r = u.rec_r$, and $u'.queue = tail(u.queue)$, as required by the definition of $receive\_msg(m)$ in A_{D}.

status:

We consider cases based on which condition (A, B, C(i)–C(vii)) s satisfies in Part 3.

Suppose s satisfies the condition in case A, C(v), C(vi), or C(vii). Then s' satisfies the same condition, so $u.status = u'.status$. Since in all cases $u.status.stat \neq ?$, leaving *status* unchanged is permitted by the definition of *receive_msg*(m) in A_{D}.

Suppose s satisfies the condition in case B, C(i), or C(ii). Then s' satisfies the same condition, so $u.status = u'.status$. In all three cases it is easy to see that $u'.queue \neq \varepsilon$ so it is allowed by the definition of *receive_msg*(m) in A_{D} to leave *status* unchanged.

Suppose s satisfies the condition in case C(iii). If $s'.buf_r \neq \varepsilon$ then s' also satisfies this condition but in this case $u'.queue \neq \varepsilon$ so it is permitted by the definition of *receive_msg*(m) in D to leave *status* unchanged. So, assume $s'.buf_r = \varepsilon$. Then s' satisfies the condition in case C(iv). Thus, $u.status = (?, \texttt{OK})$ and $u'.status = (true, \texttt{OK})$. Also, $s'.buf_s = \varepsilon$ and Invariant 8.8 Part 2 implies that both $s'.pos\text{-}list = \varepsilon$ and $s'.current\text{-}queue = \varepsilon$. Then, since $s'.buf_r = \varepsilon$, $u.queue = \varepsilon$. Thus, changing *status* from $(?, \texttt{OK})$ to $(true, \texttt{OK})$ is as required by *receive_msg*(m) in A_{D}.

Finally, the precondition of *receive_msg*(m) in A_{G} implies that s cannot satisfy the condition in case C(iv).

$a = ack(b)$

We show that $(u, ack(b), u') \in steps(A_{\mathrm{D}})$. The step clearly has the right trace.

By definition of $ack(b)$ in A_{G} we have $s' = s$ so also $u' = u$.

From the precondition of $ack(b)$ in A_{G} we have $s.mode_s = \texttt{idle}$, $s.buf_s = \varepsilon$, and $s.current\text{-}ack_s = b$. Then $u.status = (s.current\text{-}ack_s, \texttt{OK}) = (b, \texttt{OK})$ (by case C(vii) of Part 3). Thus, $ack(b)$ is enabled in u.

Since $u.status.stat = \texttt{OK}$, it is now easily seen that $(u, ack(b), u')$ is a step of D.

$a = crash_s$

We show that $(u, crash_s, u'', mark(I), u''', drop(I'), u')$, where u'', u''', I, and I' are defined below, is a finite execution fragment of A_{D} by showing that $(u, crash_s, u'')$, $(u'', mark(I), u''')$, and $(u''', drop(I'), u')$ are steps of A_{D}. Clearly the execution fragment has the right trace.

Define
$$\begin{aligned} u''.rec_s &= true \\ u''.rec_r &= u.rec_r \\ u''.queue &= u.queue \\ u''.status &= u.status \end{aligned}$$
Then clearly $(u, crash_s, u'') \in steps(A_{\mathrm{D}})$.

First let $i_{cq} = |s.buf_r| + |s.pos\text{-}list|$. Then, define

$$
\begin{array}{lcl}
u'''.rec_s & = & u''.rec_s \\
u'''.rec_r & = & u''.rec_r \\
(u'''.queue, I, I') & = & \begin{cases} (u''.queue, \emptyset, \emptyset) & \text{if } s.mode_s \in \{\mathtt{idle}, \mathtt{rec}\} \vee \\ & \quad (s.mode_s = \mathtt{send} \wedge \\ & \quad\; s.last_s \notin s.good\text{-}ids) \\ (q, \{i_{cq}\}, \emptyset) & \text{if } s.mode_s = \mathtt{send} \wedge \\ & \quad s.last_s \in s.good\text{-}ids \wedge \\ & \quad (s.current\text{-}msg_s, s.last_s) \in s.sr \\ & \text{where } q = mark(u''.queue, \{i_{cq}\}) \\ (q, \{i_{cq}\}, \{i_{cq}\}) & \text{otherwise} \\ & \text{where } q = mark(u''.queue, \{i_{cq}\}) \end{cases} \\
u'''.status & = & (u''.status.stat, \mathtt{marked})
\end{array}
$$

Since $u''.rec_s = true$, clearly $mark(I)$ is enabled in u''. To prove that $(u'', mark(I), u''') \in steps(A_{\mathrm{D}})$ it now suffices to show that all four state variables of A_{D} are handled correctly.

rec_s and rec_r:
Leaving rec_s and rec_r unchanged is as required by the definition of $mark(I)$ in A_{D}.
queue:
By explicit construction of $u'''.queue$ and I, it is easy to see that *queue* is handled correctly. A_{D}.
status:
Marking *status* is allowed by the definition of $mark(I)$ in A_{D}.
Thus, $(u'', mark, u''') \in steps(A_{\mathrm{D}})$.

Finally, we must show that $(u''', drop(I'), u') \in steps(A_{\mathrm{D}})$. Slearly $drop(I')$ is enabled in u''', so it suffices to show that the four state variables of A_{D} are handled correctly.

rec_s and rec_r:
We have $u'.rec_s = true = u'''.rec_s$ and $u'.rec_r = u.rec_r = u'''.rec_r$. Leaving rec_s and rec_r unchanged is as required by the definition of $drop(I')$ in A_{D}.
status:
We have $u'.status = (false, \mathtt{OK})$ since $s'.mode_s = \mathtt{rec}$, and this is allowed by the definition of $drop(I')$ in A_{D}.
queue:
First, assume $s.mode_s \in \{\mathtt{idle}, \mathtt{rec}\}$ or $s.mode_s = \mathtt{send} \wedge s.last_s \notin s.good\text{-}ids$. Then it is easy to see that $u'.queue = u'''.queue = u.queue$. Leaving *queue* unchanged is as required by the definition of $drop(I')$ in A_{D} since in this case $I' = \emptyset$.

Next, assume $(s.mode_s = \mathtt{send} \wedge s.last_s \in s.good\text{-}ids \wedge (s.current\text{-}msg_s, s.last_s) \notin s.sr)$ or $s.mode_s = \mathtt{needid}$. Then we have $s.current\text{-}queue = \langle (s.current\text{-}msg_s, s.current\text{-}flag) \rangle$ and $s'.current\text{-}queue = \varepsilon$. But the other three ($buf_r$, buf_s, and *pos-list*) parts that make up the abstraction of a *queue* in A_{D} are unchanged. (Note, in the definition of $u'''.queue$ is this case that the element in $u''.queue$ that corresponds to *s.current-queue* has index i_{cq}). Then, it is easy to see that $u'.queue = delete(u'''.queue, \{i_{cq}\})$. Thus, by explicit construction of I' and the definition of $drop(I')$ it is seen that *queue* is handled as required.

Finally, assume $(s.mode_s = \mathtt{send} \wedge s.last_s \in s.good\text{-}ids \wedge (s.current\text{-}msg_s, s.last_s) \in s.sr)$. Again, we have $s.current\text{-}queue = \langle (s.current\text{-}msg_s, s.current\text{-}flag) \rangle$ and $s'.current\text{-}queue = \varepsilon$. But in this case we have $s'.pos\text{-}pairs = s'.pos\text{-}pairs \cup (s.current\text{-}msg_s, s.last_s)$. Then Invariant 8.8 Part 1 implies that $s'.pos\text{-}list = s.pos\text{-}list \,\hat{}\, (s.current\text{-}msg_s, s.last_s)$. We now have that the only difference between $u'.queue$ and $u.queue$ is that one of the elements (the one corresponding to $(s.current\text{-}msg_s, s.last_s))$ in $u'.queue$ is `marked` (which it might not be in $u.queue$). But this gives us $u'.queue = u'''.queue$, and since $I' = \emptyset$ in this case, it is seen that $queue$ is handled as required by the definition of $drop(I')$ in A_D.

Thus, $(u''', drop, u') \in steps(A_D)$ as required.

$\underline{a = crash_r}$

We show that $(u, crash_r, u'', mark(I), u')$, where u'' and I are defined below, is a finite execution fragment of A_D by showing that $(u, crash_r, u'')$ and $(u'', mark(I), u')$ are steps of A_D. Clearly the execution fragment has the right trace.

$$\begin{array}{lll} \text{Define } u''.rec_r & = & true \\ \quad u''.rec_s & = & u.rec_s \\ \quad u''.queue & = & u.queue \\ \quad u''.status & = & u.status \end{array}$$

Clearly $(u, crash_r, u'') \in steps(A_D)$.

Define,

$$I = \begin{cases} \{|s.buf_r| + |s.pos\text{-}list|\} & \text{if } s.mode_s = \texttt{needid} \vee (s.mode_s = \texttt{send} \wedge s.last_s \in s.good\text{-}ids) \\ \emptyset & \text{otherwise} \end{cases}$$

We now show that $(u'', mark(I), u') \in steps(A_D)$. First note that since $u''.rec_r = true$, the definitions of I and R_{GD} imply that $mark(I)$ is enabled in u''. It thus suffices to show that the four state variables of A_D are handled correctly.

rec_s and rec_r:

We have $u'.rec_r = true = u''.rec_r$ and $u'.rec_s = u.rec_s = u''.rec_s$. Leaving rec_s and rec_r unchanged is as required by the definition of $mark(I)$ in A_D.

$queue$ and $status$:

First assume $s.mode_s = \texttt{needid}$ or $s.mode_s = \texttt{send} \wedge s.last_s \in s.good\text{-}ids$. In this case the only difference in states s and s' of the four components that make up the abstraction of a $queue$ in Part 2 is that the element in $current\text{-}queue$ is `marked` in s' whereas it might be `OK` in s. So, the only difference between $u''.queue (= u.queue)$ and $u'.queue$ is that the element with index $|s.buf_r| + |s.pos\text{-}list|$ has changed its flag to `marked`, but by definition of I in this case, this is as required by the definition of $mark(I)$ in A_D. For $status$, if $s.buf_s \neq \varepsilon$ then $u.status = u'.status = (?, \texttt{OK})$ by Part 3B. But leaving $status$ unchanged is allowed by the definition of $mark(I)$ in A_D. If $s.buf_s = \varepsilon$ then s satisfies either Part 3C(i) or 3C(ii) and s' satisfies the same part. In this case $status$ might change its flag from `OK` to `marked` but again this is allowed by the definition of $mark(I)$ in A_D.

Finally, in all other cases $u.queue = u'.queue$ and $u.status = u'.status$ so $mark(I)$ should be a no-op, but again this is allowed by the definition of $mark(I)$ in A_D since in this case $I = \emptyset$.

$\underline{a = recover_s}$

We show that $(u, mark(I), u'', drop(I), u''', recover_s, u')$, where u'', u''', and I are defined below, is a finite execution fragment of A_D by showing that $(u, mark(I), u'')$, $(u'', drop(I), u''')$, and $(u''', recover_s, u')$ are steps of D. Clearly the execution fragment has the right trace.

Define $I = \{i \mid maxidx(u.queue) - (|s.buf_s| - 1) \leq i \leq maxidx(u.queue)\}$.

Thus, I contains the indices of the last $|s.buf_s|$ elements in $u.queue$.

$$\begin{aligned}\text{Define } u''.rec_s &= u.rec_s\\ u''.rec_r &= u.rec_r\\ u''.queue &= mark(u.queue, I)\\ u''.status &= u.status\end{aligned}$$

Since $s.mode_s = \texttt{rec}$ we have $u.rec_r = true$ so the definition of I implies that $mark(I)$ is enabled in u. Then it is easy to see that $(u, mark(I), u'') \in steps(A_{\mathrm{D}})$.

$$\begin{aligned}\text{Define } u'''.rec_s &= u''.rec_s\\ u'''.rec_r &= u''.rec_r\\ u'''.queue &= delete(u''.queue, I)\\ u'''.status &= (false, \mathtt{OK})\end{aligned}$$

The definitions of I and $u''.queue$ implies that $drop(I)$ is enabled in u''. Now, to show that $(u'', drop(I), u''') \in steps(A_{\mathrm{D}})$, it suffices to show that the four state variables of A_{D} are handled correctly.

rec_s and rec_r:
  Leaving rec_s and rec_r unchanged is as required by the definition of $drop(I)$ in A_{D}.
$queue$:
  By explicit construction of $u'''.queue$, clearly $queue$ is handled correctly.
$status$:
  Since $drop(I)$ is always allowed to change $status$ to $(false, \mathtt{OK})$, $status$ is handled correctly.
Thus, $(u'', drop(I), u''') \in steps(A_{\mathrm{D}})$.

Finally, we prove that $(u''', recover_s, u') \in steps(A_{\mathrm{D}})$. Since $u'''.rec_s = u''.rec_s = u.rec_s = true$, we have that $recover_s$ is enabled in u'''. We show that the four state variables of A_{D} are handled correctly.

rec_s and rec_r:
  Leaving rec_r unchanged and changing rec_s from $true$ to $false$ is as required by the definition of $recover_s$ in A_{D}.
$queue$:
  Note that $s.current\text{-}queue = s'.current\text{-}queue = \varepsilon$, $s.pos\text{-}list = s'.pos\text{-}list$, and $s.buf_r = s'.buf_r$. So, since buf_s is emptied in the $recover_s$ step of A_{G}, the only difference between $u.queue$ and $u'.queue$ is that the last $|s.buf_s|$ elements of $u.queue$ are missing in $u'.queue$. Thus, $u'.queue = u'''.queue$ as required by the definition of $recover_s$ in A_{D}.
$status$:
  Since $s'.mode_s = \texttt{idle}$, $s'.buf_s = \varepsilon$, and $s'.current\text{-}ack_s = false$, we have $u'.status = (false, \mathtt{OK})$ by Part 3(vii). Thus, $u'.status = u'''.status$ as required by the definition of $recover_s$ in A_{D}.
Thus, $(u''', recover_s, u') \in steps(A_{\mathrm{D}})$.

$\underline{a = recover_r}$

We show that $(u, mark(I), u'', drop(I), u''', recover_r, u')$, where u'', u''', and I are defined below, is a finite execution fragment of A_{D} by showing that $(u, mark(I), u'')$, $(u'', drop(I), u''')$, and $(u''', recover_r, u')$ are steps of A_{D}. Clearly the execution fragment has the right trace.

$$\begin{aligned}\text{First, define } u''.rec_s &= u.rec_s\\ u''.rec_r &= u.rec_r\\ u'''.rec_s &= u''.rec_s\\ u'''.rec_r &= u''.rec_r\end{aligned}$$

Below we define I so that it contains indices of $u.queue$ and indices of marked elements in $u''.queue$. Then, since $s.mode_r = \texttt{rec}$ we have $u.rec_r = true$, so $mark(I)$ is enabled in u, $drop(I)$ is enabled in u'', and finally $recover_r$ is enabled in u''' since we also have $u'''.rec_r = true$.

We now show that the four state variables in A_{D} are handled correctly by all steps in the execution fragment.

rec_s and rec_r:

As in the case $a = recover_s$ above it is easy to see that rec_s and rec_r are handled correctly.

queue:

Note that $s'.good\text{-}ids \subseteq s.good\text{-}ids$ since $issued_r$ might be extended in the $recover_r$ step of A_{G}. This leads to the observations that (a) either $s'.current\text{-}queue = s.current\text{-}queue$ or $s'.current\text{-}queue = \varepsilon$, and (b) $s'.pos\text{-}pairs \subseteq s.pos\text{-}pairs$ so that $s'.pos\text{-}list$ can be obtained from $s.pos\text{-}list$ by deleting some elements. Also we have $s.buf_s = s'.buf_s$ and $s'.buf_r = \varepsilon$. Thus, $u'.queue$ can be obtained from $u.queue$ by deleting some elements. By letting I be the indices of these elements, the elements are marked in the $mark(I)$ step and then deleted in the $drop(I)$ step. Thus, *queue* is handled correctly.

status:

We consider cases based on which condition in Part 3 is satisfied by s.

Suppose s satisfies condition A. Then so does s' so we have $u.status = u'.status = (false, \texttt{OK})$ which is allowed by the execution fragment of A_{D}.

If s satisfies condition B, then so does s' so we have $u.status = u'.status = (?, \texttt{OK})$. This is allowed by the execution fragment of A_{D} provided that the element at the end of $u.queue$ was not deleted in the $drop(I)$ step but this is the case (that it was not deleted) since $s.buf_s = s'.buf_s \neq \varepsilon$.

Also, if s satisfies C(i) then so does s' (with $s.current\text{-}flag = s'.current\text{-}flag$), and this is allowed since $s.buf_s = s'.buf_s = \varepsilon$ and $s.current\text{-}queue = s'.current\text{-}queue \neq \varepsilon$ so the last element of $u.queue$ was not deleted in the $drop(I)$ step.

If s satisfies C(ii) then $s.last_s = s'.last_s \notin ids(s.rs) = ids(s'.rs)$ (by Invariant 8.10 Part 2) and $s.last_s \neq \texttt{nil}$ (by Invariant 8.1 Part 2). Now, if $s'.last_s \in s'.good\text{-}ids$ then s' satisfies C(ii) so $s.current\text{-}queue = s'.current\text{-}queue \neq \varepsilon$. As for case C(i) we see that this is allowed. If $s'.last_s \notin s'.good\text{-}ids$ then, since $s'.last_r = \texttt{nil} \neq s'.last_s$, s' satisfies condition C(vi), so $u'.status = (false, \texttt{OK})$ which is allowed by the execution fragment.

Now, suppose s satisfies C(iii). Then Invariant 8.4 Part 7 implies $s.last_s \notin s.good\text{-}ids$ which again implies $s'.last_s \notin s'.good\text{-}ids$ since $s'.good\text{-}ids \subseteq s.good\text{-}ids$. Invariant 8.9 Part 3 implies $(s.last_s, true) \notin s.rs$, i.e., $(s'.last_s, true) \notin s'.rs$. Thus, s' satisfies condition C(vi), so $u.status = (false, \texttt{OK})$ which is allowed by the execution fragment of A_{D}.

If s satisfies C(iv) we consider two subcases. If $(s.last_s, true) \notin s.rs$ the case is similar to case C(iii) above. So assume $(s.last_s, true) \in s.rs$. Then s' satisfies C(v) so $u.status = (true, \texttt{OK})$ and $u'.status = (true, \texttt{marked})$. This marking of status is allowed by $mark(I)$ in A_{D}. Then total change of status is allowed is the element at the end of $u'.queue$ is not deleted in the $drop(I)$ step. Invariant 8.8 Part 2 implies that $s.current\text{-}queue = s.pos\text{-}list = \varepsilon$ so $u.queue = \varepsilon$, thus there is no last element to be deleted. That suffices.

If s satisfies C(v), then so does s' (Invariant 8.1 Part 2 implies $s'.last_s \neq \texttt{nil} = s'.last_r$). Thus, $s.status = s'.status = (true, \texttt{marked})$. This is allowed since $u.queue = \varepsilon$ (so the last element of the *queue* cannot be deleted in the $drop(I)$ step). To see why $u.queue = \varepsilon$, we have from C(v) that $s.buf_s = \varepsilon$ and Invariants 8.8 Part 3 and 8.9 Part 3 imply $s.current\text{-}queue = s.pos\text{-}list = s.buf_r = \varepsilon$. That suffices.

If s satisfies condition C(vi) then so does s' (arguments as above). Thus, $u.status = u'.status = (\mathit{false}, \mathtt{OK})$ which is allowed by the execution fragment.

Finally, if s satisfies condition C(vii), then so does s'. We then have $u.status = u'.status = (s.current\text{-}ack_s, \mathtt{OK})$. This is easily seen to be allowed if $s.current\text{-}ack_s = \mathit{false}$. So, assume $s.current\text{-}ack_s = \mathit{true}$. Then having $u.status = u'.status = (\mathit{true}, \mathtt{OK})$ is allowed provided the element at the end of $u.queue$ is not deleted in the $drop(I)$ step. A sufficient condition is to show $u.queue = \varepsilon$. From C(vii) we have $s.buf_s = s.current\text{-}queue = \varepsilon$ and Invariants 8.8 Part 4 and 8.9 Part 4 imply $s.pos\text{-}list = s.buf_r = \varepsilon$. Thus, $u.queue = \varepsilon$.

$\underline{a = prepare}$

We consider two cases

- $s.mode_r = \mathtt{rec}$

  We show that $(u, mark(I), u') \in steps(A_{\mathrm{D}})$, where $I = |s.buf_r| + |s.pos\text{-}list|$. This step (and execution fragment) clearly has the right trace (the empty trace).

  Since $s.mode_r = \mathtt{rec}$, we have $u.rec_r = \mathit{true}$, so clearly $mark(I)$ is enabled in u.

  We show that the four state variables of A_{D} are handled correctly.

  rec_s and rec_r:

  We have $s.mode_s = \mathtt{idle}$ and $s'.mode_s = \mathtt{needid}$, so $u.rec_s = u'.rec_s = \mathit{false}$ which is as required by the definition of $mark(I)$ in A_{D}. From the case hypothesis and the definition of $prepare$ in A_{G}, we have $s.mode_r = s'.mode_r = \mathtt{rec}$, so $u.rec_r = u'.rec_r = \mathit{true}$ which is also as required by the definition of $mark(I)$.

  $queue$:

  Note that the element at the head of buf_s is moved to $current\text{-}msg_s$ in the $prepare$ step of A_{G}. From the definition of R_{GD} we have that this element goes from being $\mathtt{OK}$ when it was in buf_s to being $\mathtt{marked}$ ($s.mode_r = \mathtt{rec}$ implies, by Invariant 8.6 Part 2, $s'.current\text{-}ok = \mathit{false}$ which in turn implies $s'.current\text{-}flag = \mathtt{marked}$) when it is in $current\text{-}queue$. Neither buf_r nor $pos\text{-}list$ are changed in the $prepare$ step. Thus, $u'.queue$ is the same as $u.queue$ except that the message at position $|s.buf_r| + |s.pos\text{-}list|$ is $\mathtt{marked}$ in $u.queue$ and $\mathtt{OK}$ in $u.queue$. This is as required by the definition of $mark(I)$ in A_{D}.

  $status$:

  We have $u.status = (?, \mathtt{OK})$ since $s.buf_s \neq \varepsilon$ (from the precondition of the $prepare$ step). Either state s' satisfies Condition 3B in which case $u'.status = (?, \mathtt{OK})$ or s' satisfies condition C(i) in which case $u'.status = (?, \mathit{false})$. Both of these situations are allowed by the definition of $mark(I)$ in A_{D}.

  Thus, $(u, mark(I), u') \in steps(A_{\mathrm{D}})$.

- $s.mode_r \neq \mathtt{rec}$

  Here we have $s'.current\text{-}flag = \mathtt{OK}$ from the effect of the $prepare$ step, so with arguments similar to those used in the previous case it is easy to show show that $u' = u$. Thus, the execution fragment consisting of only the state u has the right trace. That suffices.

$\underline{a = choose\_id(id)}$

We consider two cases

- $s'.last_s \notin s'.good\text{-}ids$

  We show that $(u, drop(I), u') \in steps(A_D)$, where $I = \{|s.buf_r| + |s.pos\text{-}list|\}$. This step (and finite execution fragment) clearly has the right trace (the empty trace).

  We show that the four state variables of A_D are handled correctly.

  rec_s and rec_r:

  We have $s.mode_s = \texttt{needid}$, $s'.mode_s = \texttt{send}$, and $s.mode_r = s'.mode_r$ which implies $u.rec_s = u'.rec_s$ and $u.rec_r = u'.rec_r$ as required by the definition of $drop(I)$ in A_D.

  $queue$:

  We note that $s'.buf_s = s.buf_s$, $s'.pos\text{-}list = s.pos\text{-}list$, and $s'.buf_r = s.buf_r$. However, $s'.current\text{-}queue = \varepsilon$ but $s.current\text{-}queue \neq \varepsilon$. Thus, $u'.queue$ can be obtained from $u'.queue$ by deleting the element that corresponds to $s.current\text{-}queue$. From the case hypothesis and the definition of $choose\_id(id)$ in A_G we have $s.good_s \not\subseteq s.good\text{-}ids$ (note, $s'.good\text{-}ids = s.good\text{-}ids$). Now, since $s.mode_s = \texttt{needid}$, Invariant 8.6 Part 5 implies $s.current\text{-}ok = false$ which again implies $s.current\text{-}flag = \texttt{marked}$. Thus, the flag of the element $s.current\text{-}queue$ is $\texttt{marked}$. Now, $s.current\text{-}queue$ corresponds to position $|s.buf_r| + |s.pos\text{-}list|$ in $u.queue$. Since this element is marked, $drop(I)$ is enabled in u. Furthermore, it is easy to see that $queue$ is handled correctly.

  $status$:

  If $s.buf_s \neq \varepsilon$ then also $s'.buf_s \neq \varepsilon$ so both s and s' satisfy condition 3B. Thus, $u.status = u'.status = (?, \texttt{OK})$. This is allowed by $drop(I)$ since the element at the end of $queue$ is not deleted because $s.buf_s = s'.buf_s \neq \varepsilon$. Now, if $s.buf_s = \varepsilon$, s satisfies condition 3C(i), i.e., $u.status = (?, false)$ since $s'.current\text{-}flag = \texttt{marked}$ (see the discussion for $queue$ above). We show that s' satisfies 3C(vi) such that $u'.status = (false, \texttt{OK})$ which is allowed by $drop(I)$. This amounts to showing $s'.last_s \neq s'.last_r$ and $(s'.last_s, true) \notin s'.rs$ since the case hypothesis and the definition of $choose\_id(id)$ give us the rest:

  From the definition of $choose\_id(id)$ we get $id = s'.last_s \in s.good_s$. Invariant 8.2 Part 1 then implies $s'.last_s \notin s.used_s$. Also, $s'.last_s \neq \texttt{nil}$ by Invariant 8.1 Part 2. Invariant 8.4 Part 8 implies (since $s.last_r = s'.last_r$) that $s'.last_r = \texttt{nil}$ or $s'.last_r \in s.used_s$. Thus, we get $s'.last_s \neq s'.last_r$ as required. Also, since $s'.last_s \notin s.used_s$, Invariant 8.4 Part 6 implies $(s'.last_s, true) \notin s.rs = s'.rs$ as required.

  Thus, $(u, drop(I), u') \in steps(A_D)$.

- $s'.last_s \in s'.good\text{-}ids$

  We show $u' = u$ by comparing the four state variables of A_D in u and u'. The execution fragment u then has the right properties.

  rec_s and rec_r:

  We have $s.mode_s = \texttt{needid}$, $s'.mode_s = \texttt{send}$, and $s.mode_r = s'.mode_r$ which implies $u.rec_s = u'.rec_s$ and $u.rec_r = u'.rec_r$ as required.

  $queue$:

  Her we have $s'.current\text{-}queue = s.current\text{-}queue$. Then it is easy to see that $u'.queue = u.queue$.

  $status$:

  We have that either both s and s' satisfy condition 3B, or s satisfies 3C(i) and s' satisfies 3C(ii). In both cases $u'.status = u.status$ as required.

$\underline{a = send\_pkt_{sr}(m, id)}$

We show $u = u'$ by comparing the four state variables of A_D in u and u'. The execution fragment

u then has the right properties.

rec_s and rec_r:

We have $s.mode_s = s'.mode_s$ and $s.mode_r = s'.mode_r$ which implies $u.rec_s = u'.rec_s$ and $u.rec_r = u'.rec_r$ as required.

$queue$:

We have $s'.buf_s = s.buf_s$, $s'.current\text{-}queue = s.current\text{-}queue$ and $s'.buf_r = s.buf_r$. The $send\_pkt_{sr}(m, id)$ step might add some copies of $(m, last_s)$ to the channel sr. However, since $mode_s = \texttt{send}$, this does not change the value of $pos\text{-}pairs$, so $s'.pos\text{-}list = s.pos\text{-}list$. Thus, $u'.queue = u.queue$.

$status$:

Whatever condition in Part 3 of Definition 8.13 s satisfies, s' satisfies the same. This implies $u'.status = u.status$.

$a = receive\_pkt_{sr}(m, id)$

Since this step may remove the last copy of (m, id) from the channel sr (a multiset), we generally have $s'.pos\text{-}pairs \subseteq s.pos\text{-}pairs$. (Note, that the ordering of pairs is unchanged since $used_s$ is unchanged). Also, we have $s'.buf_s = s.buf_s$.

We consider cases.

- $s.mode_r = \texttt{rec}$

  In this case the only change in the step of A_{G} is the above mentioned change of the channel sr. We show $(u, drop(I), u') \in steps(A_{\mathrm{D}})$, where I is defined below. This step (and finite execution fragment) clearly has the right trace (the empty trace).

  Define $I = \begin{cases} \emptyset & \text{if } (m, id) \notin s.pos\text{-}list \vee (m, id) \in s'.pos\text{-}list \\ \{|s.buf_r| + i\} & \text{otherwise, where } s.pos\text{-}list[i] = (m, id) \end{cases}$

  Clearly $drop(I)$ is enabled in u (elements in $pos\text{-}list$ correspond to marked elements in $u.queue$). We show that all four state variables of A_{D} are handled correctly.

  rec_s and rec_r:

  It is easy to see that we have $u'.rec_s = u.rec_s$ and $u'.rec_r = u.rec_r (= false)$ as required by the definition of $drop(I)$ in A_{D}.

  $queue$:

  We have $s'.current\text{-}queue = s.current\text{-}queue$, $s'.buf_s = s.buf_s$, and $s'.buf_r = s.buf_r$. Then the definition of I implies that $queue$ is handled as required by the definition of $drop(I)$ in A_{D}.

  $status$:

  We have from Part 3 that $u'.status = u.status$ since none of the variables occurring in Part 3 are changed in the step of A_{G}. This is allowed by $drop(I)$ provided either the value of $status$ is $(false, \texttt{OK})$ or the element at the end of $queue$ was not deleted. For conditions A, B, C(i), C(ii), and C(vi) this is obvious. For C(ii) and C(iii) we get from Invariant 8.8 Part 2 that $pos\text{-}list = \varepsilon$, so $u'.queue = u.queue$ which suffices. For C(iv) Invariant 8.8 Part 3 implies in the same way that $u'.queue = u.queue$. Finally, for C(vii) only the case where $current\text{-}ack_s = true$ is of interest. But again we get $u'.queue = u.queue$. This time because of Invariant 8.8 Part 4.

- $s.mode_r \neq \texttt{rec}$

  We consider cases based on the if-statement in the definition of $receive\_pkt_{sr}(m, id)$ in $A_{\mathrm{G},s/r}$.

– $id \in s.good_r$
  This implies $id \in s.good\text{-}ids$.
  We show that $(u, drop(I), u'', unmark(I'), u')$, where u'', I, and I' are defined below, is a finite execution fragment of A_{D}. The execution fragment clearly has the right trace (the empty trace).
  rec_s and rec_r:
    It is easy to see that we have $u'.rec_s = u.rec_s$ and $u'.rec_r = u.rec_r (\neq false)$. Define $u''.rec_s = u.rec_s$ and $u''.rec_r = u.rec_r$. Leaving rec_s and rec_r unchanged is as required by the definitions of $drop(I)$ and $unmark(I')$ in A_{D}.
  queue:
    Since $id \in s.good\text{-}ids$ we have that $(m, id) \in s.pos\text{-}pairs \cup s.current\text{-}pair$ where, by definition, $s.pos\text{-}pairs$ and $s.current\text{-}pair$ are disjoint (all *id*s are different).
    First, assume $(m, id) \in s.pos\text{-}pairs$. The effect of receiving this pair is to remove from $good_r$ (and thus from $good\text{-}ids$) all *id*s less than or equal id. This corresponds to removing an initial prefix of $s.pos\text{-}list$ up to and including (m, id). And at the same time m is moved to the end of buf_r. Invariant 8.8 Part 1 and the fact that $s.pos\text{-}pairs$ and $s.current\text{-}pair$ are disjoint gives us $s.current\text{-}queue = s'.current\text{-}queue$. Thus, $u'.queue$ can be obtained from $u.queue$ by deleting some elements corresponding to the initial prefix of $s.pos\text{-}list$ and changing the flag of the element corresponding to (m, id) to `OK` (since now this element is in buf_r). Then clearly I and I' can be defined so that the change in *queue* is as required by the definition on $drop(I)$ and $drop(I')$ in A_{D}
    If $(m, id) \in s.current\text{-}pair$ a similar argument gives us that $u'.queue$ can be obtained from $u.queue$ by deleting *all* elements corresponding to elements in $s.pos\text{-}list$ and setting the flag of the element corresponding to $s.current\text{-}queue$ to `OK`. In this case $s'.current\text{-}queue = \varepsilon$. Again, I and I' can be defined.
  status:
    If s satisfies condition A, B, or C(i) of Part 3 then so does s'. This is allowed by $drop(I)$ and $unmark(I')$ since either $u'.status = (false, \texttt{OK})$ or the element at the end of $u.queue$ was not deleted.
    If s satisfies C(ii) then s' satisfies either C(ii) or C(iii). In both cases the element end of $u.queue$ was not deleted (as required) and the possible flag change of *status* to `OK` is allowed by $unmark(I')$.
    s cannot satisfy C(iii), C(iv), or C(v) since then Invariant 8.8 Parts 2 and 3 would imply that no packets in $s.sr$ could be received successfully which contradicts the assumption that $id \in s.good_r$.
    If s satisfies C(vi) then so does s'. This is allowed by $drop(I)$ and $unmark(I')$ in A_{D}.
    Finally, assume s satisfies C(vii). Then $s.current\text{-}ack_s = false$ since we otherwise would have a contradiction with Invariant 8.8 Part 4. Thus, $u'.status = u.status = (false, \texttt{OK})$ which is allowed by $drop(I)$ and $unmark(I)$ in A_{D}.

– $id \notin s.good_r$
  Then $(u, drop(I), u') \in steps(A_{\mathrm{D}})$.
  The proof is similar to the proof in case $s.mode_r = \texttt{rec}$ above.

$\underline{a = send\_pkt_{rs}(id, b)}$

Here it is easy to see that that $u = u'$. That suffices since then the execution fragment u of A_{D} has the right properties.

$a = receive\_pkt_{rs}(id, b)$

We consider cases

- $s.mode_s = \mathtt{send} \wedge s.last_s = id$

  We show that $(u, drop(\emptyset), u'', unmark(\emptyset), u')$, where u'' is defined below, is a finite execution fragment of A_{D}. The execution fragment clearly has the right trace (the empty trace).

  Define $u''.rec_s = u.rec_s$
  $u''.rec_r = u.rec_r$
  $u''.queue = u.queue_r$

  We will define $u''.status$ below when we consider cases.

  First note that $drop(\emptyset)$ and $unmark(\emptyset)$ are enabled in u and u'', respectively, since these actions have no precondition. We show that all four state variables of A_{D} are handled correctly by the two steps in the execution fragment.

  rec_s and rec_r:

  We obviously have $u'.rec_s = u.rec_s = u''.rec_s$ and $u'.rec_r = u.rec_r = u''.rec_r$. Leaving rec_s and rec_r unchanged is as required by the definitions of $drop(\emptyset)$ and $unmark(\emptyset)$ in A_{D}.

  $queue$:

  First observe that $s'.buf_s = s.buf_s$ and $s'.buf_r = s.buf_r$. Since $(s.last_s, b) \in s.rs$, Invariant 8.10 Part 2 implies that $s.last_s \notin s.good\text{-}ids$ thus $s.current\text{-}queue = \varepsilon$. Also $s'.current\text{-}queue = \varepsilon$ since $s'.mode_s = \mathtt{idle}$. The $receive\_pkt_{rs}(id, b)$ step in A_{G} might cause $(s.current\text{-}msg_s, s.last_s)$ to be added to $pos\text{-}pairs$ (the pair might have been put onto sr but did not figure in $s.pos\text{-}pairs$ because $s.mode_s = \mathtt{send}$). $(s.current\text{-}msg_s, s.last_s)$ is, however, not added to $pos\text{-}pairs$ since $s.last_s \notin s.good\text{-}ids$ as explained above. Thus, we have $s'.pos\text{-}list = s.pos\text{-}list$. All in all we have $u'.queue = u.queue$. Leaving $queue$ unchanged is as required by the definitions of $drop(\emptyset)$ and $unmark(\emptyset)$ in A_{D}.

  $status$:

  State s cannot satisfy conditions A, C(i), and C(vii) of Part 3 because $s.mode_s = \mathtt{send}$. If s satisfies condition B then so does s. By defining $u''.status = u.status$ we have that $status$ is unchanged in the execution fragment which is allowed by the definitions of $drop(\emptyset)$ and $unmark(\emptyset)$ in A_{D}.

  State s cannot satisfy condition C(ii) since $s'.last_s \notin s.good\text{-}ids$ as explained above.

  Also, s cannot satisfy condition C(iii). If $b = true$ then Invariant 8.9 Part 3 implies $s.buf_r = \varepsilon$ which contradicts condition C(iii). If $b = false$ then Invariant 8.11 Part 2 implies $s.last_s \neq s.last_r$ which is also a contradiction.

  Assume s satisfies condition C(vi). Then $u.status = (true, \mathtt{OK})$. From the discussion in the previous condition C(iii), we have $b = true$. Now, $s.current\text{-}ack_s = b = true$ and $s'.mode_s = \mathtt{idle}$ so s' satisfies condition C(vii). Thus, also $u'.status = (true, \mathtt{OK})$. By defining $u''.status = (true, \mathtt{OK})$ we have that $status$ is unchanges in the execution fragment which is allowed by the definitions of $drop(\emptyset)$ and $unmark(\emptyset)$ in A_{D}.

  Next, assume s satisfies condition C(v). Then $u.status = (true, \mathtt{marked})$. If $b = true$ then by condition C(vii) we have $u'.status = (true, \mathtt{OK})$. This is allowed by $drop(\emptyset)$ and $unmark(\emptyset)$ by defining $u''.status = u.status$. If $b = false$ then, again, by condition C(vii) $u'.status = (false, \mathtt{OK})$ which is allowed by $drop(\emptyset)$ and $unmark(\emptyset)$ by defining $u''.status = u'.status$.

  Finally, if s satisfies C(vi) then b must be $false$ since the condition states $(s.last_s, true) \notin$

$s.rs$. Thus, $u.status = (\mathit{false}, \mathtt{OK})$ and by condition C(vii) also $u'.status = (\mathit{false}, \mathtt{OK})$. So, by defining $u''.status = u.status$, we leave *status* unchanged, which is allowed by the definition of $drop(\emptyset)$ and $unmark(\emptyset)$ in A_{D}.

- $s.mode_s \neq \mathtt{send} \vee s.last_s \neq id$

  Then the only difference between s' and s is that s' has one less copy of (m, id) in the channel rs.

  We show that $u' = u$. Then the execution fragment u clearly has the right properties. We check the state variables of A_{D}.

  rec_s, rec_r, and *queue*:

  Obviously rec_s, rec_r, and *queue* are the same in u and u'.

  status:

  No matter which condition in Part 3 s satisfies, s' satisfies the same condition, thus, $u'.status = u.status$. The only interesting case is if s satisfies condition C(v). The condition states that $s.mode_s = \mathtt{send}$, so the case hypothesis gives us that $id \neq s.last_s$. Thus, $(m, id) \neq (s.last_s, true)$. Then, since $(s.last_s, true) \in s.rs$ by condition C(v) we also have $(s'.last_s, true) \in s'.rs$. Thus, also s' satisfies condition C(v).

$a \in \{shrink\_good_s(ids), grow\_good_s(ids)\}$

Changing $good_s$ clearly does not change anything in the mapping R_{GD}. Thus, $u' = u$. Then the finite execution fragment u clearly has the right properties.

$a = shrink\_good_r(ids)$

This step removes elements from $good_r$, thus, $s'.good\text{-}ids \subseteq s.good\text{-}ids$.

We consider cases

- $s.current\text{-}ok = \mathit{false}$

  We show $(u, drop(I), u') \in steps(A_{\mathrm{D}})$, where I is defined below. Clearly the step (and finite execution fragment) has the right trace (the empty trace).

  rec_s and rec_r:

  We clearly have $u'.rec_s = u.rec_s$ and $u'.rec_r = u.rec_r$ as required by the definition of $drop(I)$ in A_{D}.

  queue:

  By shrinking *good-ids* we might remove elements from *pos-list* and *current-queue*. But, the elements in $u.queue$ corresponding to these elements are all `marked` (for *current-queue* remember that $s.current\text{-}ok = \mathit{false}$ implies $s.current\text{-}flag = \mathtt{marked}$), so by defining I to be the indices of these elements we both get that $drop(I)$ is enabled in u and that *queue* is handled correctly.

  status:

  Assume s satisfies condition A, B, or C(i) in Part 3. Then so does s', so $u'.status = u.status$. This is allowed by $drop(I)$ since in the cases (B and C(i)) where $status \neq (\mathit{false}, \mathtt{OK})$ the element at the end of $u.queue$ is not deleted.

  If s satisfies C(ii) then either s' also satisfies C(ii) which is allowed since the element at the end of $u.queue$ (which corresponds to *current-queue* is no deleted), or s' satisfies C(vi) (it cannot satisfy C(iii)–C(v) because of Invariant 8.4 Part 7 and Invariant 8.10 Part 2) which is allowed by $drop(I)$ since $s.current\text{-}ok = \mathit{false}$ implies $u.status.flag = \mathtt{marked}$.

If s satisfies C(iii)–C(v), then so does s', so $u'.status = u.status$. But this is allowed by since we in these cases have $u'.queue = u.queue$.

If s satisfies C(v) then so does s'. In this situation the element at the end of $u.queue$ might have been deleted (corresponding to elements being removed from *pos-list*, but since $status = (false, \texttt{OK})$, *status* is handled correctly.

Finally, if s satisfies C(vii) then so does s'. If $current\text{-}ack_s = false$ then $u'.status = u.status = (false, \texttt{OK})$ which is allowed by $drop(I)$. If $current\text{-}ack_s = true$ then Invariant 8.8 Part 4 implies that $u'.queue = u.queue$. Thus the element at the end of $u.queue$ is not deleted, so it is permitted to leave *status* unchanged at $(true, \texttt{OK})$.

Thus, $(u, drop(I), u') \in steps(A_{\mathrm{D}})$.

- $s.current\text{-}ok = true$

  Again we claim that $(u, drop(I), u') \in steps(A_{\mathrm{D}})$.

  The argument is similar to the previous case except that since $current\text{-}ok = true$, we have $current\text{-}flag = \texttt{OK}$, so it is not allowed to lose an element in *current-queue* or lose *status* in case C(ii). However, the precondition to $shrink\_good_r(ids)$ ensures that these requirements are met.

$a = grow\_good_r(ids)$

The precondition $ids \cap issued_r = \emptyset$ and the effect of $grow\_good_r(ids)$ ensures that $s'.good\text{-}ids = s.good\text{-}ids$.

Then it is easy to see that $u' = u$. Thus, the execution fragment u has the right properties.

$a = cleanup_r$

We show that $u' = u$. Then the execution fragment u has the right properties. We consider the four state variables of A_{D}.

rec_s, rec_r, and *queue*:

We obviously have $u'.rec_s = u.rec_s$, $u'.rec_r = u.rec_r$, and $u'.queue = u.queue$.

status:

Here the only problem would be that $last_r$ is changed. The variable $last_r$ only occurs in the conditions of Part 3 when $mode_s = \texttt{send}$, so assume $s.mode_s = \texttt{send}$. Then $s.last_s \neq s.last_r$ from the precondition. Since also $s'.mode_s = \texttt{send}$, Invariant 8.1 Part 2 gives us $s'.last_s \neq \texttt{nil}$. Now, since $s'.last_r = \texttt{nil}$, we also have $s'.last_s \neq s'.last_r$. It is now easy to see that whatever condition in Part 3 that s satisfies, s' satisfies the same condition. Thus, $u'.status = u.status$.

This concludes the simulation proof.

■

We can now prove that A_{G} safely implements A_{D}.

Theorem 8.15 (**A_{G} safely implements A_{D}**)

$A_{\mathrm{G}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{D}}$

Proof

Directly by Lemma 8.14 and the soundness of refinement mappings with respect to the safe implementation relation (Lemma 5.8).

■

8.5.3 Correctness

We do not give a formal proof that G correctly implements D. Instead we provide some intuitive justification and refer to the formal proof that H correctly implements G which is similar.

We first give two key lemmas about the live executions of G. We use our temporal logic to state the results but we only give informal proofs. These lemmas are then use to prove that G correctly implements D.

The first lemma says that if we are in a situation where no crashes occur in the future, then whenever $mode_s = \mathtt{send}$, eventually the sender will move to `idle` mode. Note, that due to *previous* crashes the sender and the receiver do not necessarily agree on what identifiers to use. So, in some situations, the sender moves to `idle` mode because of *negative* acknowledgements from the receiver, in which case the current message might have been lost.

Lemma 8.16

$L_{\mathrm{G}} \models \Box(\Box(mode_s \neq \mathtt{rec} \land mode_r \neq \mathtt{rec}) \Longrightarrow (mode_s = \mathtt{send} \leadsto mode_s = \mathtt{idle}))$

Proof

ASSUME: 1. $\alpha \in L_{\mathrm{G}}$
2. α_1 is an arbitrary suffix of α
3. $\alpha_1 \models \Box(mode_s \neq \mathtt{rec} \land mode_r \neq \mathtt{rec})$
4. α_2 is an arbitrary suffix of α_1.
5. $\alpha_2 \models mode_s = \mathtt{send}$

PROVE: $\alpha_2 \models \Diamond(mode_s = \mathtt{idle})$

We consider what happens in α_2. Note that since $mode_s = \mathtt{send}$ and no crashes occur, $mode_s$ will stay `send` unless one of the actions $\mathit{receive\_pkt}_{rs}(last_s, \mathit{true})$ or $\mathit{receive\_pkt}_{rs}(last_s, \mathit{false})$ occurs, in which case $mode_s$ changes to `idle`. Furthermore, while $mode_s = \mathtt{send}$, $last_s$ is unchanged and the sender keeps performing $\mathit{send\_pkt}_{sr}(m, last_s)$. The latter is due to weak fairness to the set $C_{\mathrm{G},s/r1}$ containing $\mathit{send\_pkt}_{sr}(m, last_s)$ since all other actions in the set are never enabled. Now, it suffices to show that eventually $\mathit{receive\_pkt}_{rs}(last_s, \mathit{true})$ or $\mathit{receive\_pkt}_{rs}(last_s, \mathit{false})$ occurs.

$\langle 1\rangle$1. CASE: $\alpha_2 \models last_s \notin \mathit{good\text{-}ids}$

$\langle 2\rangle$1. CASE: $\alpha_2 \models (last_s, \mathit{true}) \in rs$

PROOF: By the fairness of the rs channel, eventually a $\mathit{receive\_pkt}_{rs}(last_s, \mathit{true})$ action occurs. That suffices.

$\langle 2\rangle$2. CASE: $\alpha_2 \models (last_s, \mathit{true}) \notin rs \land last_s = last_r$

PROOF: In this situation the receiver has received the current packet but not yet sent positive acknowledgements.
If $buf_r \neq \varepsilon$, weak fairness to the set $C_{\mathrm{G},s/r3}$ implies that eventually $buf_r = \varepsilon$. Furthermore, buf_r stays empty as long as the sender does not leave `send` mode.
Now, when $buf_r = \varepsilon$, we have $mode_r \in \{\mathtt{idle}, \mathtt{ack}\}$. If $mode_r = \mathtt{idle}$, it changes to `ack` when a $\mathit{receive\_pkt}_{sr}(m, last_s)$ occurs. Since the sender keeps on sending $(m, last_s)$ packets, some will continue to get through (by channel liveness), so if $mode_r = \mathtt{idle}$, eventually $mode_r = \mathtt{ack}$. When $mode_r = \mathtt{ack}$ the receiver will continue to perform $\mathit{send\_pkt}_{rs}(last_r, \mathit{true})$. Such a step can, however, change $mode_r$

to `idle`, but from above we have that eventually $mode_r$ = `ack` again and new $send\_pkt_{rs}(last_r, true)$ steps will be performed.
By channel liveness, eventually $receive\_pkt_{rs}(last_r, true)$ occurs, and since $last_r = last_s$, the result follows.

$\langle 2\rangle 3$. CASE: $\alpha_2 \models (last_s, true) \notin rs \wedge last_s \neq last_r$

PROOF: This case actually describes two situations: in the first situation the current packet never has been and never can be successfully received by the receiver. In the second situation the current packet has been successfully received but the receiver crashed before placing a positive acknowledgement in the channel. Both situations are dealt with in the same way.
Every time a $receive\_pkt_{sr}(m, last_s)$ step occurs, $last_s$ is placed into $nack\text{-}buf_r$, which leads to a $send\_pkt_{rs}(last_s, false)$ action (by fairness to the $send\_pkt_{rs}(id, false)$ actions). Since $receive\_pkt_{sr}(m, last_s)$ continues to occur, $send\_pkt_{rs}(last_s, false)$ continues to occur. By channel liveness eventually $receive\_pkt_{rs}(last_s, false)$ occurs. That suffices.

$\langle 2\rangle 4$. Q.E.D.

PROOF: By the exhaustive cases $\langle 2\rangle 1$–$\langle 2\rangle 3$.

$\langle 1\rangle 2$. CASE: $\alpha_2 \models last_s \in good\text{-}ids$

PROOF: Then either always $last_s \notin good_r$ or eventually $last_s \in good_r$.
If always $last_s \notin good_r$, then the situation is as described by the case above where $\alpha_2 \models last_s \notin good\text{-}ids \wedge (last_s, true) \notin rs \wedge last_s \neq last_r$.
If eventually $last_r \in good_r$, then still the receiver might have issued $send\_pkt_{rs}(last_s, false)$ actions in the meantime, and these packets could have gotten through to the sender in which case the result follows. So, if this is not the case, eventually $(m, last_s)$ is successfully received in which case the situation is as described by the case above where $\alpha_2 \models last_s \notin good\text{-}ids \wedge (last_s, true) \notin rs \wedge last_s = last_r$.

$\langle 1\rangle 3$. Q.E.D.

PROOF: By the exhaustive cases $\langle 1\rangle 1$–$\langle 1\rangle 2$.

The result now follows from Lemma 3.5 and the definition of $\leadsto$.

■

The next lemma states that if there are elements in the four parts that make up the abstraction of a *queue* in A_D (cf. Definition 8.13), then eventually a $receive\_msg(m)$ action occurs. Thus, messages cannot be blocked in the G protocol.

Below we use the notation $receive\_msg(\_)$ to denote the set $\{receive\_msg(m) \mid m \in Msg\}$.

Lemma 8.17

$$L_G \models \Box(\Box(mode_s \neq \texttt{rec} \wedge mode_r \neq \texttt{rec} \wedge (buf_r \neq \varepsilon \vee pos\text{-}list \neq \varepsilon \vee current\text{-}queue \neq \varepsilon \vee buf_s \neq \varepsilon)) \Longrightarrow \Diamond\langle receive\_msg(\_)\rangle)$$

Proof

ASSUME: 1. $\alpha \in L_G$
2. α_1 is an arbitrary suffix of α

3. $\alpha_1 \models \Box(mode_s \neq \texttt{rec} \wedge mode_r \neq \texttt{rec} \wedge$
$(buf_r \neq \varepsilon \vee pos\text{-}list \neq \varepsilon \vee current\text{-}queue \neq \varepsilon \vee buf_s \neq \varepsilon))$

PROVE: $\alpha_1 \models \Diamond\langle receive\_msg(\_)\rangle$

$\langle 1\rangle 1.$ CASE: $\alpha_1 \models buf_r \neq \varepsilon$

PROOF: The result follows by weak fairness to the set $C_{\mathrm{G},s/r3}$.

$\langle 1\rangle 2.$ CASE: $\alpha_1 \models pos\text{-}list \neq \varepsilon$

PROOF: The packets in *pos-list* represent "old" packets in the *sr* channel that might still successfully be received by the receiver since the packets all have identifiers in *good-ids*. Due to channel liveness (the weak fairness requirement on each packet), the packets in *pos-list* will eventually be received. Two situations can occur.

First, a packet from *pos-list* is accepted because it has an identifier in $good_r$ at the time it is received. In this case the message of the packet is placed in buf_r, and $\langle 1\rangle 1$ gives the result.

Second, no packets from *pos-list* are ever accepted. Then eventually *pos-list* becomes empty (no new packets can be added to *pos-list* since no crashes occur, and each packet in *pos-list* has only finitely many copies in *sr* and these will eventually all be received (but not accepted) and thus removed from *sr*). However, then one of the other disjuncts in Part 3 of the Assumption must be satisfied, so we refer to the other cases.

$\langle 1\rangle 3.$ CASE: $\alpha_1 \models current\text{-}queue \neq \varepsilon$

$\langle 2\rangle 1.$ CASE: $\alpha_1 \models current\text{-}ok = true$

PROOF: In this situation the sender either will (because of liveness on $choose\_id(id)$ actions) or has chosen a current identifier $last_s$ which is in $good_r$ (and stays there until the current packet is accepted). The sender will send the current packet repeatedly, so by channel liveness it will eventually be received and thus accepted. The message will be placed into buf_r and Case $\langle 1\rangle 1$ gives the result.

$\langle 2\rangle 2.$ CASE: $\alpha_1 \models current\text{-}ok = false$

PROOF: Here, due to the fact that the receiver was crashed during the last *prepare* action, the sender may choose an identifier which is not in $good_r$. The sender will send the current packet repeatedly, and two things can happen.

Either, the current packet will be accepted at some point by the receiver because $last_s$ was in *good-ids* initially and has been added to $good_r$ in the meantime. Then the message is placed in buf_r and Case $\langle 1\rangle 1$ gives the result.

Or, the current packet will never be accepted by the receiver. However, since the current packet will keep on being received by the receiver (due to channel liveness), the receiver will keep on issuing negative acknowledgements for the current identifier $last_s$. By channel liveness such a negative acknowledgement will eventually get through and move the sender to `idle` mode. This has the effect of emptying *current-queue*, so one of the other disjuncts in Part 3 of the Assumption must be satisfied, so we refer to the other cases.

$\langle 2\rangle 3.$ Q.E.D.

PROOF: By exhaustive cases $\langle 2\rangle 1$ and $\langle 2\rangle 2$.

$\langle 1\rangle 4.$ CASE: $\alpha_1 \models buf_s \neq \varepsilon$

PROOF: By Fairness to the set $C_{G,s/r1}$, eventually a *prepare* action will occur. Since $mode_r \neq \texttt{rec}$, the sender ends up in `needid` mode with $current\text{-}ok = true$. The result is now implied by the first subcase of Case $\langle 1\rangle 3$.

$\langle 1\rangle 5$. Q.E.D.

PROOF: By exhaustive cases $\langle 1\rangle 1$–$\langle 1\rangle 4$.

The result now follows from Lemma 3.5.

■

With the two lemmas above we can prove the main ingredient in our liveness proofs, namely, if α is a live execution of G and α' is an execution of A_D such that $(\alpha, \alpha') \in R_{GD}$, then α' is live. We prove the result by contradiction (cf. the similar lemma (Lemma 7.17) in the proof that D correctly implements S). Thus, we assume that α' is not live and then derive a contradiction with the fact that α is live.

Lemma 8.18

Let $\alpha \in exec(A_G)$ and $\alpha' \in exec(A_D)$ be arbitrary executions of A_G and A_D, respectively, with $(\alpha, \alpha') \in R_{GD}$. Assume $\alpha \models Q_G$. Then $\alpha' \models Q_D$.

Proof

We prove the conjecture by contradiction. Thus,

ASSUME: $\alpha' \not\models Q_D$
PROVE: False

$\langle 1\rangle 1$. $\alpha' \models \neg WF(C_{D,1}, rec_s = false \wedge rec_r = false) \vee \neg WF(C_{D,2}, rec_s = false \wedge rec_r = false) \vee \neg WF(C_{D,3}) \vee \neg WF(C_{D,4})$

PROOF: Immediate by the Assumption, definition of Q_D, and the Boolean operators.

$\langle 1\rangle 2$. CASE: $\alpha' \models \neg WF(C_{D,1}, rec_s = false \wedge rec_r = false)$

$\langle 2\rangle 1$. $\alpha' \models \Diamond\Box(status.stat \in \mathsf{Bool} \wedge rec_s = false \wedge rec_r = false) \wedge \Diamond\Box\neg\langle\{ack(true), ack(false)\}\rangle$

PROOF: By Assumption $\langle 1\rangle$, the definitions of WF and $C_{D,1}$, and the fact that $ack(b)$ actions are enabled when $status.stat \in \mathsf{Bool}$.

$\langle 2\rangle 2$. $\alpha \models \Diamond\Box(mode_s \neq \texttt{rec} \wedge mode_r \neq \texttt{rec} \wedge buf_s = \varepsilon \wedge ((mode_s = \texttt{send} \wedge last_s = last_r \wedge buf_r = \varepsilon) \vee (mode_s = \texttt{send} \wedge last_s \neq last_r \wedge (last_s, true) \in rs) \vee (mode_s = \texttt{send} \wedge last_s \neq last_r \wedge (last_s, true) \notin rs \wedge last_s \notin good\text{-}ids) \vee (mode_s = \texttt{idle}))) \wedge \Diamond\Box\neg\langle\{ack(true), ack(false)\}\rangle$

PROOF: By $\langle 2\rangle 1$, Lemmas 5.10 and 5.11, the definition of R_{GD}, and the fact that $ack(b)$ actions are external.

$\langle 2\rangle 3$. $\alpha \models \Diamond\Box(mode_s = \texttt{idle} \wedge buf_s = \varepsilon) \wedge \Diamond\Box\neg\langle\{ack(true), ack(false)\}\rangle$

PROOF: By $\langle 2\rangle 2$, Lemma 8.16, and the fact that when $mode_s$ becomes $\texttt{idle}$, it stays $\texttt{idle}$ since no crashes occur and no *prepare* action can occur (since $buf_s = \varepsilon$ forever).

$\langle 2\rangle 4$. $\alpha \models \Diamond\Box(mode_s = \texttt{idle} \wedge buf_s = \varepsilon) \wedge \Diamond\Box\neg\langle C_{\mathrm{G},s/r1}\rangle$

PROOF: By $\langle 2\rangle 3$ since the $ack(b)$ actions are in $C_{\mathrm{G},s/r1}$ and no other actions in $C_{\mathrm{G},s/r1}$ can occur when $mode_s = \texttt{idle}$ and $buf_s = \varepsilon$.

$\langle 2\rangle 5$. $\alpha \models \neg\, WF(C_{\mathrm{G},s/r1})$

PROOF: By $\langle 2\rangle 4$, the definition of WF, and the fact that $mode_s = \texttt{idle} \wedge buf_s = \varepsilon$ implies the enabling condition of $C_{\mathrm{G},s/r1}$.

$\langle 2\rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 5$ contradicts the assumption that α is live.

$\langle 1\rangle 3$. CASE: $\alpha' \models \neg\, WF(C_{\mathrm{D},2}, rec_s = \mathit{false} \wedge rec_r = \mathit{false})$

$\langle 2\rangle 1$. $\alpha' \models \Diamond\Box(queue \neq \varepsilon \wedge rec_s = \mathit{false} \wedge rec_r = \mathit{false}) \wedge \Diamond\Box\neg\langle receive\_msg(\_)\rangle$

PROOF: By Assumption $\langle 1\rangle$, the definitions of WF and $C_{\mathrm{D},2}$, and the fact that $C_{\mathrm{D},2}$ is enabled when $queue \neq \varepsilon$.

$\langle 2\rangle 2$. $\alpha \models \Diamond\Box(mode_s \neq \texttt{rec} \wedge mode_r \neq \texttt{rec} \wedge$
$(buf_r \neq \varepsilon \vee pos\text{-}list \neq \varepsilon \vee current\text{-}queue \neq \varepsilon \vee buf_s \neq \varepsilon)) \wedge$
$\Diamond\Box\neg\langle receive\_msg(\_)\rangle$

PROOF: By $\langle 2\rangle 1$, Lemmas 5.10 and 5.11, the definition of R_{GD}, and the fact that $receive\_msg(m)$ actions are external.

$\langle 2\rangle 3$. Q.E.D.

PROOF: $\langle 2\rangle 2$ contradicts Lemma 8.17.

$\langle 1\rangle 4$. CASE: $\alpha' \models \neg\, WF(C_{\mathrm{D},3})$

$\langle 2\rangle 1$. $\alpha' \models \Diamond\Box(rec_s = \mathit{true}) \wedge \Diamond\Box\neg\langle recover_s\rangle$

PROOF: By expanding WF in Assumption $\langle 1\rangle$.

$\langle 2\rangle 2$. $\alpha \models \Diamond\Box(mode_s = \texttt{rec}) \wedge \Diamond\Box\neg\langle recover_s\rangle$

PROOF: By $\langle 2\rangle 1$, Lemmas 5.10 and 5.11, the definition of R_{GD}, and the fact that $recover_s$ is external.

$\langle 2\rangle 3$. $\alpha \models \Diamond\Box(mode_s = \texttt{rec}) \wedge \Diamond\Box\neg\langle C_{\mathrm{G},s/r1}\rangle$

PROOF: From $\langle 2\rangle 2$ since $recover_s \in C_{\mathrm{G},s/r1}$ and none of the other actions in $C_{\mathrm{G},s/r1}$ are enabled when $mode_s = \texttt{rec}$.

$\langle 2\rangle 4$. $\alpha \models \neg\, WF(C_{\mathrm{G},s/r1})$

PROOF: From $\langle 2\rangle 3$, the definition of WF and the fact that $mode_s = \texttt{rec}$ implies the enabling condition for $C_{\mathrm{G},s/r1}$.

$\langle 2\rangle 5$. Q.E.D.

PROOF: $\langle 2\rangle 4$ contradicts the assumption that α is live.

$\langle 1\rangle 5$. CASE: $\alpha' \models \neg\, WF(C_{\mathrm{D},4})$

PROOF: Similar to $\langle 1\rangle 4$.

$\langle 1 \rangle 6$. Q.E.D.

PROOF: By $\langle 1 \rangle 1$ and the exhaustive cases $\langle 1 \rangle 2$–$\langle 1 \rangle 5$.

■

Finally, we can show that G correctly implements D.

Theorem 8.19

$G \sqsubseteq_L D$

Proof

Immediate by Lemmas 8.14, 8.18, and 5.9.

■

We are now ready to consider the two low-level protocols: the Five-Packet Handshake Protocol H and the Clock-Based Protocol C. The next chapter deals with H and then, in Chapter 10, we consider C.

Chapter 9

The Five-Packet Handshake Protocol H

We have now reached the point where we can present the first of the low-level protocols we consider, namely, the Five-Packet Handshake Protocol of Belsnes [Bel76], which in this work is denoted by H. The H protocol is entirely distributed: it consists of a sender process, a receiver process, and two channels as depicted in Figure 9.1.

H is the standard protocol for setting up network connections, used in TCP, ISO TP-4, and many other transport protocols. During normal operation it goes through three phases (cf. Figure 9.2):

Agree on identifier: The sender picks an identifier, called jd to distinguish it from the identifier id used below for the actual communication of the message, and sends it in a `needid` packet. On receipt of this packet, the receiver pairs jd with a new identifier id, and sends the pair (jd, id) back to the sender. On receipt of this pair, the sender knows that it should associate id to the current message.

Send and acknowledge: This phase is similar to the send/acknowledge phase of G. The sender sends the current packet in `send` packets, and the receiver acknowledges the receipt with `ack` packets.

Clean up: When the sender has received the acknowledgement, it issues a `done` packet in order to inform the receiver that it may forget about the last message accepted.

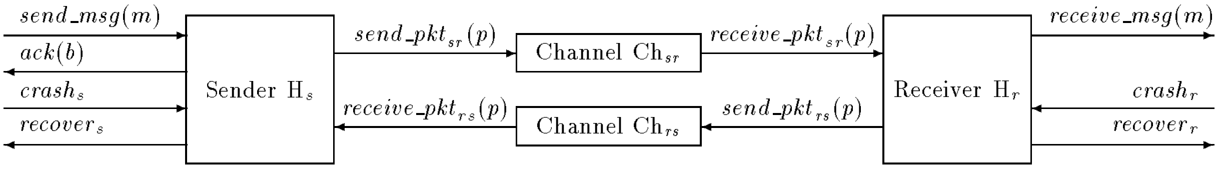


Figure 9.1

The Five-Packet Handshake Protocol H.

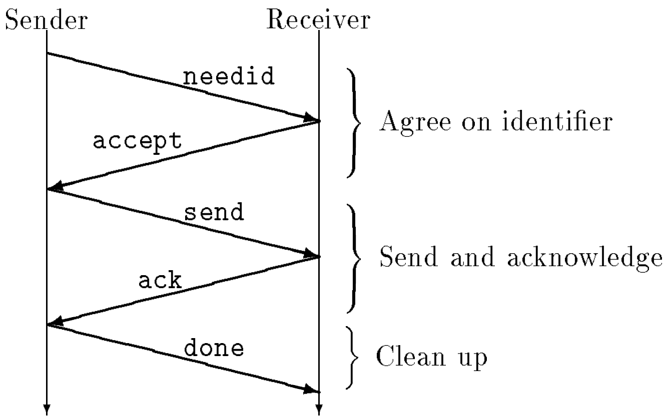


Figure 9.2

The phases of H.

Below we look at different abnormal situations which can arise due to crashes. H is sometimes called the three-way handshake, because only three packet types are needed for message delivery (the first three in Figure 9.2).

The rest of this chapter is organized as follows. Section 9.1 considers the channels in H. Then, in Section 9.2, we present the sender and receiver processes, and in Section 9.3 we show how H is obtained from the subprocesses. Finally, in Section 9.4 we prove that H correctly implements G.

9.1 The Channels

We use the same channels as at the G level (cf. Section 8.2). However, the actual packets that are communicated are different in H and G. This only means that in H we should instantiate the set P of possible packets with a different set of packets than in G.

9.2 The Sender and the Receiver

In this section we specify the sender and receiver processes as two live I/O automata $\mathrm{H}_s = (A_{\mathrm{H},s}, L_{\mathrm{H},s})$ and $\mathrm{H}_r = (A_{\mathrm{H},r}, L_{\mathrm{H},r})$, respectively. In the subsection defining $steps(A_{\mathrm{H},s})$ and $steps(A_{\mathrm{H},r})$ below, we provide more intuition about the H protocol.

9.2.1 States and Start States

The sender and receiver processes both contain a *stable* set of used identifiers. This means that these sets should survive crashes when implemented on a physical machine. Specifically, we model the stability of a state variable by not resetting it on recovery.

For instance, the stable set $issued_r$ includes all identifiers ever considered "good" by the receiver. Thus, every time the receiver issues a new identifier id (to be sent to the sender in an `accept` packet) this should be remembered forever by adding id to $issued_r$. This is an expensive solution since it requires updates to a stable variable for every message. The fix to this problem would be to introduce a normal volatile (i.e., non-stable) variable $unused_r$ which is filled with new (i.e., non-$issued_r$) identifiers now and then in steps that update the stable variable $issued_r$ by adding these new identifiers. Then, for each message, the identifier can be chosen from $unused_r$ and no updates to stable variables need to be performed. Of course, $unused_r$ will be lost in crashes, so it should not be kept too big, but on the other side, the less identifiers it contains, the more frequently updates to the stable variable $issued_r$ needs to be performed. This is a typical trade-off.

We do not consider the addition of the variable $unused_r$ to H_r, but the changes needed are both few and simple.

Sender

The sender chooses identifiers jd from the set JD. This set is similar to the set ID introduced in Section 8.1. We call it JD to distinguish it from ID, which are identifiers chosen by the receiver.

Variable		Type	Initially	Description
$mode_s$		{`idle`, `needid`, `send`, `rec`}	`idle`	The mode of the sender. Similar to the mode of the sender at the G level.
buf_s		Msg^*	ε	The list of messages at the sender side.
jd_s		$JD \cup \{$`nil`$\}$	`nil`	The jd chosen for the current message by the sender.
$jd\text{-}used_s$	S	$\mathcal{P}(JD)$	$\emptyset$	A set including all the jds ever used by the sender.
id_s		$ID \cup \{$`nil`$\}$	`nil`	The id received from the receiver. Similar to $last_s$ at the G level.
$current\text{-}msg_s$		$Msg \cup \{$`nil`$\}$	`nil`	The message about to be sent to the receiver. Same as at the G level.
$current\text{-}ack_s$		Bool	*false*	Acknowledgement from the receiver. Same as at the G level.
$done\text{-}buf_s$		ID^*	ε	A list of ids for which the sender must issue an `done` packet to the receiver.

S = Stable

Receiver

Variable		Type	Initially	Description
$mode_r$		{`idle`, `accept`, `rcvd`, `ack`, `rec`}	`idle`	The mode of the receiver. Similar to the receiver mode at the G level, except for the extra `accept` mode. In mode `accept` the receiver is sending `accept` packets, which contain the chosen message identifier.
buf_r		Msg^*	ε	The list of messages accepted. Same as at the G level.
jd_r		$JD \cup \{\mathtt{nil}\}$	`nil`	The jd received from the receiver.
id_r		$ID \cup \{\mathtt{nil}\}$	`nil`	The id chosen for the received jd.
$last_r$		$ID \cup \{\mathtt{nil}\}$	`nil`	This variable contains (when non-`nil`) the id of the last packet accepted.
$issued_r$	S	$\mathcal{P}(ID)$	ε	A set including all ids ever issued by the receiver. Same as at the G level.
$nack\text{-}buf_r$		ID^*	ε	A list of ids for which the receiver will issue negative acknowledgements. Same as at the G level.

S = Stable

9.2.2 Actions

Sender

Input:
- $send\_msg(m)$, $m \in Msg$
- $crash_s$
- $receive\_pkt_{rs}(\mathtt{accept}, jd, id)$, $jd \in JD$, $id \in ID$
- $receive\_pkt_{rs}(\mathtt{ack}, id, b)$, $id \in ID$, $b \in \mathsf{Bool}$

Output:
- $ack(b)$, $b \in \mathsf{Bool}$
- $recover_s$
- $send\_pkt_{sr}(\mathtt{needid}, jd)$, $jd \in JD$
- $send\_pkt_{sr}(\mathtt{send}, m, id)$, $m \in Msg$, $id \in ID$
- $send\_pkt_{sr}(\mathtt{done}, id)$, $id \in ID$

Internal:
- $choose\_jd(jd)$, $jd \in JD$
- $grow\text{-}jd\text{-}used_s(jds)$, $jds \in \mathcal{P}(JD)$

Receiver

Input:
- $crash_r$
- $receive\_pkt_{sr}(\mathtt{needid}, jd)$, $jd \in JD$
- $receive\_pkt_{sr}(\mathtt{send}, m, id)$, $m \in Msg$, $id \in ID$
- $receive\_pkt_{sr}(\mathtt{done}, id)$, $id \in ID$

Output:
- $receive\_msg(m)$, $m \in Msg$
- $recover_r$
- $send\_pkt_{rs}(\mathtt{accept}, jd, id)$, $jd \in JD$, $id \in ID$

$send\_pkt_{rs}(\mathsf{ack}, id, b),\ id \in ID,\ b \in \mathsf{Bool}$

Internal:

$grow\text{-}issued_r(ids),\ ids \in \mathcal{P}(ID)$

9.2.3 Steps

We now formally define $steps(A_{\mathrm{H},s})$ and $steps(A_{\mathrm{H},r})$. As at the G level we increase readability by listing the definition of $steps(A_{\mathrm{H},s})$ in the left column and the definition of $steps(A_{\mathrm{H},r})$ in the right, and by aligning *send-pkt* with the corresponding *receive-pkt*.

After the definition, we provide more intuition about how H works.

$send\_msg(m)$
Effect:
  if $mode_s \neq \mathtt{rec}$ then
    $buf_s := buf_s \,\hat{}\, m$

$choose\_jd(jd)$
Precondition:
  $mode_s = \mathtt{idle} \wedge buf_s \neq \varepsilon \wedge$
  $jd \notin jd\text{-}used_s$
Effect:
  $mode_s := \mathtt{needid}$
  $jd_s := jd$
  $jd\text{-}used_s := jd\text{-}used_s \cup \{jd\}$
  $current\text{-}msg_s := head(buf_s)$
  $buf_s := tail(buf_s)$

$send\_pkt_{sr}(\mathtt{needid}, jd)$
Precondition:
  $mode_s = \mathtt{needid} \wedge jd_s = jd$
Effect:
  none

$receive\_pkt_{sr}(\mathtt{needid}, jd)$
Effect:
  if $mode_r = \mathtt{idle}$ then
    $mode_r := \mathtt{accept}$
    choose an id not in $issued_r$
    $jd_r := jd$
    $id_r := id$
    $issued_r := issued_r \cup \{id\}$

$receive\_pkt_{rs}(\mathtt{accept}, jd, id)$
Effect:
  if $mode_s \neq \mathtt{rec}$ then
    if $mode_s = \mathtt{needid} \wedge jd_s = jd$ then
      $mode_s := \mathtt{send}$
      $id_s := id$
    else if $id_s \neq id$ then
      $done\text{-}buf_s := done\text{-}buf_s \,\hat{}\, id$

$send\_pkt_{rs}(\mathtt{accept}, jd, id)$
Precondition:
  $mode_r = \mathtt{accept} \wedge jd_r = jd \wedge id_r = id$
Effect:
  none

$send\_pkt_{sr}(\mathtt{send}, m, id)$
Precondition:
  $mode_s = \mathtt{send} \wedge current\text{-}msg_s = m \wedge id_s = id$
Effect:
  none

$receive\_pkt_{sr}(\mathtt{send}, m, id)$
Effect:
  if $mode_r \neq \mathtt{rec}$ then
    if $mode_r = \mathtt{accept} \wedge id_r = id$ then
      $mode_r := \mathtt{rcvd}$
      $buf_r := buf_r \,\hat{}\, m$
      $last_r := id$
    else if $last_r \neq id$ then
      $nack\text{-}buf_r := nack\text{-}buf_r \,\hat{}\, id$

$receive\_msg(m)$
Precondition:
$mode_r = \mathtt{rcvd} \wedge buf_r \neq \varepsilon \wedge$
$head(buf_r) = m$
Effect:
$buf_r := tail(buf_r)$
if $buf_r = \varepsilon$ then
$mode_r := \mathtt{ack}$

$receive\_pkt_{rs}(\mathsf{ack}, id, b)$
Effect:
if $mode_s \neq \mathtt{rec}$ then
if $mode_s = \mathtt{send} \wedge id_s = id$ then
$mode_s := \mathtt{idle}$
$current\text{-}ack_s := b$
$jd_s := \mathtt{nil}$
$id_s := \mathtt{nil}$
$current\text{-}msg_s := \mathtt{nil}$
if $b = true$ then
$done\text{-}buf_s := done\text{-}buf_s \,\hat{}\, id$

$send\_pkt_{rs}(\mathsf{ack}, id, true)$
Precondition:
$mode_r = \mathtt{ack} \wedge last_r = id$
Effect:
none

$send\_pkt_{rs}(\mathsf{ack}, id, false)$
Precondition:
$mode_r \neq \mathtt{rec} \wedge nack\text{-}buf_r \neq \varepsilon \wedge$
$head(nack\text{-}buf_r) = id$
Effect:
$nack\text{-}buf_r := tail(nack\text{-}buf_r)$

$send\_pkt_{sr}(\mathsf{done}, id)$
Precondition:
$mode_s \neq \mathtt{rec} \wedge done\text{-}buf_s \neq \varepsilon \wedge$
$head(done\text{-}buf_s) = id$
Effect:
$done\text{-}buf_s := tail(done\text{-}buf_s)$

$receive\_pkt_{sr}(\mathsf{done}, id)$
Effect:
if $(mode_r = \mathtt{accept} \wedge id_r = id) \vee$
$(mode_r = \mathtt{ack} \wedge last_r = id)$ then
$mode_r := \mathtt{idle}$
$jd_r := \mathtt{nil}$
$id_r := \mathtt{nil}$
$last_r := \mathtt{nil}$

$ack(b)$
Precondition:
$mode_s = \mathtt{idle} \wedge buf_s = \varepsilon \wedge$
$current\text{-}ack_s = b$
Effect:
none

$crash_s$
Effect:
$mode_s := \mathtt{rec}$

$crash_r$
Effect:
$mode_r := \mathtt{rec}$

$recover_s$
Precondition:
$mode_s = \mathtt{rec}$
Effect:
$mode_s := \mathtt{idle}$
$jd_s := \mathtt{nil}$
$id_s := \mathtt{nil}$
$buf_s := \varepsilon$
$current\text{-}msg_s := \mathtt{nil}$
$current\text{-}ack_s := false$
$done\text{-}buf_s := \varepsilon$

$recover_r$
Precondition:
$mode_r = \mathtt{rec}$
Effect:
$mode_r := \mathtt{idle}$
$jd_r := \mathtt{nil}$
$id_r := \mathtt{nil}$
$last_r := \mathtt{nil}$
$buf_r := \varepsilon$
$nack\text{-}buf_r := \varepsilon$

$grow\text{-}jd\text{-}used_s(jds)$
Precondition:
$|JD \setminus (jd\text{-}used_s \cup jds)| = \infty$
Effect:
$jd\text{-}used_s := jd\text{-}used_s \cup jds$

$grow\text{-}issued_r(ids)$
Precondition:
$|ID \setminus (issued_r \cup ids)| = \infty$
Effect:
$issued_r := issued_r \cup ids$

The following note about the $receive\_pkt_{sr}(\mathtt{needid}, jd)$ steps should be made: $A_{\mathrm{H},r}$ is required to be input-enabled and therefore we do not specify preconditions for input actions. However, in the effect clause of $receive\_pkt_{sr}(\mathtt{needid}, jd)$ we must choose an id not in $issued_r$. But this is only possible if $issued_r \neq ID$. However, Invariant 9.11 Part 8.12 below states that this is indeed the case for all reachable states. However, since there exists (non-reachable) states with $issued_r = ID$, $A_{\mathrm{H},r}$ is not input-enabled. This is not a problem in practice, but to make $A_{\mathrm{H},r}$ input-enabled we interpret the definition of $receive\_pkt_{sr}(\mathtt{needid}, jd)$ such that an arbitrary id is chosen if $issued_r = ID$.

We first describe the normal mode of operation: the sender performs a $choose\_jd(jd)$ action (which corresponds to *prepare* of G) and moves to mode `needid`, where it repeatedly sends $(\mathtt{needid}, jd)$ to the receiver. By channel liveness these packets will continue to get through. One of the major problems in the liveness proof below is to show that eventually the receiver will be in `idle` mode. When this happens, the receiver accepts $(\mathtt{needid}, jd)$, associates a new identifier id with jd, and moves to `accept` mode, where it repeatedly issues $(\mathtt{accept}, jd, id)$ packets. Again by channel liveness, such a packet gets through and since jd is equal to the current jd (kept in jd_s) of the sender, the sender accepts this packet. The value jd is no longer needed, but id is used for the actual communication.

On receipt of $(\mathtt{accept}, jd, id)$ the sender moves to mode `send`. Note how the `accept` packets work as acknowledgements for the `needid` packets. In `send` mode the sender repeatedly sends the current packet $(\mathtt{send}, m, id)$. When one gets through, it is accepted since the id in the packet corresponds to the current id (kept in id_r) of the receiver. The message m is placed in buf_r and the identifier id for which the receiver shall eventually issue positive acknowledgements is remembered in the $last_r$ variable. (Note the difference between id_r and $last_r$: id_r remembers the identifier that the receiver will accept, whereas $last_r$ remembers the identifier for which the receiver must issue positive acknowledgements. Due to this difference the identifiers are kept in separate variables.) Now, eventually m is delivered to the user and the receiver moves to `ack` mode. Note how the `send` packets work as acknowledgements for the `accept` packets.

In `ack` mode the receiver repeatedly sends positive acknowledgements in $(\mathtt{ack}, id, true)$ packets. When one gets through, the sender leaves `send` mode and issues a positive acknowledgement $ack(b)$ to the user at the sender side.

The receiver has no knowledge of whether an $(\mathtt{ack}, id, true)$ packet has gotten through yet or not, so it continues to issue the packets. Somehow the receiver must be informed that the sender has received the acknowledgement. The `done` packets are used for this purpose. It would not work if the sender entered a mode where it repeatedly issued `done` packets because then the receiver would have to acknowledge the receipt of a `done` packet, and so on. Instead, every time the sender receives $(\mathtt{ack}, id, true)$ it adds id to $done\text{-}buf_s$, and this leads to one $send\_pkt_{sr}(\mathtt{done}, id)$ being issued. There is no guarantee that the packet is not lost, but if it is, the sender will eventually receive another $(\mathtt{ack}, id, true)$ packet, which gives rise to another $send\_pkt_{sr}(\mathtt{done}, id)$ step. This cannot go on forever because of channel liveness, so eventually the receiver will receive $(\mathtt{done}, id)$ and since id is equal to $last_r$, the receiver leaves `ack` mode and moves to `idle` mode, where it is allowed to forget everything about jd_r, id_r, and $last_r$.

The above discussion has concentrated on normal mode of operation, where the sender and receiver are synchronized. However, because both the sender and the receiver have modes where they repeatedly send certain packets and await acknowledgements, they would be very vulnerable to crashes of the other node if we did not have some means of informing the node about crashes. The "bad" modes are `accept` for the receiver and `send` for the sender.

First consider a situation where the receiver is in `accept` mode but where the sender due to crashes is not in the expected `needid` mode with $jd_s = jd_r$. The sender could be in `idle` mode or even in `needid` mode with a new jd identifier such that $jd_s \neq jd_r$. Now, every time the sender receives a bad `accept` packet, it places the associated identifier id in $done\text{-}buf_s$ which leads to a $send\_pkt_{sr}(\mathtt{done}, id)$ step, which may or may not succeed in putting the packet into the channel. If it succeeds, the packet will eventually be received and the receiver will be dislodged (cf. the definition of the $receive\_pkt_{sr}(\mathtt{done}, id)$ steps of the receiver). If it does not succeed, the sender will eventually receive another `accept` packet, which gives rise to another $send\_pkt_{sr}(\mathtt{done}, id)$ step. This cannot go on forever because of channel liveness, so eventually the receiver will receive $(\mathtt{done}, id)$. Thus, the `done` packets are used to inform the receiver to leave a bad `accept` mode in the same way `done` packets were used during normal mode of operation to inform the receiver that the sender has received the positive acknowledgement. An additional problem arises because the receiver immediately could receive an old `needid` packet and thus reenter a bad `accept` mode. However, there can only be finitely many such old `needid` packets in the channel, so this cannot go on forever. Below we shall see how this is proved formally.

Another "bad" situation occurs when the sender is in `send` mode but where the receiver due to crashes is not in the expected `accept` mode with $id_r = id_s$. The receiver could be in `idle` mode or it could have received an old `needid` packet and thus be in `accept` mode with $id_r \neq id_s$. Now, every time the receiver receives a $(\mathtt{send}, m, id)$ packet it will, since $id \neq id_r$, add id to $nack\text{-}buf_r$, which leads to $send\_pkt_{rs}(\mathtt{ack}, id, \mathit{false})$. This continues, as for the `done` packets above, until $(\mathtt{ack}, id, \mathit{false})$ is receiver by the sender and at that point the sender resets to `idle` mode.

The actions $grow\text{-}jd\text{-}used_s(jds)$ and $grow\text{-}issued_r(ids)$ allow identifiers to be added to the sets of used identifiers of the sender and receiver, respectively, as long as there are still "enough" (i.e., infinitely many) unused identifiers left. These actions are not required for the correctness of H but allow a final implementation on a physical machine to throw away some identifiers. This is typically required by algorithms for generating unused identifiers.

It may seem strange that the sender and receiver need to engage in the initial `needid/accept` handshake. Why don't they just agree on using, say, the natural numbers in increasing order as identifiers? Then the receiver will only accept a message if the associated identifier is greater than the identifier of the last message accepted. The answer is that H is designed so that the receiver can use the same set of identifiers for *several* senders. Thus, as defined, the sender does not have to remember (in stable storage!) the last identifier used by each individual sender. We do not in this report show how the receiver should work for several senders.

The discussion above has partly been based on liveness assumptions on the sender and receiver. We now consider how to specify this liveness formally.

9.2.4 Liveness

Sender

We define the following two sets of the locally-controlled actions of the sender:

$$\begin{aligned} C_{\mathrm{H},s1} \;\triangleq\;\; & \{ack(true), ack(\mathit{false}), recover_s\} \;\cup \\ & \{send\_pkt_{sr}(\mathtt{needid}, jd) \mid jd \in JD\} \;\cup \end{aligned}$$

$$\{send\_pkt_{sr}(\texttt{send}, m, id) \mid m \in Msg \wedge id \in ID\}$$
$$C_{\mathrm{H},s2} \triangleq \{send\_pkt_{sr}(\texttt{done}, id) \mid id \in ID\}$$

The liveness formula $Q_{\mathrm{H},s}$ that induces the liveness condition $L_{\mathrm{H},s}$ for $A_{\mathrm{H},s}$ is now defined as

$$Q_{\mathrm{H},s} \triangleq WF(C_{\mathrm{H},s1}) \wedge WF(C_{\mathrm{H},s2})$$

Note, that the reason we need weak fairness to $C_{\mathrm{H},s2}$ separately is that sending of `done` packets can occur at any time. Then, if we only had weak fairness to $C_{\mathrm{H},s1} \cup C_{\mathrm{H},s2}$, there would be no requirement to issue `done` packets if the sender is in `send` mode and keeps sending `send` packets. This would not lead to correct operation of H.

Thus, H_s can intuitively be seen as consisting of two parallel processes: one dealing with the actions in $C_{\mathrm{H},s1}$ and one dealing with issuing `done` packets. Since the liveness requirements are weak fairness, the liveness of H_s can be implemented on a physical machine by a scheduler giving fair turns to the two parallel processes.

By Lemma 4.7, $Q_{\mathrm{H},s}$ is an environment-free liveness formula for $A_{\mathrm{H},s}$. Thus, H_s is a live I/O automaton. Furthermore, by Lemma 4.8, $Q_{\mathrm{H},s}$ is stuttering-insensitive.

Receiver

We define the following two sets of locally-controlled actions of the receiver:

$$\begin{aligned} C_{\mathrm{H},r1} &\triangleq \{recover_r\} \cup \\ &\quad \{receive\_msg(m) \mid m \in Msg\} \cup \\ &\quad \{send\_pkt_{rs}(\texttt{accept}, jd, id) \mid jd \in JD \wedge id \in ID\} \cup \\ &\quad \{send\_pkt_{rs}(\texttt{ack}, id, true) \mid id \in ID\} \\ C_{\mathrm{H},r2} &\triangleq \{send\_pkt_{sr}(\texttt{ack}, id, false) \mid id \in ID\} \end{aligned}$$

The liveness formula that induces the liveness condition for the receiver of H can now be expressed as

$$Q_{\mathrm{H},r} \triangleq WF(C_{\mathrm{H},r1}) \wedge WF(C_{\mathrm{H},r2})$$

The reason why we need weak fairness to *two* sets of actions is similar to the reason given above for the sender.

By Lemma 4.7, $Q_{\mathrm{H},r}$ is an environment-free liveness formula for $A_{\mathrm{H},r}$. Thus, H_r is a live I/O automaton. Furthermore, by Lemma 4.8, $Q_{\mathrm{H},r}$ is stuttering-insensitive.

9.3 The Specification of H

As depicted in Figure 9.1, H consists of the sender and receiver processes and the two channels. So, first define $\mathrm{H}'' = (A''_{\mathrm{H}}, L''_{\mathrm{H}})$ to be the following live I/O automaton.

$$\mathrm{H}'' \triangleq \mathrm{H}_s \| \mathrm{H}_r \| \mathrm{Ch}_{sr} \| \mathrm{Ch}_{rs}$$

Since $Q_{\mathrm{H},s}$, $Q_{\mathrm{H},r}$, $Q_{\mathrm{Ch},sr}$, and $Q_{\mathrm{Ch},rs}$ are all stuttering-insensitive, Proposition 4.4 implies that L''_{H} is induced by

$$Q_{\mathrm{H}} \triangleq Q_{\mathrm{H},s} \wedge Q_{\mathrm{H},r} \wedge Q_{\mathrm{Ch},sr} \wedge Q_{\mathrm{Ch},rs}$$

By Definition 2.2 the channel actions $send\_pkt_{sr}(\ldots)$, $receive\_pkt_{sr}(\ldots)$, $send\_pkt_{rs}(\ldots)$, and $receive\_pkt_{rs}(\ldots)$ are all output actions of H″. We need to hide these in order to get a live I/O automaton with the same external actions as S.

However, recall from Lemma 5.10 that the existence of an index mapping between executions at two levels of abstraction allows one to conclude certain properties of the (common) *external* actions of the executions. Thus, the more external actions of two levels, the stronger the correspondence between the executions.

At the G level we defined G′ to be the system where channel communication is external, i.e., G′ was simply the parallel composition of the sender/receiver process and the channels—similar to H″ above. Now, the actions $send\_pkt_{sr}(m, id)$, $receive\_pkt_{sr}(m, id)$, $send\_pkt_{rs}(id, b)$, and $receive\_pkt_{rs}(id, b)$ of G′ correspond to the $send\_pkt_{sr}(\texttt{send}, m, id)$, $receive\_pkt_{sr}(\texttt{send}, m, id)$, $send\_pkt_{rs}(\texttt{ack}, id, b)$, and $receive\_pkt_{rs}(\texttt{ack}, id, b)$ actions at the H level. Thus, the channel actions at the H level which deal with `needid`, `accept`, and `done` packets do not correspond to any external actions of G′. Thus, we first hide these actions from H″ to get H′. Let

$$
\begin{aligned}
\mathcal{A}'_{\mathrm{H}} \;\triangleq\; & \{send\_pkt_{sr}(\texttt{needid}, id) \mid id \in ID\} \cup \\
& \{receive\_pkt_{sr}(\texttt{needid}, id) \mid id \in ID\} \cup \\
& \{send\_pkt_{rs}(\texttt{accept}, jd, id) \mid jd \in JD \wedge id \in ID\} \cup \\
& \{receive\_pkt_{rs}(\texttt{accept}, jd, id) \mid jd \in JD \wedge id \in ID\} \cup \\
& \{send\_pkt_{sr}(\texttt{done}, id) \mid id \in ID\} \cup \\
& \{receive\_pkt_{sr}(\texttt{done}, id) \mid id \in ID\}
\end{aligned}
$$

Then $\mathrm{H}' = (A'_{\mathrm{H}}, L'_{\mathrm{H}})$ is defined as

$$\mathrm{H}' \;\triangleq\; \mathrm{H}'' \setminus \mathcal{A}'_{\mathrm{H}}$$

By Proposition 4.5, L'_{H} is induced by Q_{H}.

Finally, to get the H protocol, we hide the remaining channel actions. Let

$$
\begin{aligned}
\mathcal{A}_{\mathrm{H}} \;\triangleq\; & \{send\_pkt_{sr}(\texttt{send}, m, id) \mid m \in Msg \wedge id \in ID\} \cup \\
& \{receive\_pkt_{sr}(\texttt{send}, m, id) \mid m \in Msg \wedge id \in ID\} \cup \\
& \{send\_pkt_{rs}(\texttt{ack}, id, b) \mid id \in ID \wedge b \in \mathsf{Bool}\} \cup \\
& \{receive\_pkt_{rs}(\texttt{ack}, id, b) \mid id \in ID \wedge b \in \mathsf{Bool}\}
\end{aligned}
$$

Thus, $\mathrm{H} = (A_{\mathrm{H}}, L_{\mathrm{H}})$ is defined as

$$\mathrm{H} \;\triangleq\; \mathrm{H}' \setminus \mathcal{A}_{\mathrm{H}}$$

Again, by Proposition 4.5, L_{H} is induced by Q_{H}.

Now, in the proof below we prove that H′ correctly implements G′ (or actually a slightly different version of G′ in which the channel actions are renamed to completely match the (remaining) external channel actions of H′). Then the substitutivity results of Proposition 2.16 are used to infer that H correctly implements G.

9.4 Correctness of H

The correctness of H with respect to G is now considered. We first add history variables to H′ to get $\mathrm{H}^{h\,\prime} = (A^{h\,\prime}_{\mathrm{H}}, L^{h\,\prime}_{\mathrm{H}})$ as described in Section 5.1.5. Then we state some invariants of $A^{h\,\prime}_{\mathrm{H}}$ and show the existence of a refinement mapping from $A^{h\,\prime}_{\mathrm{H}}$ to $A^{\rho\,\prime}_{\mathrm{G}}$, where $A^{\rho\,\prime}_{\mathrm{G}}$ is a slightly modified version of A'_{G} obtained by renaming some channel actions. This refinement mapping is then used to show that $\mathrm{H}^{h'}$ correctly implements $\mathrm{G}^{\rho'}$, which, in turn, allows us to conclude that H correctly implements G.

9.4.1 Adding History Variables to H$'$

We add three history variables to H$'$ and denote the resulting live I/O automaton by $\mathrm{H}^{h'} = (A_{\mathrm{H}}^{h'}, L_{\mathrm{H}}^{h'})$.

Variable		**Type**	**Initially**	**Description**
$used_s$	H	ID^*	ε	A history variable giving the list of *id*s ever used by the sender (and thus accepted in `accept` packets from the receiver). Same as at the G level.
$seen_r$	H	$\mathcal{P}(JD \times ID)$	$\emptyset$	A history variable consisting of all the (jd, id) pairs the receiver has ever seen.
current-ok	H	Bool	*false*	A history variable describing the state of the current message. Same as at the G level.
H = History				

By the results in Section 5.1.5, we are allowed to change the history variables anywhere in the effect clauses of the step rules defining the steps of A'_{H}. The effect clauses of step rules of A'_{H} are, in turn, defined by the corresponding effect clauses of the components of H$'$ as described in Section 4.1.1.1. We show where the changes to the history variables should be placed in the effect clauses. We omit the assignments to the original variables (by writing ... instead) but outline the if-then-else statements.

choose_jd(jd)
  Precondition:
    (∗ Precondition from H_s ∗)
    ...
  Effect:
    (∗ Effect clause from H_s ∗)
    ...
    if $mode_r \neq$ `rec` then
      current-ok := *true*

$receive\_pkt_{sr}$(`needid`, jd)
  Precondition:
    (∗ Precondition from Ch_{sr} ∗)
    ...
  Effect:
    (∗ Effect clause from Ch_{sr} ∗)
    ...
    (∗ Effect clause from H_r ∗)
    if $mode_s$ = `idle` then
      ...
      $seen_r := seen_r \cup \{(jd_r, id_r)\}$

$receive\_pkt_{rs}(\texttt{accept}, jd, id)$
  Precondition:
    (* Precondition from Ch_{rs} *)
    ...
  Effect:
    (* Effect clause from Ch_{rs} *)
    ...
    (* Effect clause from H_s *)
    if $mode_s \neq \texttt{rec}$ then
      if $mode_s = \texttt{needid} \wedge jd_s = jd$ then
        ...
        $used_s := used_s \,\hat{}\, id$
      else if $id_s \neq id$ then
        ...

$receive\_pkt_{sr}(\texttt{send}, m, id)$
  Precondition:
    (* Precondition from Ch_{sr} *)
    ...
  Effect:
    (* Effect clause from Ch_{sr} *)
    ...
    (* Effect clause from H_s *)
    if $mode_r \neq \texttt{rec}$ then
      if $mode_r = \texttt{accept} \wedge id_r = id$ then
        ...
        if $id = id_s$ then
          $current\text{-}ok := false$
      else if $last_r \neq id$ then
        ...

$crash_s$
  Effect:
    (* Effect clause from H_s *)
    ...
    $current\text{-}ok := false$

$crash_r$
  Effect:
    (* Effect clause from H_r *)
    ...
    $current\text{-}ok := false$

From Lemma 5.16 we know that $L_{\mathrm{H}}^{h\,\prime}$ is induced by Q_{H}.

9.4.2 Invariants

To help us in the refinement mapping proof below, we state some invariants of $A_{\mathrm{H}}^{h\,\prime}$ without proofs. The proofs could be performed similarly to the proofs of the A_{G} invariants in Appendix C.

The first invariant states properties of $issued_r$.

Invariant 9.1

1. If $id_r \neq nil$ then $id_r \in issued_r$
2. If $last_r \neq nil$ then $last_r \in issued_r$
3. If $(\texttt{accept}, jd, id) \in rs$ then $id \in issued_r$
4. $used_s \subseteq issued_r$

■

Define in any state of $A_{\mathrm{H}}^{h}{}'$ $jds(sr)$ to be the set of jd components of the packets in the sr channel. Formally, since only `needid` packets have jd components in the sr channel, we have

$$jds(sr) = \{jd \mid (\mathtt{needid}, jd) \in sr\}$$

Similarly,

$$jds(rs) = \{jd \mid (\mathtt{accept}, jd, id) \in rs\}$$

The following invariant then states that all jds in the system are used by the sender.

Invariant 9.2

1. $jd_s \in jd\text{-}used_s$ if $jd_s \neq \mathtt{nil}$
2. $jds(sr) \subseteq jd\text{-}used_s$
3. $jd_r \in jd\text{-}used_s$ if $jd_r \neq \mathtt{nil}$
4. $jds(rs) \subseteq jd\text{-}used_s$

■

The following invariants state simple properties.

Invariant 9.3

1. If $mode_r \in \{\mathtt{idle}, \mathtt{accept}\}$ then $last_r = \mathtt{nil}$

■

Invariant 9.4

1. If $mode_r = \mathtt{accept}$ then $id_r \neq \mathtt{nil}$

■

Invariant 9.5

1. If $mode_s = \mathtt{rec} \vee mode_r = \mathtt{rec}$ then $current\text{-}ok = false$

■

Invariant 9.6

1. If $id_s \neq \mathtt{nil}$ then $mode_s \in \{\mathtt{send}, \mathtt{rec}\}$

■

The next invariant states the identifiers in the system are in most cases registered in the history variable $used_s$.

Invariant 9.7

1. If $id_s \neq \mathtt{nil}$ then $id_s \in used_s$
2. If $(\mathtt{send}, m, id) \in sr$ then $id \in used_s$

3. If $mode_r = \texttt{rcvd}$ then $last_r \in used_s$
4. If $mode_r = \texttt{ack}$ then $last_r \in used_s$
5. If $(\texttt{ack}, id, b) \in rs$ then $id \in used_s$

■

The identifiers for which the sender will issue or has issued `done` packets can never be equal to the current identifier of the sender.

Invariant 9.8

1. If $id \in done\text{-}buf_s$ then $id \neq id_s$
2. If $(\texttt{done}, id) \in sr$ then $id \neq id_s$

■

The history variable $seen_r$ records all the (jd, id) pairs the receiver has ever seen. Thus, when the receiver associates an identifier id to a received jd, the pair (jd, id) is added to $seen_r$. Due to crashes the receiver might associate two different id identifiers to the same jd identifier. However, it can never happen that the receiver associates the same id to different jds.

Invariant 9.9

1. If $id_r \neq \texttt{nil}$ then $(jd_r, id_r) \in seen_r$
2. If $(jd, id) \in seen_r \wedge (jd', id) \in seen_r$ then $jd = jd'$
3. If $(\texttt{accept}, jd, id) \in rs$ then $(jd, id) \in seen_r$

■

Invariant 9.10

1. If $mode_s = \texttt{needid} \wedge mode_r = \texttt{accept} \wedge jd_s = jd_r$ then
   $(\texttt{send}, \_, id_r) \notin sr \wedge (\texttt{done}, id_r) \notin sr$

■

The final invariant corresponds to Invariant 8.12 at the G level. It states that there are always enough unused ids and jds left.

Invariant 9.11

1. $|ID \setminus issued_r| = \infty$
2. $|JD \setminus jd\text{-}used_s| = \infty$

■

Below we refer to the conjunction of the invariants by I_{H^h}.

9.4.3 Safety

The safe I/O automata $A^{h\prime}_{\mathrm{H}}$ and A'_{G} do not agree on their input and output actions. The difference is however very small: $A^{h\prime}_{\mathrm{H}}$ adds packets to the channel in $send\_pkt_{sr}(\mathtt{send}, m, id)$ steps, whereas the corresponding steps in A'_{G} are $send\_pkt_{sr}(m, id)$. There is a similar difference with respect to $send\_pkt_{rs}(\mathtt{ack}, id, b)$ steps and the corresponding $receive\_pkt_{sr}$ and $receive\_pkt_{rs}$ steps. So, define the following action mapping:

$$\begin{array}{rcl}\rho & \triangleq & [send\_pkt_{sr}(m, id) \mapsto send\_pkt_{sr}(\mathtt{send}, m, id) \mid m \in Msg \wedge id \in ID] \cup \\ & & [receive\_pkt_{sr}(m, id) \mapsto receive\_pkt_{sr}(\mathtt{send}, m, id) \mid m \in Msg \wedge id \in ID] \cup \\ & & [send\_pkt_{rs}(id, b) \mapsto send\_pkt_{rs}(\mathtt{ack}, id, b) \mid id \in ID \wedge b \in \mathsf{Bool}] \cup \\ & & [receive\_pkt_{rs}(id, b) \mapsto receive\_pkt_{rs}(\mathtt{ack}, id, b) \mid id \in ID \wedge b \in \mathsf{Bool}] \cup \\ & & [a \mapsto a \mid a \in acts(A'_{\mathrm{G}}) \setminus \mathcal{A}_{\mathrm{G}}]\end{array}$$

where $\mathcal{A}_{\mathrm{G}}$ is defined in Section 8.4 and contains all the actions which are not being renamed by ρ. Clearly ρ is applicable to G', so define $\mathrm{G}^{\rho\prime} = (A^{\rho\prime}_{\mathrm{G}}, L^{\rho\prime}_{\mathrm{G}})$ as follows.

$$\mathrm{G}^{\rho\prime} \quad \triangleq \quad \rho(\mathrm{G}')$$

By Proposition 4.6, $L^{\rho\prime}_{\mathrm{G}}$ is induced by $\rho(Q_{\mathrm{G}})$.

We now define a function from $states(A^{h\prime}_{\mathrm{H}})$ to $states(A^{\rho\prime}_{\mathrm{G}})$. Below, in Lemma 9.13, this function is proved to be a refinement mapping from $A^{h\prime}_{\mathrm{H}}$ to $A^{\rho\prime}_{\mathrm{G}}$ with respect to I_{H^h} and I_{G}. (Note, that the invariant I_{G} of A_{G} is also an invariant of $A^{\rho\prime}_{\mathrm{G}}$.)

Definition 9.12 (Refinement Mapping from $A^{h\prime}_{\mathrm{H}}$ to $A^{\rho\prime}_{\mathrm{G}}$)
If $s \in states(A^{h\prime}_{\mathrm{H}})$ then define $R_{\mathrm{HG}}(s)$ to be the state $u \in states(A^{\rho\prime}_{\mathrm{G}})$ such that

1. $$\begin{array}{lcl} u.mode_s & = & s.mode_s \\ u.buf_s & = & s.buf_s \\ u.used_s & = & s.used_s \\ u.current\text{-}msg_s & = & s.current\text{-}msg_s \\ u.current\text{-}ack_s & = & s.current\text{-}ack_s \\ u.last_r & = & s.last_r \\ u.buf_r & = & s.buf_r \\ u.issued_r & = & s.issued_r \\ u.nack\text{-}buf_r & = & s.nack\text{-}buf_r \\ u.current\text{-}ok & = & s.current\text{-}ok \end{array}$$
2. $u.last_s \quad = \quad s.id_s$
3. $$\begin{array}{lcl} u.good_s & = & (\text{if } s.mode_s = \mathtt{needid} \text{ then} \\ & & \quad \{id \mid (\mathtt{accept}, s.jd_s, id) \in s.rs\} \cup \\ & & \quad (\text{if } s.mode_r = \mathtt{accept} \wedge s.jd_r = s.jd_s \text{ then } \{s.id_r\} \text{ else } \emptyset) \\ & & \text{else } \emptyset) \end{array}$$
4. $u.mode_r \quad = \quad (\text{if } s.mode_r = \mathtt{accept} \text{ then } \mathtt{idle} \text{ else } s.mode_r)$
5. $u.good_r \quad = \quad (\text{if } s.mode_r = \mathtt{accept} \text{ then } \{s.id_r\} \text{ else } \emptyset)$
6. The packets in each channel in u are exactly the `send` and `ack` packets in the same channel in s.

■

Lemma 9.13

$A_{\mathrm{H}}^{h\,\prime} \leq_R A_{\mathrm{G}}^{\rho\,\prime}$ *via* R_{HG}.

Proof

We prove that R_{HG} is a refinement mapping from $A_{\mathrm{H}}^{h\,\prime}$ to $A_{\mathrm{G}}^{\rho\,\prime}$ with respect to I_{H^h} and I_{G}. We check the two conditions (which we call base case and inductive case, respectively) of Definition 5.2.

Base Case

It is easy to see that for the start state s of $A_{\mathrm{H}}^{h\,\prime}$, $R_{\mathrm{GD}}(s)$ is a start state of $A_{\mathrm{G}}^{\rho\,\prime}$.

Inductive Case

Assume $(s, a, s') \in \mathit{steps}(A_{\mathrm{H}}^{h\,\prime})$ such that s and s' satisfy I_{H^h} and $R_{\mathrm{HG}}(s)$ satisfies I_{G}. Below we consider cases based on a (and sometimes subcases of each case) and for each (sub)case we define a finite execution fragment α of $A_{\mathrm{G}}^{\rho\,\prime}$ of the form $(R_{\mathrm{HG}}(s), a', u'', a'', u''', \ldots, R_{\mathrm{HG}}(s'))$ with $\mathit{trace}(\alpha) = \mathit{trace}(a)$. For brevity we let u denote $R_{\mathrm{HG}}(s)$ and u' denote $R_{\mathrm{HG}}(s')$.

Unless otherwise stated we let Part 1–6 refer to the three parts of Definition 9.12.

$\underline{a \in \{\mathit{send\_msg}(m), \mathit{receive\_msg}(m), \mathit{ack}(b), \mathit{recover}_s\}}$

Then it is easy to see that $(u, a, u') \in \mathit{steps}(A_{\mathrm{G}}^{\rho\,\prime})$.

$\underline{a = \mathit{crash}_s}$

We show that $(u, \mathit{crash}_s, u'', \mathit{shrink\_good}_s(I), u')$, where u'' and I are defined below, is a finite execution fragment of $A_{\mathrm{G}}^{\rho\,\prime}$ by showing that $(u, \mathit{crash}_s, u'')$ and $(u'', \mathit{shrink\_good}_s(I), u')$ are steps of $A_{\mathrm{G}}^{\rho\,\prime}$. Clearly the execution fragment has the right trace.

Define u'' to be the same as u' except that $u''.\mathit{good}_s = u.\mathit{good}_s$. Then it is easy to see that $(u, \mathit{crash}_s, u'') \in \mathit{steps}(A_{\mathrm{G}}^{\rho\,\prime})$.

Now, if $s.\mathit{mode}_s = \mathtt{needid}$ then $u''.\mathit{good}_s$ might be nonempty whereas $u'.\mathit{good}_s = \emptyset$ according to R_{HG}. So, define $I = u''.\mathit{good}_s$. (Note, $I = \emptyset$ if $s.\mathit{mode}_s \neq \mathtt{needid}$.) Then, obviously, $(u'', \mathit{shrink\_good}_s(I), u') \in \mathit{steps}(A_{\mathrm{G}}^{\rho\,\prime})$.

$\underline{a = \mathit{crash}_r}$

We show that $(u, \mathit{crash}_r, u'', \mathit{shrink\_good}_r(I), u')$, where $I = u.\mathit{good}_r$ and u'' is defined below, is a finite execution fragment of $A_{\mathrm{G}}^{\rho\,\prime}$ by showing that $(u, \mathit{crash}_r, u'')$ and $(u'', \mathit{shrink\_good}_r(I), u')$ are steps of $A_{\mathrm{G}}^{\rho\,\prime}$. Clearly the execution fragment has the right trace.

Define u'' to be the same as u', except that $u''.\mathit{good}_r = u.\mathit{good}_r$.

It is easy to see that $(u, \mathit{crash}_r, u'') \in \mathit{steps}(A_{\mathrm{G}}^{\rho\,\prime})$. The only interesting case is to show that good_r is handled correctly but from the definition of u'' we have $u''.\mathit{good}_r = u.\mathit{good}_r$, which is as required.

Since $u'.mode_r = \texttt{rec}$, we get from Invariant 9.5 that $u'.current\text{-}ok$ and then also $u''.current\text{-}ok$ are *false*, so $shrink\_good_r(I)$ is enabled in u''. The only difference between u'' and u' is the value of $good_r$. We have $u''.good_r = I$ and $u'.good_r = \emptyset$ since $s'.mode_r = \texttt{rec} \neq \texttt{accept}$. This change in $good_r$ is as required by the definition of $shrink\_good_r(I)$ in $A_G^{\rho}{}'$.

$a = recover_r$

We show that $(u, recover_r, u') \in steps(A_G^{\rho}{}')$. This step (and finite execution fragment) clearly has the right trace.

First note that $recover_r$ is enabled in u. We then carry out a case-by-case check to see that all state variables change appropriately. The only interesting cases are $good_r$ and $issued_r$.

Both $u.good_r = \emptyset$ and $u'.good_r = \emptyset$ by the definition of R_{HG} since $mode_r \neq \texttt{accept}$ in s and s'. Thus, the value of $good_r$ is unchanged as required by the definition of $recover_r$ in $A_G^{\rho}{}'$.

From the definition of $recover_r$ in $A_H^{h}{}'$ and R_{HG} we have that $u.issued_r = u'.issued_r$. To show that it is allowed by $recover_r$ in $A_G^{\rho}{}'$ to leave $issued_r$ unchanged, we must show that $u.used_s \subseteq u.issued_r$ and $u.good_s \subseteq u.issued_r$. But both of these requirements follow directly from the definition of R_{HG} and Invariant 9.1.

$a = choose\_jd(jd)$

We show that $(u, prepare, u') \in steps(A_G^{\rho}{}')$. This step (and finite execution fragment) clearly has the right trace.

Since $choose\_jd(jd)$ is enabled in s and $u = R_{\mathrm{HG}}(s)$, it is immediate that $prepare$ is enabled in u. A case analysis on the variables of $A_G^{\rho}{}'$ shows that all are modified properly; the only interesting case is that of $good_s$. There, the definition of $prepare$ in $A_G^{\rho}{}'$ requires that $u'.good_s = \emptyset$. We must show that that is the case:

First, assume $s.jd_r = \texttt{nil}$. By the definition of $choose\_jd(jd)$ in $A_H^{h}{}'$ we have $s'.jd_s \neq \texttt{nil}$, so since $s'.jd_r = s.jd_r$, we have $s'.jd_s \neq s'.jd_r$.
Now assume $s.jd_r \neq \texttt{nil}$. Then Invariant 9.2 gives us that $s.jd_r \in s.js\text{-}used_s$ and since $s'.jd_r = s.jd_r$ we have $s'.jd_r \in s.js\text{-}used_s$. By the definition of $choose\_jd(jd)$ in $A_H^{h}{}'$ we have $s'.jd_s \notin s'.jd\text{-}used_s$, so also in this case we get $s'.jd_s \neq s'.jd_r$.

From Invariant 9.2 we get $jds(s.rs) \subseteq s.jd\text{-}used_s$. By the definition of $choose\_jd(jd)$ in $A_H^{h}{}'$ we have $s'.jds(s'.rs) = jds(s.rs)$ and $s'.jd_s \notin s.jd\text{-}used_s$, so we get $s'.jd_s \notin jds(s'.rs)$.

Finally, since $s'.jd_s \neq s'.jd_r$ and $s'.jd_s \notin jds(s'.rs)$, we get from the definition of R_{HG} that $u'.good_s = \emptyset$ as required.

$a = send\_pkt_{sr}(\texttt{needid}, jd)$

We show that $u' = u$. Then the execution fragment u of $A_G^{\rho}{}'$ clearly has the right properties.

The only difference between s and s' is that s contains an additional `needid` message in the sr channel. But this does not affect the values of any of the variables of $A_G^{\rho}{}'$ according to the definition of R_{HG}.

$a = receive\_pkt_{sr}(\texttt{needid}, jd)$

We consider two cases.

1. $s.mode_r \neq$ `idle`.
   Then the only difference between s and s' is that the latter is missing one `needid` packet from the sr channel. But this does not affect the values of any variables of ${A_{\mathrm{G}}^{\rho}}'$, so that $u' = u$. Then the execution fragment u of ${A_{\mathrm{G}}^{\rho}}'$ clearly has the right properties.
2. $s.mode_r =$ `idle`.
   There are two subcases.
   (a) $s.mode_s \neq$ `needid` or $jd \neq s.jd_s$.
   We show that $(u, grow\_good_r(\{id\}), u') \in steps(A_{\mathrm{G}}^{\rho})$, where id is the identifier chosen in the step of ${A_{\mathrm{H}}^{h}}'$, i.e., $id = s'.id_r$. Clearly the step has the right trace (the empty trace).
   The definition of the step in ${A_{\mathrm{H}}^{h}}'$ implies that $id \notin s.issued_r$. From the definition of R_{HG} we have $u.issued_r = s.issued_r$, so that $grow\_good_r(\{id\})$ is enabled in u.
   We consider the state changes. From the definition of R_{HG} we have $u.good_r = \emptyset$ and $u'.good_r = \{id\}$. This is the change to $good_r$ specified by the definition of $grow\_good_r(\{id\})$. Also, the step of ${A_{\mathrm{H}}^{h}}'$ explicitly adds id to $issued_r$, which is as required by the definition of $grow\_good_r(\{id\})$ in ${A_{\mathrm{G}}^{\rho}}'$.
   We claim that all variables of ${A_{\mathrm{G}}^{\rho}}'$ other than $good_r$ and $issued_r$ have the same values in u and u'. This is immediate for $mode_s$, buf_s, $used_s$, $current\text{-}msg_s$, $current\text{-}ack_s$, buf_r, $last_r$ $nack\text{-}buf_r$, $current\text{-}ok$, and $last_s$. For $mode_r$, we have a change at the H level, from `idle` to `accept`. But both of these correspond to `idle` at the G level.
   We now show that $u.good_s = u'.good_s$. We make a case analysis based on the definition of this case. First assume $s.mode_s \neq$ `needid`. Then also $s'.mode_s \neq$ `needid` so from the definition of R_{HG} we have $u.good_s = u'.good_s = \emptyset$ as needed.
   Now, assume $s.mode_s =$ `needid` and $jd \neq s.jd_s$. Since $s'.jd_r = jd$ and $s'.jd_s = s.jd_s$ we get $s'.jd_s \neq s'.jd_r$, so even though $mode_r$ changes to `accept` in ${A_{\mathrm{H}}^{h}}'$, it is easy to see from the definition of R_{HG} that $u.good_s = u'.good_s$.
   Finally, the only difference between the channels in s and s' is that the sr channel in s' is missing one `needid` packet. But then the values of the channels in u and u' are the same.
   (b) $s.mode_s =$ `needid` and $jd = s.jd_s$.
   We show that $(u, grow\_good_r(\{id\}), u'', grow\_good_s(\{id\}), u')$, where u'' is defined below and $id = s'.id_r$, is a finite execution fragment of ${A_{\mathrm{G}}^{\rho}}'$. We do this by showing that $(u, grow\_good_r(\{id\}), u'')$ and $(u'', grow\_good_s(\{id\}), u')$ are steps of ${A_{\mathrm{G}}^{\rho}}'$. The execution fragment clearly has the right trace.
   Define u'' to be the same as u', except that $u''.good_s = u'.good_s \setminus \{id\}$.
   The argument that $(u, grow\_good_r(\{id\}), u'')$ is a step of ${A_{\mathrm{G}}^{\rho}}'$ is the same as the argument for the previous case, except for the part about $good_s$. Here, $u.good_s = u''.good_s$ by explicit construction.
   To show that $(u'', grow\_good_s(\{id\}), u')$ is a step of ${A_{\mathrm{G}}^{\rho}}'$, it suffices to note that $id \in u''.issued_r$, $id \in u''.good_r$, and $id \notin u''.used_s$. (This latter claim uses Invariant 9.1.)

$a = send\_pkt_{rs}(\texttt{accept}, jd, id)$

We show that $u' = u$. Then the execution fragment u of ${A_{\mathrm{G}}^{\rho}}'$ clearly has the right properties.

The only difference between s and s' is that s' contains an additional `accept` message in the sr channel. We claim that this does not affect the values of any of the ${A_{\mathrm{G}}^{\rho}}'$ variables.

The only interesting case to check is the value of $good_s$. The only way the step can modify this variable according to R_{HG} is to add an id to $good_s$, by putting id_r to $good_s$, by putting an $(\mathtt{accept}, s'.jd_s, id)$ message into the rs channel. By definition of the step in H, it must be that $s'.jd_s = s.jd_r$ and $id = s.id_r$. Since $s.jd_s = s'.jd_s$, it follows that $s.jd_s = s.jd_r$. But then $id \in u.good_s$. This contradicts the assumption that the step modified this variable.

$a = receive\_pkt_{rs}(\mathtt{accept}, jd, id)$

There are two cases.

1. $s.mode_s = \mathtt{rec}$

   In this case the only difference between s' and s is that s has an extra $(\mathtt{accept}, id, jd)$ packet on rs, but from the definition of R_{HG} we see that this does not affect any of the variables in $A_{\mathrm{G}}^{\rho}{}'$ since $s.mode_s \neq \mathtt{needid}$. Thus $u' = u$. The the execution fragment u of $A_{\mathrm{G}}^{\rho}{}'$ has the right properties.
2. $s.mode_s \neq \mathtt{rec}$

   We consider cases

   (a) $s.mode_s \neq \mathtt{needid}$ or $jd \neq s.jd_s$.

   We show that $u' = u$. The the execution fragment u of $A_{\mathrm{G}}^{\rho}{}'$ has the right properties. The only difference between s and s' is that s' removes a single $\mathtt{accept}$ message in the sr channel and that $done\text{-}buf_s$ might be updated. We claim that this does not affect the values of any of the $A_{\mathrm{G}}^{\rho}{}'$ variables; the only interesting case to check is that of $good_s$, and there, the fact that $s.mode_s \neq \mathtt{needid}$ or $jd \neq s.jd_s$ implies that $good_s$ has the same value in u and u'.

   (b) $s.mode_s = \mathtt{needid}$ and $jd = s.jd_s$.

   We show that $(u, choose\_id(id), u'', shrink\_good_s(I), u')$, where $I = u.good_s$ and u'' is defined below, is an execution fragment of $A_{\mathrm{G}}^{\rho}{}'$ by showing that $(u, choose\_id(id), u'')$ and $(u'', shrink\_good_s(I), u')$ are steps of $A_{\mathrm{G}}^{\rho}{}'$. Clearly the execution fragment has the right trace.

   Define u'' to be the same as u' except that $u''.good_s = I$.

   First consider $(u, choose\_id(id), u'')$. Since $s.mode_s = \mathtt{needid}$, we have $u.mode_s = \mathtt{needid}$. Then, to prove that $choose\_id(id)$ is enabled in u, we need to show that $id \in u.good_s$. In s, we have $(\mathtt{accept}, id, jd)$ in the rs channel, and moreover $jd = s.jd_s$. Thus, from the definition of R_{HG} we have $id \in u.good_s$ as needed.

   Now we consider the effects on the variables in $A_{\mathrm{G}}^{\rho}{}'$. A case analysis shows that the changes reflected in u'' are as specified by the step of $A_{\mathrm{G}}^{\rho}{}'$. The only interesting case is that of $good_s$, where the definition of $u''.good_s = I = u.good_s$ ensures that the value is unchanged, as required by the definition of $choose\_id(id)$ in $A_{\mathrm{G}}^{\rho}{}'$.

   To see that $(u'', shrink\_good_s(I), u')$ is a step of $A_{\mathrm{G}}^{\rho}{}'$, note that $u'.good_s = \emptyset$. Therefore, the changes are as required by the definition of $shrink\_good_s(I)$ in $A_{\mathrm{G}}^{\rho}{}'$.

$a = send\_pkt_{sr}(\mathtt{send}, m, id)$

Then it is easy to see that $(u, send\_pkt_{sr}(m, id), u') \in steps(A_{\mathrm{G}}^{\rho}{}')$. This step (and execution fragment) clearly has the right trace.

$a = \mathit{receive\_pkt}_{sr}(\mathtt{send}, m, id)$

We show that $(u, \mathit{receive\_pkt}_{sr}(m, id), u') \in \mathit{steps}(A_{\mathrm{G}}^{\rho\,\prime})$. This step (and execution fragment) has the right trace.

We consider four (exclusive and exhaustive) cases.

1. $s.\mathit{mode}_r = \mathtt{rec}$.

   Then the only change from s to s' is the removal of the single message from the sr channel. Since also $u.\mathit{mode}_r = \mathtt{rec}$, this corresponds to the right change in $A_{\mathrm{G}}^{\rho\,\prime}$.

2. $s.\mathit{mode}_r = \mathtt{accept}$ and $id = s.id_r$.

   Then, from the definition of R_{HG} we have that $u.\mathit{mode}_r = \mathtt{idle}$ and $id \in u.\mathit{good}_r$, such that the required state change of the receiver variables of $A_{\mathrm{G}}^{\rho\,\prime}$ is described by the first alternative in the nested if-then-else construct in the step rule for $\mathit{receive\_pkt}_{sr}(m, id)$. A case analysis shows that all variables of $A_{\mathrm{G}}^{\rho\,\prime}$ are handled correctly. The interesting cases are *current-ok* and good_r.

   For *current-ok*, we consider two cases.

   First, if $id = s.id_s$, then we have $id = u.\mathit{last}_s$. Moreover, $s.\mathit{mode}_s \in \{\mathtt{send}, \mathtt{rec}\}$ by Invariant 9.6. If $s.\mathit{mode}_s = \mathtt{rec}$ then Invariant 9.5 implies that $s.\mathit{current\text{-}ok}$ is already *false*, so setting it to *false* in $A_{\mathrm{H}}^{h\,\prime}$ is a no-op, as required by the step in $A_{\mathrm{G}}^{\rho\,\prime}$. If $s.\mathit{mode}_s = \mathtt{send}$ both algorithms set *current-ok* to *false*.

   On the other hand, if $id \neq s'.id_s$, then also $id \neq u.\mathit{last}_s$. Thus in this case neither level changes *current-ok*.

   For good_r, note that $u.\mathit{good}_r = \{s.id_r\}$ since $s.\mathit{mode}_r = \mathtt{accept}$ and $u'.\mathit{good}_r = \emptyset$ since $s'.\mathit{mode}_r \neq \mathtt{accept}$. Since $id = s.id_r$, this change is as required by the definition of $\mathit{receive\_pkt}_{sr}(m, id)$ of $A_{\mathrm{G}}^{\rho\,\prime}$.

3. $s.\mathit{mode}_r \neq \mathtt{rec}$ and $(s.\mathit{mode}_r \neq \mathtt{accept}$ or $id \neq s.id_r)$

   We show that the required state changes of the receiver variables of $A_{\mathrm{G}}^{\rho\,\prime}$ are not described by the first alternative inside the nested if-then-else construct. First, if $s.\mathit{mode}_r \neq \mathtt{accept}$ then $u.\mathit{good}_r = \emptyset$ which gives the result. Next, if $s.\mathit{mode}_r = \mathtt{accept}$ we have $u.\mathit{good}_r = \{s.id_r\}$, but from the definition of this case we must have $id \neq s.id_r$, so again the result follows.

   We now consider two cases

   (a) $id \neq s.\mathit{last}_r$

   Then we have $s'.\mathit{nack\text{-}buf}_r = s.\mathit{nack\text{-}buf}_r \mathbin{\hat{}} id$. Since $id \neq u.\mathit{last}_r$, by the definition of R_{HG}, we also have $u'.\mathit{nack\text{-}buf}_r = u.\mathit{nack\text{-}buf}_r \mathbin{\hat{}} id$. It is now easy to see that all state variables of $A_{\mathrm{G}}^{\rho\,\prime}$ are handled correctly.

   (b) $id = s.\mathit{last}_r$

   In this case, the $A_{\mathrm{H}}^{h\,\prime}$ level makes no changes (that is, the only difference between s and s' is that the latter has the one message deleted from the sr channel). We must thus show that all variables but sr of $A_{\mathrm{G}}^{\rho\,\prime}$ have the same values in u and u'.

   First we note that the $A_{\mathrm{G}}^{\rho\,\prime}$ step does not choose the second alternative inside the nested if-then-else construct since the definition of this case and R_{HG} gives us that $id = u.\mathit{last}_r$.

   We must show that $A_{\mathrm{G}}^{\rho\,\prime}$ does not choose the third alternative. The only way $A_{\mathrm{G}}^{\rho\,\prime}$ can choose the third alternative is if $u.\mathit{mode}_r = \mathtt{idle}$. From the definition of R_{HG} we see that this is the case if $s.\mathit{mode}_r \in \{\mathtt{idle}, \mathtt{accept}\}$. Now, Invariant 9.3 gives us that

$s.last_r = \mathtt{nil}$, but this contradicts the definition of this case ($id = s.last_r$), thus, we cannot have $u.mode_r = \mathtt{idle}$ which again implies that ${A_G^\rho}'$ does not choose the third alternative.
That suffices.

$a = send\_pkt_{sr}(\mathtt{done}, id)$

This step of ${A_H^h}'$ changes $done\text{-}buf_s$ and may change the channel sr, but from the definition of R_{HG} we see that this does not change any of the variables in ${A_G^\rho}'$, so we have $u = u'$. Thus, the finite execution fragment u clearly has the right properties.

$a \in \{send\_pkt_{rs}(\mathtt{ack}, id, b), receive\_pkt_{rs}(\mathtt{ack}, id, b)\}$

Then it is easy to see that $(u', send\_pkt_{rs}(id, b), u)$ and $(u', receive\_pkt_{rs}(id, b), u)$, respectively, are steps of ${A_G^\rho}'$.

$a = receive\_pkt_{sr}(\mathtt{done}, id)$

We consider cases.

1. $s.mode_r = \mathtt{accept}$ and $id = s.id_r$.
   There are two subcases.
   (a) $s.mode_s \neq \mathtt{needid}$ or
   ($s.mode_s = \mathtt{needid}$ and $s.jd_r \neq s.jd_s$) or
   ($s.mode_s = \mathtt{needid}$ and $s.jd_r = s.jd_s$ and $(\mathtt{accept}, s.jd_s, s.id_r) \in s.rs$)
   We show that $(u, shrink\_good_r(\{id\}), u') \in steps({A_G^\rho}')$. This step (and execution fragment) clearly has the right trace.
   First, we show that $shrink\_good_r(\{id\})$ is enabled in u.
      i. $s.mode_s \neq \mathtt{needid}$
      Then the precondition of $shrink\_good_r(\{id\})$ is satisfied by u. The only interesting case is if $s.mode_s = \mathtt{send}$. In this case we must show that $u.last_s \neq id$, i.e., that $s.id_s \neq id$. Since $(\mathtt{done}, id) \in s.sr$, Invariant 9.8 gives the result.
      ii. $s.mode_s = \mathtt{needid}$ and $s.jd_r \neq s.jd_s$
      Here, it suffices to show that $id \notin u.good_s$. From R_{HG} we get that $u.good_s = \{id' \mid (\mathtt{accept}, s.jd_s, id')\}$. From Invariant 9.9 Part 3 we get that $u.good_s$ is a subset of the set S defined as $S = \{id' \mid (s.jd_s, id') \in s.seen_r\}$, so it suffices to show that $id \notin S$. Since $s.id_r = id \neq \mathtt{nil}$, we get from Invariant 9.9 Part 1 that $(s.jd_r, id) \in s.seen_r$ and Part 2 of the same invariant then implies that $(s.jd_s, id) \notin s.seen_r$ since $s.jd_s \neq s.jd_r$ in the case we consider here. Thus, the result follows.
      iii. $s.mode_s = \mathtt{needid}$ and $s.jd_r = s.jd_s$ and $(\mathtt{accept}, s.jd_s, s.id_r) \in s.rs$
      Invariant 9.10 implies that this situation cannot occur.
   We now show that the variable changes are allowed by the step of ${A_G^\rho}'$.
   First, we show that $good_r$ is handled correctly. By definition of this case and R_{HG}, we get that $u.good_r = \{id\}$ and $u'.good_r = \emptyset$. Thus, $good_r$ changes in a way allowed by $shrink\_good_r(\{id\})$ in ${A_G^\rho}'$.
   We must show that no other variables have different values in u' and u. The interesting cases are $mode_r$, $last_r$, and $good_s$.

For $mode_r$ we have $s.mode_r = \texttt{accept}$ and $s'.mode_r = \texttt{idle}$, but then R_{HG} gives us $u'.mode_r = u.mode_r = \texttt{idle}$, as needed.

For $last_r$ we have $u.last_r = \texttt{nil}$ from Invariant 9.3 since $s.mode_r = \texttt{accept}$, and $u'.last_r = \texttt{nil}$ from the definition of the $A_{\mathrm{H}}^{h\,\prime}$ step. Thus, $last_r$ is unchanged as needed.

Finally, we consider $good_s$

i. $s.mode_s \neq \texttt{needid}$
Then, since also $s'.mode_s \neq \texttt{needid}$, R_{HG} gives us $u'.good_s = u.good_s (= \emptyset)$ as needed.

ii. $s.mode_s = \texttt{needid}$ and $s.jd_r \neq s.jd_s$
Since $s'.mode_s = \texttt{needid}$, we have $s'.jd_s \neq \texttt{nil}$ (easy invariant), so since $s'.jd_r = \texttt{nil}$ we have $s'.jd_r \neq s'.jd_s$. Now, since jd_s and rs are unchanged in the $A_{\mathrm{H}}^{h\,\prime}$ step, we clearly get from R_{HG} that $u'.good_s = u.good_s$ as needed.

iii. $s.mode_s = \texttt{needid}$ and $s.jd_r = s.jd_s$ and $(\texttt{accept}, s.jd_s, s.id_r) \in s.rs$
Again, Invariant 9.10 implies that this situation cannot occur.

(b) $s.mode_s = \texttt{needid}$, $s.jd_r = s.jd_s$, and $(\texttt{accept}, s.jd_s, s.id_r) \notin s.rs$
We show that $(u, shrink\_good_s(\{id\}), u'', shrink\_good_r(\{id\}), u')$, where u'' is defined below, is an execution fragment of $A_{\mathrm{G}}^{\rho\,\prime}$ by showing that $(u, shrink\_good_s(\{id\}), u'')$ and $(u'', shrink\_good_r(\{id\}), u')$ are steps of $A_{\mathrm{G}}^{\rho\,\prime}$. The execution fragment clearly has the right trace.
Define u'' to be the same as u except that $u''.good_s = u.good_s \setminus \{id\}$.
Then obviously $(u, shrink\_good_s(\{id\}), u'') \in steps(A_{\mathrm{G}}^{\rho\,\prime})$.
We show that also $(u'', shrink\_good_r(\{id\}), u') \in steps(A_{\mathrm{G}}^{\rho\,\prime})$.
Since $u''.mode_s = u.mode_s = \texttt{needid}$ and $id \notin u''.good_s$, $shrink\_good_r(\{id\})$ is enabled in u''.
We show that all variables are handled correctly.
For all other variables than $good_s$ the arguments are as in the case above.
We show that $u''.good_s = u'.good_s$. We have $s'.jd_r = \texttt{nil} \neq s'.jd_s$ (since $s'.mode_s = \texttt{needid}$), so the definition of R_{HG} and u'' gives us:
$u''.good_s = (\{id' \mid (\texttt{accept}, s.jd_s, id') \in s.rs\} \cup \{id\}) \setminus \{id\}$ and
$u'.good_s = \{id' \mid (\texttt{accept}, s'.jd_s, id') \in s'.rs\}$.
Since jd_s and rs are unchanged, it suffices to show $id \notin \{id' \mid (\texttt{accept}, s.jd_s, id')\}$, but since $id = s.id_r$, this follows directly from the definition of this subcase.
That suffices.

2. $s.mode_r = \texttt{ack}$ and $id = s.last_r$.
We show that $(u, cleanup_r, u') \in steps(A_{\mathrm{G}}^{\rho\,\prime})$. This step (and execution fragment) clearly has the right trace.
Since $(\texttt{done}, id) \in s.sr$ we get from Invariant 9.8 that $id \neq s.id_s$, so from the definition of R_{HG} and the hypothesis we get $u.last_s \neq u.last_r$. Also, since $s.mode_r = \texttt{ack}$, we have $u.mode_r = \texttt{ack}$. Thus, $cleanup_r$ is enabled in u.
All variables are handled correctly. The changes to $last_r$ and $mode_r$ in $A_{\mathrm{H}}^{h\,\prime}$ clearly are as required by the definition of $cleanup_r$ in $A_{\mathrm{G}}^{\rho\,\prime}$. Since $mode_r \neq \texttt{accept}$ we have $u.good_r = u'.good_r (= \emptyset)$ as needed. The only other interesting case is $good_s$. But since $mode_r \neq \texttt{accept}$ and jd_s and rs are unchanged by the step in $A_{\mathrm{H}}^{h\,\prime}$, we get from R_{HG} that $u'.good_s = u.good_s$ as needed.

3. Otherwise
Then we claim that $u' = u$.

The only difference between s and s' is the removal of the `done` packet from the sr channel. This does not affect any of the $A_{\mathrm{G}}^{\rho\,\prime}$ variables.

$a = \mathit{grow\text{-}jd\text{-}used}_s(jds)$

This step adds some elements to $\mathit{jd\text{-}used}_s$, but since $\mathit{jd\text{-}used}_s$ is not used in the mapping R_{HG}, we have $u = u'$. Thus, the execution fragment u has the right properties.

$a = \mathit{grow\text{-}issued}_r(ids)$

This transition adds elements to issued_r in $A_{\mathrm{H}}^{h\,\prime}$.
We show that $(u, \mathit{grow\_good}_r(I), u'', \mathit{shrink\_good}_r(I), u')$, where u'' is defined below and $I = s'.\mathit{issued}_r \setminus s.\mathit{issued}_r$, is an execution fragment of $A_{\mathrm{G}}^{\rho\,\prime}$ by showing that $(u, \mathit{grow\_good}_r(I), u'')$ and $(u'', \mathit{shrink\_good}_r(I), u')$ are steps of $A_{\mathrm{G}}^{\rho\,\prime}$. The execution fragment clearly has the right trace.

Define u'' to be the same as u' except that $u''.\mathit{good}_r = u.\mathit{good}_r \cup I$.

From the definition of R_{HG} we get that $I = u'.\mathit{issued}_r \setminus u.\mathit{issued}_r$ which implies that $I \cap u.\mathit{issued}_r = \emptyset$. Thus, $\mathit{grow\_good}_r(I)$ is enabled in u. Now, the only difference between u and u'' is that $u''.\mathit{good}_r = u.\mathit{good}_r \cup I$ (by explicit construction) and $u''.\mathit{issued}_r = u.\mathit{issued}_r \cup I$ (by the definition of $\mathit{grow\text{-}issued}_r$, R_{HG} and u''), but this is as required by $\mathit{grow\_good}_r(I)$ in $A_{\mathrm{G}}^{\rho\,\prime}$.

We now consider $(u'', \mathit{shrink\_good}_r(I), u')$. To show that $\mathit{shrink\_good}_r(I)$ is enabled in u'', we show that $I \cap u''.\mathit{good}_s = \emptyset$ and that $u''.\mathit{last}_s \notin I$.

First, consider the claim that $I \cap u''.\mathit{good}_s = \emptyset$. Since $u''.\mathit{good}_s = u.\mathit{good}_s$ we must show that $I \cap u.\mathit{good}_s = \emptyset$. From Invariant 9.1 and R_{HG} we get that $u.\mathit{good}_s \subseteq s.\mathit{issued}_r$, but since $I \cap s.\mathit{issued}_r = \emptyset$ (by the definition of I) the result follows directly.

Then, consider the claim that $u''.\mathit{last}_s \notin I$. Since $u''.\mathit{last}_s = u.\mathit{last}_s = s.\mathit{id}_s$, we must show that $s.\mathit{id}_s \notin I$. If $s.\mathit{id}_s = \mathtt{nil}$ this is obvious, so assume $s.\mathit{id}_s \neq \mathtt{nil}$. Then Invariant 9.7 gives us that $s.\mathit{id}_s \in s.\mathit{used}_s$, and Invariant 9.1 implies that $s.\mathit{id}_s \in s.\mathit{issued}_r$. Again, since $I \cap s.\mathit{issued}_r = \emptyset$, we get the result.

Thus, $\mathit{shrink\_good}_r(I)$ is enabled in u''.

The only difference between u'' and u' is by the definition of u'' that $u''.\mathit{good}_r = u.\mathit{good}_r \cup I = u'.\mathit{good}_r \cup I$. (The latter equality uses the definitions of $\mathit{grow\text{-}issued}_r$ and R_{HG} to see that $u'.\mathit{good}_r = u.\mathit{good}_r$). To satisfy the requirements in $A_{\mathrm{G}}^{\rho\,\prime}$ we must show that $u'.\mathit{good}_r = u''.\mathit{good}_r \setminus I$. This is only the case if $u'.\mathit{good}_r \setminus I = u'.\mathit{good}_r$, i.e., if $u'.\mathit{good}_r \cap I = \emptyset$. Now, either $u'.\mathit{good}_r = \emptyset$ in which case this result follows directly or $u'.\mathit{good}_r = \{s'.\mathit{id}_r\}$ (with $s'.\mathit{id}_r \neq \mathtt{nil}$). In the latter case we observe that $s'.\mathit{id}_r = s.\mathit{id}_r$, so Invariant 9.1 implies that $u'.\mathit{good}_r \subseteq s.\mathit{issued}_r$, and since $I \cap s.\mathit{issued}_r = \emptyset$, we get that $u'.\mathit{good}_r \cap I = \emptyset$, as needed.

This concludes the simulation proof.

■

With this simulation result we can prove that A_{H} safely implements A_{G}.

Theorem 9.14 (A_{H} safely implements A_{G})

$A_{\mathrm{H}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{G}}$

Proof

By Lemma 9.13 and the soundness of refinement mappings (Lemma 5.8) we get $A_{\mathrm{H}}^{h\,\prime} \sqsubseteq_{\mathrm{S}} A_{\mathrm{G}}^{\rho\,\prime}$, and from Lemma 5.14 we get $A'_{\mathrm{H}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{H}}^{h\,\prime}$. Thus,

$A'_{\mathrm{H}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{G}}^{\rho\,\prime}$

which by substitutivity (Lemma 2.16) implies

$A'_{\mathrm{H}} \setminus \mathcal{A}_{\mathrm{H}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{G}}^{\rho\,\prime} \setminus \mathcal{A}_{\mathrm{H}}$

Then, by the definition of ρ, $\mathcal{A}_{\mathrm{H}}$, and $\mathcal{A}_{\mathrm{G}}$ we get

$A'_{\mathrm{H}} \setminus \mathcal{A}_{\mathrm{H}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{G}}^{\rho\,\prime} \setminus \rho(\mathcal{A}_{\mathrm{G}})$

Now, since ρ only renames actions which are subsequently hidden, this implies

$A'_{\mathrm{H}} \setminus \mathcal{A}_{\mathrm{H}} \sqsubseteq_{\mathrm{S}} A'_{\mathrm{G}} \setminus \mathcal{A}_{\mathrm{G}}$

which finally, by definition, yields the result

$A_{\mathrm{H}} \sqsubseteq_{\mathrm{S}} A_{\mathrm{G}}$

■

9.4.4 Correctness

We can now turn attention to formally proving that $\mathrm{H}^{h\,\prime}$ correctly implements $\mathrm{G}^{\rho\prime}$, which, in turn, then allows us to prove that H correctly implements G.

We start out by giving some basic results about $A_{\mathrm{H}}^{h\,\prime}$. The first results (Lemma 9.15 and Lemma 9.16) describe certain possible steps of $A_{\mathrm{H}}^{h\,\prime}$ in the absence of crashes. The lemmas have one part for each mode in the system and each part is furthermore divided into two subparts. The first subpart states that if the system reaches a certain state, then it will stay in that state at least until a certain action (or certain actions) occur(s). The second subpart then states the resulting state if such an action indeed occurs.

In the remainder of this section we use notation like $\mathit{send\_pkt}_{rs}(\mathtt{accept}, \_, \_)$ to denote the action function $\{\mathit{send\_pkt}(\mathtt{accept}, \mathit{jd}, \mathit{id}) \mid \mathit{jd} \in \mathit{JD} \wedge \mathit{id} \in \mathit{ID}\}$. Similarly, the expression, e.g., $\mathit{send\_pkt}_{rs}(\mathtt{accept}, \_, \mathit{id}_s)$ denotes the action function $\{\mathit{send\_pkt}(\mathtt{accept}, \mathit{jd}, \mathit{id}_s) \mid \mathit{jd} \in \mathit{JD}\}$.

Lemma 9.15

$A_{\mathrm{H}}^{h\,\prime}$ satisfies each of the following formulas

1. (a) $\Box(\Box(\mathit{mode}_s \neq \mathtt{rec}) \wedge \mathit{mode}_s = \mathtt{idle} \Longrightarrow (\mathit{mode}_s = \mathtt{idle}\ \mathcal{W}_i\ \langle \mathit{choose\_jd}(\_)\rangle))$

   (b) $\Box(\mathit{mode}_s = \mathtt{idle} \wedge \langle \mathit{choose\_jd}(\_)\rangle \Longrightarrow \mathit{mode}_s^\circ = \mathtt{needid})$

2. (a) $\forall \mathit{jd} : \Box(\Box(\mathit{mode}_s \neq \mathtt{rec}) \wedge \mathit{mode}_s = \mathtt{needid} \wedge \mathit{jd}_s = \mathit{jd} \Longrightarrow$
   $(\mathit{mode}_s = \mathtt{needid} \wedge \mathit{jd}_s = \mathit{jd}\ \mathcal{W}_i\ \langle \mathit{receive\_pkt}_{rs}(\mathtt{accept}, \mathit{jd}, \_)\rangle))$

   (b) $\Box(\mathit{mode}_s = \mathtt{needid} \wedge \langle \mathit{receive\_pkt}_{rs}(\mathtt{accept}, \mathit{jd}_s, \_)\rangle \Longrightarrow \mathit{mode}_s^\circ = \mathtt{send})$

3. (a) $\forall \mathit{jd} : \forall \mathit{id} : \Box(\Box(\mathit{mode}_s \neq \mathtt{rec}) \wedge \mathit{mode}_s = \mathtt{send} \wedge \mathit{jd}_s = \mathit{jd} \wedge \mathit{id}_s = \mathit{id} \Longrightarrow$
   $(\mathit{mode}_s = \mathtt{send} \wedge \mathit{jd}_s = \mathit{jd} \wedge \mathit{id}_s = \mathit{id}\ \mathcal{W}_i\ \langle \mathit{receive\_pkt}_{rs}(\mathtt{ack}, \mathit{id}, \_)\rangle))$

   (b) $\Box(\mathit{mode}_s = \mathtt{send} \wedge \langle \mathit{receive\_pkt}_{rs}(\mathtt{ack}, \mathit{id}_s, \_)\rangle \Longrightarrow \mathit{mode}_s^\circ = \mathtt{idle})$

Proof

Easy by careful inspection of the steps of $A_{\mathrm{H}}^{h\,\prime}$.

■

Lemma 9.16

$A_{\mathrm{H}}^{h\,\prime}$ satisfies each of the following formulas

1. (a) $\Box(\Box(mode_r \neq \mathtt{rec}) \wedge mode_r = \mathtt{idle} \Longrightarrow$
$(mode_r = \mathtt{idle}\ \mathcal{W}_i\ \langle receive\_pkt_{sr}(\mathtt{needid}, \_)\rangle))$

   (b) $\forall jd : \Box(mode_r = \mathtt{idle} \wedge \langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle \Longrightarrow$
$mode_r^{\circ} = \mathtt{accept} \wedge jd_r^{\circ} = jd)$

2. (a) $\forall jd : \forall id : \Box(\Box(mode_r \neq \mathtt{rec}) \wedge mode_r = \mathtt{accept} \wedge jd_r = jd \wedge id_r = id \Longrightarrow$
$(mode_r = \mathtt{accept} \wedge jd_r = jd \wedge id_r = id\ \mathcal{W}_i$
$\langle receive\_pkt_{sr}(\mathtt{send}, \_, id)\rangle \vee \langle receive\_pkt_{sr}(\mathtt{done}, id)\rangle))$

   (b) $\Box(mode_r = \mathtt{accept} \wedge \langle receive\_pkt_{sr}(\mathtt{send}, \_, id_r)\rangle \Longrightarrow mode_r^{\circ} = \mathtt{rcvd})$
$\Box((mode_r = \mathtt{accept} \wedge \langle receive\_pkt_{sr}(\mathtt{done}, id_r)\rangle) \Longrightarrow mode_r^{\circ} = \mathtt{idle})$

3. (a) $\forall id : \Box(\Box(mode_r \neq \mathtt{rec}) \wedge mode_r = \mathtt{rcvd} \wedge last_r = id \Longrightarrow$
$(mode_r = \mathtt{rcvd} \wedge last_r = id\ \mathcal{W}_i\ \langle receive\_msg(\_)\rangle \wedge buf_r^{\circ} = \varepsilon))$

   (b) $\Box(mode_r = \mathtt{rcvd} \wedge \langle receive\_msg(\_)\rangle \wedge buf_r^{\circ} = \varepsilon \Longrightarrow mode_r^{\circ} = \mathtt{ack})$

4. (a) $\forall id : \Box(\Box(mode_r \neq \mathtt{rec}) \wedge mode_r = \mathtt{ack} \wedge last_r = id \Longrightarrow$
$(mode_r = \mathtt{ack} \wedge last_r = id\ \mathcal{W}_i\ \langle receive\_pkt_{sr}(\mathtt{done}, id)\rangle))$

   (b) $\Box(mode_r = \mathtt{ack} \wedge \langle receive\_pkt_{sr}(\mathtt{done}, last_r)\rangle \Longrightarrow mode_r^{\circ} = \mathtt{idle})$

Proof

Easy by careful inspection of the steps of $A_{\mathrm{H}}^{h\,\prime}$.

■

In the proofs below we furthermore need the following simple lemma.

Lemma 9.17

$A_{\mathrm{H}}^{h\,\prime} \models \Box(mode_s = \mathtt{needid} \wedge mode_r = \mathtt{accept} \wedge jd_s = jd_r \Longrightarrow$
$\neg\langle receive\_pkt_{sr}(\mathtt{send}, \_, id_r)\rangle \wedge \neg\langle receive\_pkt_{sr}(\mathtt{done}, id_r)\rangle)$

Proof

Directly by Invariant 9.10.

■

We now turn attention to more interesting results about the live executions of $\mathrm{H}^{h\,\prime}$. The first lemma states that if the sender stays in `needid` mode, then it will issue infinitely many `needid` packets. This result is actually a simple consequence of weak fairness to the set $C_{\mathrm{H},s1}$. We give the proof in all formal detail.

Lemma 9.18 (`needid` liveness)

$L_{\mathrm{H}}^{h\,\prime} \models \forall jd : \Box(\Box(mode_s = \mathtt{needid} \wedge jd_s = jd) \Longrightarrow \Box\Diamond\langle send\_pkt_{sr}(\mathtt{needid}, jd)\rangle)$

Proof

ASSUME: $\alpha \in L_{\mathrm{H}}^{h\,\prime}$

PROVE: $\alpha \models \forall jd : \Box(\Box(mode_s = \mathtt{needid} \wedge jd_s = jd) \Longrightarrow \Box\Diamond\langle send\_pkt_{sr}(\mathtt{needid}, jd)\rangle)$

$\langle 1\rangle 1.$ ASSUME: jd is arbitrary
PROVE: $\alpha \models \Box(\Box(mode_s = \mathtt{needid} \wedge jd_s = jd) \Longrightarrow \Box\Diamond\langle send\_pkt_{sr}(\mathtt{needid}, jd)\rangle)$

$\langle 2\rangle 1.$ ASSUME: α_1 is an arbitrary suffix of α
PROVE: $\alpha_1 \models \Box(mode_s = \mathtt{needid} \wedge jd_s = jd) \Longrightarrow \Box\Diamond\langle send\_pkt_{sr}(\mathtt{needid}, jd)\rangle$

$\langle 3\rangle 1.$ ASSUME: $\alpha_1 \models \Box(mode_s = \mathtt{needid} \wedge jd_s = jd)$
PROVE: $\alpha_1 \models \Box\Diamond\langle send\_pkt_{sr}(\mathtt{needid}, jd)\rangle$

$\langle 4\rangle 1.$ $\alpha_1 \models WF(C_{\mathrm{H},s1})$

PROOF: By the assumption $\alpha \in L_{\mathrm{H}}^{h'}$ we have $\alpha \models WF(C_{\mathrm{H},s1})$. Then Assumption $\langle 2\rangle$ and Lemma 3.5 Part 1 give the result.

$\langle 4\rangle 2.$ $\alpha_1 \models \Diamond\Box(mode_s \in \{\mathtt{rec}, \mathtt{needid}, \mathtt{send}\} \vee (mode_s = \mathtt{idle} \wedge buf_s = \varepsilon)) \Longrightarrow \Box\Diamond\langle C_{\mathrm{H},s1}\rangle$

PROOF: From $\langle 4\rangle 1$ by expanding WF and noting that $enabled(C_{\mathrm{H},s1}) = (mode_s \in \{\mathtt{rec}, \mathtt{needid}, \mathtt{send}\} \vee (mode_s = \mathtt{idle} \wedge buf_s = \varepsilon))$.

$\langle 4\rangle 3.$ $\alpha_1 \models \Box(mode_s \in \{\mathtt{rec}, \mathtt{needid}, \mathtt{send}\} \vee (mode_s = \mathtt{idle} \wedge buf_s = \varepsilon)) \Longrightarrow \Box\Diamond\langle C_{\mathrm{H},s1}\rangle$

PROOF: Directly from $\langle 4\rangle 2$.

$\langle 4\rangle 4.$ $\alpha_1 \models \Box\Diamond\langle C_{\mathrm{H},s1}\rangle$

PROOF: By Assumption $\langle 3\rangle$, $\langle 4\rangle 3$, and Rule **MP**.

$\langle 4\rangle 5.$ Q.E.D.

PROOF: By $\langle 4\rangle 4$ since Assumption $\langle 3\rangle$ yields that $send\_pkt_{sr}(\mathtt{needid}, jd)$ is the only action in $C_{\mathrm{H},s1}$ which is enabled anywhere in α_1.

$\langle 3\rangle 2.$ Q.E.D.

PROOF: By $\langle 3\rangle 1$ and the definition of implication.

$\langle 2\rangle 2.$ Q.E.D.

PROOF: By $\langle 2\rangle 1$ and Lemma 3.5 Part 2.

$\langle 1\rangle 2.$ Q.E.D.

PROOF: By $\langle 1\rangle 1$ and Lemma 3.5 Part 5.

■

The following lemmas (Lemmas 9.19–9.23) state similar basic results about the live executions of $\mathrm{H}^{h'}$.

Lemma 9.19 (done liveness)

1. $L_{\mathrm{H}}^{h'} \models \forall id : (\Box(mode_s \neq \mathtt{rec}) \wedge id \in done\text{-}buf_s) \leadsto \langle send\_pkt_{sr}(\mathtt{done}, id)\rangle$

2. $L_{\mathrm{H}}^{h'} \models \forall id : \Box(\Box(mode_s \neq \mathtt{rec}) \wedge \Box\Diamond\langle receive\_pkt_{rs}(\mathtt{ack}, id, true)\rangle \Longrightarrow \Box\Diamond\langle send\_pkt_{sr}(\mathtt{done}, id)\rangle$

3. $L_{\mathrm{H}}^{h'} \models \forall jd : \forall id : \Box(\Box(mode_s = \texttt{needid} \wedge jd_s \neq jd) \wedge$
$\Box\Diamond\langle receive\_pkt_{rs}(\texttt{accept}, jd, id)\rangle \Longrightarrow$
$\Box\Diamond\langle send\_pkt_{sr}(\texttt{done}, id)\rangle)$

Proof

We sketch the proof.

1. Consider an arbitrary suffix of a live execution of $\mathrm{H}^{h'}$ and assume that the sender is never crashed in this suffix. In the first state of the suffix, let id be an arbitrary element of $done\text{-}buf_s$ and id' the first element of $done\text{-}buf_s$. Then $send\_pkt_{sr}(\texttt{done}, id')$ is enabled (since $\Box(mode_s \neq \texttt{rec})$) and by fairness eventually $send\_pkt_{sr}(\texttt{done}, id')$ occurs and id' is removed from $done\text{-}buf_s$. By repeating this argument, we get that eventually id is first on $done\text{-}buf_s$ and then eventually $send\_pkt_{sr}(\texttt{done}, id)$ occurs.

2. Here id will infinitely often be put into $done\text{-}buf_s$ by the $receive\_pkt_{rs}(\texttt{ack}, id, true)$ events since $\Box(mode_s \neq \texttt{rec})$. Then Part 1 of this lemma implies the result.

3. Similar to Part 2. When $mode_s = \texttt{needid}$, Invariant 9.6 implies $id_s = \texttt{nil}$. Then, since $jd_s \neq jd$, the each $receive\_pkt_{rs}(\texttt{accept}, jd, id)$ step leads to id being inserted into $done\text{-}buf_s$. Part 1 of this lemma then implies the result.

■

Lemma 9.20 (accept liveness)

1. $L_{\mathrm{H}}^{h'} \models \forall jd : \forall id :$
$\Box(\Box(mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id) \Longrightarrow \Box\Diamond\langle send\_pkt_{rs}(\texttt{accept}, jd, id)\rangle)$

2. $L_{\mathrm{H}}^{h'} \models \forall jd : \forall id : \Box(\Box(mode_r \neq \texttt{rec}) \wedge mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id \Longrightarrow$
$\Diamond\langle receive\_pkt_{sr}(\texttt{send}, \_, id)\rangle \vee$
$\Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle \vee$
$\Box\Diamond\langle send\_pkt_{rs}(\texttt{accept}, jd, id)\rangle)$

Proof

1. Similar to the proof of Lemma 9.18.
2. ASSUME: 1. $\alpha \in L_{\mathrm{H}}^{h'}$
   2. jd and id are arbitrary
   3. α_1 is an arbitrary suffix of α

   PROVE: $\alpha_1 \models \Box(mode_r \neq \texttt{rec}) \wedge mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id \Longrightarrow$
   $\Diamond\langle receive\_pkt_{sr}(\texttt{send}, \_, id)\rangle \vee$
   $\Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle \vee$
   $\Box\Diamond\langle send\_pkt_{rs}(\texttt{accept}, jd, id)\rangle$

   $\langle 1\rangle 1.$ $\alpha_1 \models \Box(mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id) \Longrightarrow \Diamond\langle send\_pkt_{rs}(\texttt{accept}, jd, id)\rangle$

   PROOF: From Part 1 of this lemma, the Assumptions, and Lemma 3.5.

   $\langle 1\rangle 2.$ $\alpha_1 \models \Box(\Box(mode_r \neq \texttt{rec}) \wedge mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id \Longrightarrow$
   $((mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id)\ \mathcal{W}_i$
   $(\langle receive\_pkt_{sr}(\texttt{send}, \_, id)\rangle \vee \langle receive\_pkt_{sr}(\texttt{done}, id)\rangle)))$

PROOF: By Lemma 9.16 Part 2(a), The Assumptions, and Lemma 3.5.

⟨1⟩3. Q.E.D.

PROOF: By ⟨1⟩1, ⟨1⟩2, and Rule **Unl1**.

By Lemma 3.5 the result follows.

■

Lemma 9.21 (rcvd $\leadsto$ ack)

$L_{\mathrm{H}}^{h'} \models \Box(\Box(mode_r \neq \mathtt{rec}) \Longrightarrow (mode_r = \mathtt{rcvd} \leadsto mode_r = \mathtt{ack}))$

Proof

We only sketch this proof. During any live execution of $\mathrm{H}^{h'}$, if the receiver is in rcvd mode and never crashes, then, by the definition of $steps(A_{\mathrm{H}}^{h'})$, the only mode change of the receiver is a mode change to ack in a $receive\_msg(m)$ step that empties buf_r. Furthermore, when $mode_r = \mathtt{rcvd}$ no messages can be put into buf_r (which actually implies that buf_r will always contain zero or one element). Then, by fairness to $receive\_msg(m)$ steps, buf_r will eventually be emptied and hence the result follows.

■

Lemma 9.22 (ack **liveness**)

1. $L_{\mathrm{H}}^{h'} \models \forall id : \Box(\Box(mode_r = \mathtt{ack} \wedge last_r = id) \Longrightarrow \Box\Diamond\langle send\_pkt_{rs}(\mathtt{ack}, id, true)\rangle)$
2. $L_{\mathrm{H}}^{h'} \models \forall id : \Box(\Box(mode_r \neq \mathtt{rec}) \wedge mode_r = \mathtt{ack} \wedge last_r = id \Longrightarrow \Diamond\langle receive\_pkt_{sr}(\mathtt{done}, id)\rangle \vee \Box\Diamond\langle send\_pkt_{rs}(\mathtt{ack}, id, true)\rangle)$

Proof

Similar to the proof of Lemma 9.20.

■

Lemma 9.23 (ack $\leadsto$ idle)

$L_{\mathrm{H}}^{h'} \models \Box(\Box(mode_s \neq \mathtt{rec} \wedge mode_r \neq \mathtt{rec}) \Longrightarrow (mode_r = \mathtt{ack} \leadsto mode_r = \mathtt{idle}))$

Proof

By Lemma 3.5 the following proof suffices.

ASSUME: 1. $\alpha \in L_{\mathrm{H}}^{h'}$
2. α_1 is an arbitrary suffix of α
3. id is arbitrary
4. $\alpha_1 \models \Box(mode_s \neq \mathtt{rec} \wedge mode_r \neq \mathtt{rec})$

PROVE: $\alpha_1 \models mode_r = \mathtt{ack} \leadsto mode_r = \mathtt{idle}$

⟨1⟩1. $\alpha_1 \models \Box(\Box(mode_r \neq \mathtt{rec}) \wedge mode_r = \mathtt{ack} \wedge last_r = id \Longrightarrow \Diamond\langle receive\_pkt_{sr}(\mathtt{done}, id)\rangle \vee \Box\Diamond\langle send\_pkt_{rs}(\mathtt{ack}, id, true)\rangle)$

PROOF:By Lemma 9.22 Part 2, the Assumptions, and Lemma 3.5.

$\langle 1\rangle 2.$ $\alpha_1 \models \Box(\Box(mode_r \neq \texttt{rec}) \wedge mode_r = \texttt{ack} \wedge last_r = id \Longrightarrow$
$\Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle \vee \Box\Diamond\langle receive\_pkt_{rs}(\texttt{ack}, id, true)\rangle)$

PROOF: By $\langle 1\rangle 1$ and Channel Liveness ($Q_{\mathrm{Ch},rs}$).

$\langle 1\rangle 3.$ $\alpha_1 \models \Box(\Box(mode_r \neq \texttt{rec}) \wedge mode_r = \texttt{ack} \wedge last_r = id \Longrightarrow$
$\Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle \vee \Box\Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle)$

PROOF:By $\langle 1\rangle 2$, Lemma 9.19 Part 2, Rule **MP**, and Channel Liveness ($Q_{\mathrm{Ch},sr}$).

$\langle 1\rangle 4.$ $\alpha_1 \models \Box(\Box(mode_r \neq \texttt{rec}) \wedge mode_r = \texttt{ack} \wedge last_r = id \Longrightarrow$
$\Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle)$

PROOF: Directly from $\langle 1\rangle 3$.

$\langle 1\rangle 5.$ $\alpha_1 \models \Box(\Box(mode_r \neq \texttt{rec}) \wedge mode_r = \texttt{ack} \wedge last_r = id \Longrightarrow$
$((mode_r = \texttt{ack} \wedge last_r = id)\, \mathcal{U}_i\, \Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle))$

PROOF: By $\langle 1\rangle 4$, Lemma 9.16 Part 4(a), and the definition of $\mathcal{U}_i$.

$\langle 1\rangle 6.$ $\alpha_1 \models \Box(mode_r = \texttt{ack} \wedge last_r = id \Longrightarrow$
$\Diamond(mode_r = \texttt{ack} \wedge last_r = id \wedge \langle receive\_pkt_{sr}(\texttt{done}, id)\rangle))$

PROOF: By $\langle 1\rangle 5$, The Assumptions, Rule **MP**, and the definition of $\mathcal{U}_i$.

$\langle 1\rangle 7.$ $\alpha_1 \models mode_r = \texttt{ack} \wedge last_r = id \leadsto$
$mode_r = \texttt{ack} \wedge last_r = id \wedge \langle receive\_pkt_{sr}(\texttt{done}, id)\rangle$

PROOF: Directly from $\langle 1\rangle 6$ and the definition of $\leadsto$.

$\langle 1\rangle 8.$ $\alpha_1 \models (mode_r = \texttt{ack} \wedge last_r = id) \leadsto mode_r = \texttt{idle}$

PROOF: By $\langle 1\rangle 7$, the $\leadsto$ property implied by Lemma 9.16 Part 4(b), and transitivity of $\leadsto$.

$\langle 1\rangle 9.$ Q.E.D.

PROOF: Directly from $\langle 1\rangle 8$.

■

We are now ready to state and prove a very important result about the live executions of $\mathrm{H}^{h'}$. In Section 9.2.3 we provided some intuitive justification of the mode of operation of the H protocol. One bad situation that we touched upon was when the sender is in `needid` mode but the receiver is in some “bad” mode other than `idle`. We argued that eventually, due to `done` packets, the receiver would always be reset to `idle` but that it immediately could enter a bad `accept` mode again as a result of receiving an old `needid` packet (i.e., a `needid` packet ($\texttt{needid}, jd$) for which $jd \neq jd_s$) from the channel. However, since each channel step can only add a finite number of packets to a channel, at any point during execution there are only finitely many packets—and consequently only finitely many old `needid` packets—in the *sr* channel. Therefore, since the sender only adds new `needid` packets to *sr*, the receiver can only enter a bad `accept` state finitely many times. Thus, sooner or later either the receiver receives a new `needid` packet (even though there are still old ones in the channel) or all old `needid` packets have been received, in which case the receiver will eventually be reset to `idle` mode and thereafter receive a new `needid` packet. This is formalized in the following lemma. In the proof we use the induction rule **Ind**.

First, we need the following definition: in any state where $mode_s = \texttt{needid}$, define the number of old `needid` packets, written $\#_{old}\texttt{needid}$, to be the number of `needid` packets (including duplicates) in the *sr* channel with $jd \neq jd_s$.

Lemma 9.24
$L_{\mathrm{H}}^{h\,\prime} \models \forall jd : \Box(\Box(mode_s = \texttt{needid} \wedge jd_s = jd \wedge mode_r \neq \texttt{rec}) \Longrightarrow$
$\quad \Diamond(mode_r = \texttt{accept} \wedge jd_r = jd))$

Proof

ASSUME: $\alpha \in L_{\mathrm{H}}^{h\,\prime}$
PROVE: $\alpha \models \forall jd : \Box(\Box(mode_s = \texttt{needid} \wedge jd_s = jd \wedge mode_r \neq \texttt{rec}) \Longrightarrow$
$\quad \Diamond(mode_r = \texttt{accept} \wedge jd_r = jd))$

$\langle 1\rangle 1$. ASSUME: 1. jd is arbitrary
2. α_1 is an arbitrary suffix of α
3. $\alpha_1 \models \Box(mode_s = \texttt{needid} \wedge jd_s = jd \wedge mode_r \neq \texttt{rec})$
PROVE: $\alpha_1 \models \Diamond(mode_r = \texttt{accept} \wedge jd_r = jd)$

$\langle 2\rangle 1$. CASE: $\alpha_1 \models mode_r = \texttt{accept} \wedge jd_r = jd$

$\langle 3\rangle 1$. Q.E.D.

PROOF: Case Assumption $\langle 2\rangle$ implies the goal.

$\langle 2\rangle 2$. CASE: $\alpha_1 \models \neg(mode_r = \texttt{accept} \wedge jd_r = jd)$

$\langle 3\rangle 1$. $\alpha_1 \models \Diamond(mode_r = \texttt{idle})$

$\langle 4\rangle 1$. CASE: $\alpha_1 \models mode_r = \texttt{idle}$

$\langle 5\rangle 1$. Q.E.D.

PROOF: Assumption $\langle 4\rangle$ implies the goal.

$\langle 4\rangle 2$. CASE: $\alpha_1 \models mode_r = \texttt{ack}$

$\langle 5\rangle 1$. Q.E.D.

PROOF: By Assumptions $\langle 4\rangle$ and $\langle 1\rangle$.3, and Lemma 9.23.

$\langle 4\rangle 3$. CASE: $\alpha_1 \models mode_r = \texttt{rcvd}$

$\langle 5\rangle 1$. Q.E.D.

PROOF: By Assumptions $\langle 4\rangle$ and $\langle 1\rangle$.3, and Lemmas 9.21 and 9.23.

$\langle 4\rangle 4$. CASE: $\alpha_1 \models mode_r = \texttt{accept} \wedge jd_r \neq jd$

$\langle 5\rangle 1$. $\alpha_1 \models mode_r = \texttt{accept} \wedge jd_r \neq jd \wedge jd_r = jd' \wedge id_r = id$

PROOF: From Assumption $\langle 4\rangle$ by letting jd' and id be the values of jd_r and id_r, respectively, in the first state of α_1.

$\langle 5\rangle 2$. $\alpha_1 \models \Diamond\langle receive\_pkt_{sr}(\texttt{send}, \_, id)\rangle \vee \Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle \vee$
$\quad \Box\Diamond\langle send\_pkt_{rs}(\texttt{accept}, jd', id)\rangle$

PROOF: By Lemma 9.20 Part 2, Lemma 3.5, $\langle 5\rangle 1$, Assumption $\langle 1\rangle$.3, and Rule **MP**.

$\langle 5\rangle 3$. $\alpha_1 \models \Diamond\langle receive\_pkt_{sr}(\texttt{send}, \_, id)\rangle \vee \Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle \vee$
$\quad \Box\Diamond\langle receive\_pkt_{sr}(\texttt{done}, id)\rangle$

PROOF: By $\langle 5\rangle 2$, Channel Liveness ($Q_{\mathrm{Ch},sr}$ and $Q_{\mathrm{Ch},rs}$), Lemma 9.19 Part 3, the Assumptions, Lemma 3.5, and Rule 3.5.

$\langle 5\rangle 4.$ $\alpha_1 \models \Diamond\langle receive\_pkt_{sr}(\mathtt{send}, \_, id)\rangle \vee \Diamond\langle receive\_pkt_{sr}(\mathtt{done}, id)\rangle$

PROOF: Directly by $\langle 5\rangle 3$.

$\langle 5\rangle 5.$ $\alpha_1 \models mode_r = \mathtt{accept} \wedge jd_r = jd' \wedge id_r = id\ \mathcal{U}_i$
$\langle receive\_pkt_{sr}(\mathtt{send}, \_, id)\rangle \vee \langle receive\_pkt_{sr}(\mathtt{done}, id)\rangle$

PROOF: By $\langle 5\rangle 4$, Lemma 9.16 Part 2(a), Lemma 3.5, the Assumptions, and Rule **MP**.

$\langle 5\rangle 6.$ $\alpha_1 \models \Diamond(mode_r = \mathtt{accept} \wedge jd_r = jd' \wedge id_r = id \wedge$
$\langle receive\_pkt_{sr}(\mathtt{send}, \_, id)\rangle) \vee$
$\Diamond(mode_r = \mathtt{accept} \wedge jd_r = jd' \wedge id_r = id \wedge$
$\langle receive\_pkt_{sr}(\mathtt{done}, id)\rangle)$

PROOF: Implied by $\langle 5\rangle 5$.

$\langle 5\rangle 7.$ $\alpha_1 \models \Diamond(mode_r = \mathtt{rcvd}) \vee \Diamond(mode_r = \mathtt{idle})$

PROOF: By $\langle 5\rangle 6$, Lemma 9.16 Part 2(b), the Assumptions, Lemma 3.5, and Rule **MP**.

$\langle 5\rangle 8.$ Q.E.D.

PROOF: By $\langle 5\rangle 7$, Lemmas 9.21 and 9.23, and the Assumptions.

$\langle 4\rangle 5.$ Q.E.D.

PROOF: By Assumption $\langle 2\rangle$ and the exhaustive cases $\langle 4\rangle 1$–$\langle 4\rangle 4$.

$\langle 3\rangle 2.$ $\alpha_1 \models \Box(\#_{old}\mathtt{needid}^\circ \leq \#_{old}\mathtt{needid})$

PROOF: By Assumption $\langle 1\rangle.3$, $\#_{old}\mathtt{needid}$ is defined in all states of α_1 and jd_s does not change in α_1. Then, since the only actions that can add $\mathtt{needid}$ packets to sr add packets with $jd \neq jd_s$, the result follows.

$\langle 3\rangle 3.$ Base Case
$\alpha_1 \models (mode_r = \mathtt{idle} \wedge \#_{old}\mathtt{needid} = 0) \leadsto (mode_r = \mathtt{accept} \wedge jd_r = jd)$

$\langle 4\rangle 1.$ ASSUME: 1. α_2 is an arbitrary suffix of α_1
2. $\alpha_2 \models mode_r = \mathtt{idle} \wedge \#_{old}\mathtt{needid} = 0$
PROVE: $\alpha_2 \models \Diamond(mode_r = \mathtt{accept} \wedge jd_r = jd)$

$\langle 5\rangle 1.$ $\alpha_2 \models \Box(\#_{old}\mathtt{needid} = 0)$

PROOF: By $\langle 3\rangle 2$ and Assumption $\langle 4\rangle.2$.

$\langle 5\rangle 2.$ $\alpha_2 \models \Box\neg\langle\{receive\_pkt_{sr}(\mathtt{needid}, jd') \mid jd' \neq jd\}\rangle$

PROOF: By $\langle 5\rangle 1$, Assumption $\langle 1\rangle.2$, Lemma 3.5 Part 1, and the definition of the steps of $A_{\mathrm{H}}^{h\,\prime}$.

$\langle 5\rangle 3.$ $\alpha_2 \models mode_r = \mathtt{idle}\ \mathcal{W}_i\ \langle receive\_pkt_{sr}(\mathtt{needid}, \_)\rangle$

PROOF: From Lemma 3.5 Part 1, the fact that α_2 is a suffix of α (Assumptions $\langle 1\rangle.2$ and $\langle 4\rangle.1$), Lemma 9.16 Part 1(a), Assumptions $\langle 1\rangle.3$ and $\langle 4\rangle.2$, and Rule **MP**.

$\langle 5\rangle 4.$ $\alpha_2 \models mode_r = \mathtt{idle}\ \mathcal{W}_i\ \langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle$

PROOF: By $\langle 5\rangle 2$ and $\langle 5\rangle 3$.

$\langle 5\rangle 5.$ $\alpha_2 \models \Diamond\langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle$

PROOF: From Lemma 9.18, Channel Liveness $Q_{\mathrm{Ch},sr}$, Assumption $\langle 1\rangle.3$, and Rule **MP**.

$\langle 5\rangle 6.$ $\alpha_2 \models mode_r = \mathtt{idle}\,\mathcal{U}_i\ \langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle$

PROOF: By $\langle 5\rangle 4$, $\langle 5\rangle 5$, and the definition of $\mathcal{U}_i$.

$\langle 5\rangle 7.$ $\alpha_2 \models \Diamond(mode_r = \mathtt{idle} \wedge \langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle)$

PROOF: By $\langle 5\rangle 6$ and the definition of $\mathcal{U}_i$.

$\langle 5\rangle 8.$ Q.E.D.

PROOF: By $\langle 5\rangle 7$, Lemma 9.16 Part 1(b), and **MP**1 (and, as always, Lemma 3.5 Part 1 and the assumption that α_2 is a suffix of α).

$\langle 4\rangle 2.$ Q.E.D.

PROOF: $\langle 3\rangle 3$, the definition of implication, and Lemma 3.5 Part 2 gives $\alpha_1 \models \Box(mode_r = \mathtt{idle} \wedge \#_{old}\mathtt{needid} = 0 \Longrightarrow \Diamond(mode_r = \mathtt{accept} \wedge jd_r = jd))$ which, by definition of $\leadsto$, immediately gives the result.

$\langle 3\rangle 4.$ Inductive Case

$$\begin{aligned}\alpha_1 \models \forall k : (k > 0 \Longrightarrow\\ &\exists l : (l < k \wedge\\ &\quad (mode_r = \mathtt{idle} \wedge \#_{old}\mathtt{needid} = k \leadsto\\ &\quad\ (mode_r = \mathtt{idle} \wedge \#_{old}\mathtt{needid} = l) \vee\\ &\quad\ (mode_r = \mathtt{accept} \wedge jd_r = jd))))\end{aligned}$$

$\langle 4\rangle 1.$ ASSUME: 1. k is an arbitrary positive number
2. α_2 is an arbitrary suffix of α_1
3. $\alpha_2 \models mode_r = \mathtt{idle} \wedge \#_{old}\mathtt{needid} = k$

PROVE: $\alpha_2 \models \Diamond((mode_r = \mathtt{idle} \wedge \#_{old}\mathtt{needid} < k) \vee (mode_r = \mathtt{idle} \wedge jd_r = jd))$

$\langle 5\rangle 1.$ $\alpha_2 \models mode_r = \mathtt{idle}\,\mathcal{W}_i\ (\langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle \vee \langle\{receive\_pkt_{sr}(\mathtt{needid}, jd') \mid jd' \neq jd\}\rangle)$

PROOF: By Lemma 9.16 Part 1(a), Assumptions $\langle 1\rangle.3$ and $\langle 4\rangle.3$, and Rule **MP**.

$\langle 5\rangle 2.$ $\alpha_2 \models \Diamond\langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle$

PROOF: By Lemma 9.18, Assumption $\langle 1\rangle.3$, Rule **MP**, and Channel Liveness $Q_{\mathrm{Ch},sr}$.

$\langle 5\rangle 3.$ $\alpha_2 \models mode_r = \mathtt{idle}\,\mathcal{U}_i\ (\langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle \vee \langle\{receive\_pkt_{sr}(\mathtt{needid}, jd') \mid jd' \neq jd\}\rangle)$

PROOF: By $\langle 5\rangle 1$, $\langle 5\rangle 2$, and the definition of $\mathcal{U}_i$.

$\langle 5\rangle 4.$ $\alpha_2 \models \Diamond(mode_r = \mathtt{idle} \wedge \langle receive\_pkt_{sr}(\mathtt{needid}, jd)\rangle) \vee \Diamond(mode_r = \mathtt{idle} \wedge \langle\{receive\_pkt_{sr}(\mathtt{needid}, jd') \mid jd' \neq jd\}\rangle \wedge \#_{old}\mathtt{needid} \leq k)$

PROOF: By $\langle 5\rangle 3$, the definition of $\mathcal{U}_i$, Assumption $\langle 4\rangle.3$, and $\langle 3\rangle 2$.

$\langle 5\rangle 5.$ $\alpha_2 \models \Diamond(mode_r = \texttt{accept} \wedge jd_r = jd) \vee$
$\Diamond(mode_r = \texttt{accept} \wedge jd_r \neq jd \wedge \#_{old}\texttt{needid} < k)$

PROOF: By $\langle 5\rangle 4$, Lemma 9.16 Part 1(b) and the fact that receiving an old `needid` packet reduces $\#_{old}\texttt{needid}$ by one.

$\langle 5\rangle 6.$ $\Diamond(mode_r = \texttt{accept} \wedge jd_r = jd) \vee$
$\Diamond(mode_r = \texttt{idle} \wedge \#_{old}\texttt{needid} < k)$

PROOF: Similar to Case $\alpha_1 \models (mode_r = \texttt{accept} \wedge jd_r \neq jd)$ of $\langle 3\rangle 1$ above (and $\langle 3\rangle 2$).

$\langle 5\rangle 7.$ Q.E.D.

PROOF: Directly from $\langle 5\rangle 6$.

$\langle 4\rangle 2.$ Q.E.D.

PROOF: From $\langle 4\rangle 1$, The definition of $\leadsto$, and Lemma 3.5.

$\langle 3\rangle 5.$ $\alpha_1 \models \forall n : \Box(mode_r = \texttt{idle} \wedge \#_{old}\texttt{needid} = n \Longrightarrow$
$\Diamond(mode_r = \texttt{accept} \wedge jd_r = jd))$

PROOF: By $\langle 3\rangle 3$, $\langle 3\rangle 4$, Rule **Ind**, and the definition of $\leadsto$.

$\langle 3\rangle 6.$ For some number n',
$\alpha_1 \models \Diamond(mode_r = \texttt{idle} \wedge \#_{old}\texttt{needid} = n')$

PROOF: Directly from $\langle 3\rangle 1$ when we let n' be the value of $\#_{old}\texttt{needid}$ in some state of α_1 where $mode_r = \texttt{idle}$.

$\langle 3\rangle 7.$ $\alpha_1 \models \Box(mode_r = \texttt{idle} \wedge \#_{old}\texttt{needid} = n' \Longrightarrow$
$\Diamond(mode_r = \texttt{accept} \wedge jd_r = jd))$

PROOF: By $\langle 3\rangle 5$ and Lemma 3.5 Part 6.

$\langle 3\rangle 8.$ Q.E.D.

PROOF: By $\langle 3\rangle 6$, $\langle 3\rangle 7$, and Rule **MP**1.

$\langle 2\rangle 3.$ Q.E.D.

PROOF: By the exhaustive cases $\langle 2\rangle 1$ and $\langle 2\rangle 2$.

$\langle 1\rangle 2.$ Q.E.D.

PROOF: By $\langle 1\rangle 1$ using the definition of implication and Lemma 3.5 Parts 2 and 5.

■

Now, since the receiver will eventually enter `accept` mode with the right jd_r, eventually the sender will receive a $(\texttt{accept}, jd_s, id)$ packet as formalized by the following lemma.

Lemma 9.25

$$L_{\mathrm{H}}^{h'} \models \forall jd : \Box(\Box(mode_s = \texttt{needid} \wedge jd_s = jd \wedge mode_r \neq \texttt{rec}) \Longrightarrow$$
$$\Diamond\langle receive\_pkt_{rs}(\texttt{accept}, jd, \_)\rangle)$$

Proof

ASSUME: $\alpha \in L_{\mathrm{H}}^{h'}$
PROVE: $\alpha \models \forall jd : \Box(\Box(mode_s = \texttt{needid} \wedge jd_s = jd \wedge mode_r \neq \texttt{rec}) \Longrightarrow$

$\Diamond(\langle receive\_pkt_{rs}(\texttt{accept}, jd, \_)\rangle))$

$\langle 1\rangle 1.$ ASSUME: 1. jd is arbitrary
2. α_1 is an arbitrary suffix of α
3. $\alpha_1 \models \Box(mode_s = \texttt{needid} \wedge jd_s = jd \wedge mode_r \neq \texttt{rec})$

PROVE: $\alpha_1 \models \Diamond\langle receive\_pkt_{rs}(\texttt{accept}, jd, \_)\rangle$

$\langle 2\rangle 1.$ $\alpha_1 \models \Diamond(mode_r = \texttt{accept} \wedge jd_r = jd)$

PROOF: By Lemma 9.24, Assumption $\langle 1\rangle$, Lemma 3.5, and Rule **MP**.

$\langle 2\rangle 2.$ ASSUME: 1. α_2 is a suffix of α_1 such that
2. $\alpha_2 \models mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id$

PROVE: $\alpha_2 \models \Diamond\langle receive\_pkt_{rs}(\texttt{accept}, jd, \_)\rangle$

$\langle 3\rangle 1.$ $\alpha_2 \models (mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id)\ \mathcal{W}_i$
$(\langle receive\_pkt_{sr}(\texttt{send}, \_, id)\rangle \vee \langle receive\_pkt_{sr}(\texttt{done}, id)\rangle)$

PROOF: By Lemma 9.16 Part 2(a), Lemma 3.5, Assumptions $\langle 1\rangle$ and $\langle 2\rangle$, and Rule **MP**.

$\langle 3\rangle 2.$ $\alpha_2 \models \Box(mode_r = \texttt{accept} \wedge jd_r = jd \wedge id_r = id)$

PROOF: By $\langle 3\rangle 1$, Lemma 9.17, Lemma 3.5, and Rule **Unl**.

$\langle 3\rangle 3.$ $\alpha_2 \models \Box\Diamond\langle send\_pkt_{rs}(\texttt{accept}, jd, id)\rangle$

PROOF: By $\langle 3\rangle 2$, Lemma 9.20 Part 1, Lemma 3.5, and Rule **MP**.

$\langle 3\rangle 4.$ $\alpha_2 \models \Box\Diamond\langle receive\_pkt_{rs}(\texttt{accept}, jd, id)\rangle$

PROOF: The form of $Q_{\mathrm{Ch},rs}$ implies that since $\alpha \models Q_{\mathrm{Ch},rs}$ (α is live) and α_2 is a suffix of α, then $\alpha_2 \models Q_{\mathrm{Ch},rs}$. This and $\langle 3\rangle 3$ together with Rule **MP** give the result.

$\langle 3\rangle 5.$ Q.E.D.

PROOF: Directly from $\langle 3\rangle 4$.

$\langle 2\rangle 3.$ Q.E.D.

PROOF: By $\langle 2\rangle 1$ and $\langle 2\rangle 2$.

$\langle 1\rangle 2.$ Q.E.D.

PROOF: By $\langle 1\rangle 1$, the definition of implication, and Lemma 3.5.

■

Lemma 9.26

$$A_{\mathrm{H}}^{h'} \models \Box(\Box(mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec}) \Longrightarrow \Diamond(mode_s = \texttt{send}))$$

Proof

Directly from Lemma 9.25 and Lemma 9.15 Part 2(b).

■

We are now ready to prove the main part of the liveness proof that $\mathrm{H}^{h'}$ correctly implements $\mathrm{G}^{\rho'}$, namely, if α is a live execution of $\mathrm{H}^{h'}$ and α' is an execution of $\mathrm{G}^{\rho'}$ such that $(\alpha, \alpha') \in R_{\mathrm{HG}}$, then α' is live. As usual, we prove this result by contradiction. Thus, we assume that α' is not live and then derive a contradiction with the fact that α is live.

Lemma 9.27

Let $\alpha \in exec(A_{\mathrm{H}}^{h\,\prime})$ and $\alpha' \in exec(A_{\mathrm{G}}^{\rho\,\prime})$ be arbitrary executions of $A_{\mathrm{H}}^{h\,\prime}$ and $A_{\mathrm{G}}^{\rho\,\prime}$, respectively, with $(\alpha, \alpha') \in R_{\mathrm{HG}}$. Assume $\alpha \models Q_{\mathrm{H}}$. Then $\alpha' \models \rho(Q_{\mathrm{G}})$.

Proof

We prove the conjecture by contradiction. Thus,

ASSUME: $\alpha' \not\models \rho(Q_{\mathrm{G}})$
PROVE: False

$\langle 1\rangle 1.$ $\alpha' \models \neg WF(\rho(C_{\mathrm{G},s/r1})) \vee$
$\neg\Box(\Box(mode_s = \mathtt{needid} \wedge mode_r \neq \mathtt{rec}) \Longrightarrow \Diamond\langle\rho(C_{\mathrm{G},s/r2})\rangle) \vee$
$\neg WF(\rho(C_{\mathrm{G},s/r3})) \vee$
$\neg WF(\rho(C_{\mathrm{G},s/r4})) \vee$
$\neg\forall p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{sr}(p)\rangle) \vee$
$\neg\forall p : WF(receive\_pkt_{sr}(p)) \vee$
$\neg\forall p : (\Box\Diamond\langle send\_pkt_{rs}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{rs}(p)\rangle) \vee$
$\neg\forall p : WF(receive\_pkt_{sr}(p))$

PROOF: Immediate by the Assumption, definition of $\rho(Q_{\mathrm{G}})$, and the Boolean operators.

$\langle 1\rangle 2.$ CASE: $\alpha' \models \neg WF(\rho(C_{\mathrm{G},s/r1}))$

$\langle 2\rangle 1.$ $\alpha' \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}) \wedge \Diamond\Box\neg\langle\rho(C_{\mathrm{G},s/r1})\rangle$

PROOF: From Case Hypothesis $\langle 1\rangle$ by noting that $enabled(\rho(C_{\mathrm{G},s/r1})) = (mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\})$ and by expanding WF.

$\langle 2\rangle 2.$ $\alpha \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}) \wedge \Diamond\Box\neg\langle\rho(C_{\mathrm{G},s/r1}) \setminus \{prepare\}\rangle$

PROOF: From $\langle 2\rangle 1$ by definition of R_{HG} and by Lemmas 5.10 and 5.11.

$\langle 2\rangle 3.$ $\alpha \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}) \wedge$
$\Diamond\Box\neg\langle\rho(C_{\mathrm{G},s/r1}) \setminus \{prepare\}\rangle \wedge$
$\Diamond\Box\neg\langle\{send\_pkt_{sr}(\mathtt{needid}, jd) \mid jd \in JD\}\rangle$

PROOF: By $\langle 2\rangle 2$ there is a suffix of α where always $mode_s \in \{\mathtt{idle}, \mathtt{rec}, \mathtt{send}\}$. Thus we get that no $send\_pkt_{sr}(\mathtt{needid}, \_)$ actions occur in that suffix, since such actions are only enabled when $mode_s = \mathtt{needid}$.

$\langle 2\rangle 4.$ $\alpha \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}, \mathtt{needid}\}) \wedge$
$\Diamond\Box\neg\langle(\rho(C_{\mathrm{G},s/r1}) \setminus \{prepare\}) \cup \{send\_pkt_{sr}(\mathtt{needid}, jd) \mid jd \in JD\}\rangle$

PROOF: By $\langle 2\rangle 3$ by noting that if $mode_s$ is in $\{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}$, it is also in the bigger set $\{\mathtt{idle}, \mathtt{send}, \mathtt{rec}, \mathtt{needid}\}$.

$\langle 2\rangle 5.$ $\alpha \models \neg WF(C_{\mathrm{H},s1})$

PROOF: From $\langle 2\rangle 4$ by using the definitions of WF and $C_{\mathrm{H},s1}$.

$\langle 2\rangle 6.$ Q.E.D.

PROOF: $\langle 2\rangle 5$ contradicts the assumption that $\alpha \models Q_{\mathrm{H}}$.

$\langle 1 \rangle 3$. CASE: $\alpha' \models \neg\Box(\Box(mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec}) \Longrightarrow \Diamond\langle \rho(C_{\mathrm{G},s/r2})\rangle)$

$\langle 2 \rangle 1$. $\alpha' \models \Diamond(\Box(mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec}) \wedge \Box\neg\langle \rho(C_{\mathrm{G},s/r2})\rangle)$

PROOF: Directly from Assumption $\langle 1 \rangle$.

$\langle 2 \rangle 2$. $\alpha' \models \Diamond\Box(mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec}) \wedge \Diamond\Box\neg\langle \rho(C_{\mathrm{G},s/r2})\rangle$

PROOF: Directly from $\langle 2 \rangle 1$.

$\langle 2 \rangle 3$. $\alpha \models \Diamond\Box(mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec})$

PROOF: From $\langle 2 \rangle 2$ by Lemma 5.11 and the definition of R_{HG}.

$\langle 2 \rangle 4$. There exists a suffix α_1 of α such that
$\alpha_1 \models \Box(mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec})$

PROOF: From $\langle 2 \rangle 3$ using Lemma 3.5 Part 3.

$\langle 2 \rangle 5$. $\alpha_1 \models \Box(mode_s = \texttt{needid} \wedge mode_r \neq \texttt{rec}) \Longrightarrow \Diamond(mode_s = \texttt{send})$

PROOF: By Lemma 9.26, Lemma 3.5 Part 1, and Rule **Par**.

$\langle 2 \rangle 6$. $\alpha_1 \models \Diamond(mode_s = \texttt{send})$

PROOF: By $\langle 2 \rangle 4$, $\langle 2 \rangle 5$, and Rule **MP**.

$\langle 2 \rangle 7$. Q.E.D.

PROOF: $\langle 2 \rangle 6$ contradicts $\langle 2 \rangle 4$.

$\langle 1 \rangle 4$. CASE: $\alpha' \models \neg\, WF(\rho(C_{\mathrm{G},s/r3}))$

$\langle 2 \rangle 1$. $\alpha' \models \Diamond\Box(mode_r = \texttt{rec} \vee (mode_r = \texttt{rcvd} \wedge buf_r \neq \varepsilon) \vee mode_r = \texttt{ack}) \wedge$
$\Diamond\Box\neg\langle \rho(C_{\mathrm{G},s/r3})\rangle$

PROOF: By Assumption $\langle 1 \rangle$ and the definitions of WF and $enabled(\rho(C_{\mathrm{G},s/r3}))$.

$\langle 2 \rangle 2$. $\alpha \models \Diamond\Box(mode_r = \texttt{rec} \vee (mode_r = \texttt{rcvd} \wedge buf_r \neq \varepsilon) \vee mode_r = \texttt{ack}) \wedge$
$\Diamond\Box\neg\langle \rho(C_{\mathrm{G},s/r3})\rangle$

PROOF: From $\langle 2 \rangle 1$ by definition of R_{HG}, the fact that $\rho(C_{\mathrm{G},s/r3})$ contains external actions only, and Lemmas 5.10 and 5.11.

$\langle 2 \rangle 3$. $\alpha \models \Diamond\Box(mode_r = \texttt{rec} \vee (mode_r = \texttt{rcvd} \wedge buf_r \neq \varepsilon) \vee mode_r = \texttt{ack}) \wedge$
$\Diamond\Box\neg\langle \rho(C_{\mathrm{G},s/r3})\rangle \wedge$
$\Diamond\Box\neg\langle \{send\_pkt_{rs}(\texttt{accept}, jd, id) \mid jd \in JD \wedge id \in ID\}\rangle$

PROOF: Since, by $\langle 2 \rangle 2$, there is a suffix of α where always $mode_r \in \{\texttt{rec}, \texttt{rcvd}, \texttt{ack}\}$ we get that no $send\_pkt_{rs}(\texttt{accept}, \_, \_)$ actions occur in that suffix, since such actions are only enabled when $mode_r = \texttt{accept}$.

$\langle 2 \rangle 4$. $\alpha \models \Diamond\Box((mode_r = \texttt{rcvd} \wedge buf_r \neq \varepsilon) \vee mode_r \in \{\texttt{rec}, \texttt{ack}, \texttt{accept}\}) \wedge$
$\Diamond\Box\neg\langle \rho(C_{\mathrm{G},s/r3}) \cup \{send\_pkt_{rs}(\texttt{accept}, jd, id) \mid jd \in JD \wedge id \in ID\}\rangle$

PROOF: By $\langle 2 \rangle 3$ by noting that if eventually $mode_r$ is always in $\{\texttt{rec}, \texttt{rcvd}, \texttt{ack}\}$, then it is eventually always in the bigger set $\{\texttt{rec}, \texttt{rcvd}, \texttt{ack}, \texttt{accept}\}$.

$\langle 2 \rangle 5$. $\alpha \models \neg\, WP(C_{\mathrm{H},r1})$

PROOF: By $\langle 2 \rangle 4$ using the definition of WF and the fact that $C_{\mathrm{H},r1} = \rho(C_{\mathrm{G},s/r3}) \cup \{send\_pkt_{rs}(\texttt{accept}, jd, id) \mid jd \in JD \wedge id \in ID\}$.

$\langle 2 \rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 5$ contradicts the assumption that $\alpha \models Q_{\mathrm{H}}$.

$\langle 1\rangle 5$. CASE: $\alpha' \models \neg\, WF(\rho(C_{\mathrm{G},s/r4}))$

$\langle 2\rangle 1$. $\alpha' \models \Diamond\Box(mode_r \neq \mathtt{rec} \wedge nack\text{-}buf_r \neq \varepsilon) \wedge \Diamond\Box\neg\langle\rho(C_{\mathrm{G},s/r4})\rangle$

PROOF: From Assumption $\langle 1\rangle$ by using the definition of WF, and the fact that $enabled(\rho(C_{\mathrm{G},s/r4})) = (mode_r \neq \mathtt{rec} \wedge nack\text{-}buf_r \neq \varepsilon)$.

$\langle 2\rangle 2$. $\alpha \models \Diamond\Box(mode_r \neq \mathtt{rec} \wedge nack\text{-}buf_r \neq \varepsilon) \wedge \Diamond\Box\neg\langle\rho(C_{\mathrm{G},s/r4})\rangle$

PROOF: By $\langle 2\rangle 1$, the definition of R_{HG}, the fact that $\rho(C_{\mathrm{G},s/r4})$ consists of external actions only, and Lemmas 5.10 and 5.11.

$\langle 2\rangle 3$. $\alpha \models \neg\, WF(C_{\mathrm{H},r2})$

PROOF: By $\langle 2\rangle 2$ using the definition of WF and the fact that $C_{\mathrm{H},r2} = \rho(C_{\mathrm{G},s/r4})$.

$\langle 2\rangle 4$. Q.E.D.

PROOF: $\langle 2\rangle 3$ contradicts the assumption that $\alpha \models Q_{\mathrm{H}}$.

$\langle 1\rangle 6$. CASE: $\alpha' \models \neg\forall p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{sr}(p)\rangle)$

$\langle 2\rangle 1$. $\alpha' \models \exists p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle)$

PROOF: Directly from Assumption $\langle 1\rangle$.

$\langle 2\rangle 2$. There exists $m \in Msg$ and $id \in ID$ such that
$\alpha' \models \Box\Diamond\langle send\_pkt_{sr}(\mathtt{send}, m, id)\rangle \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(\mathtt{send}, m, id)\rangle$

PROOF: By $\langle 2\rangle 1$ and Lemma 3.5 Part 8.

$\langle 2\rangle 3$. $\alpha \models \Box\Diamond\langle send\_pkt_{sr}(\mathtt{send}, m, id)\rangle \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(\mathtt{send}, m, id)\rangle$

PROOF: By $\langle 2\rangle 2$, Lemma 5.10, and the fact that the actions $send\_pkt_{sr}(\mathtt{send}, m, id)$ and $receive\_pkt_{sr}(\mathtt{send}, m, id)$ are external.

$\langle 2\rangle 4$. $\alpha \models \exists p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle)$

PROOF: By $\langle 2\rangle 3$ and Lemma 3.5 Part 7. (Note that the bound variable p ranges over all packets of the form $(\mathtt{needid}, id)$, $(\mathtt{send}, m, id)$, and $(\mathtt{done}, id)$, whereas the bound variable in $\langle 2\rangle 1$ only ranges over packets of the form $(\mathtt{send}, m, id)$.)

$\langle 2\rangle 5$. $\alpha \models \neg\forall p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{sr}(p)\rangle)$

PROOF: Directly from $\langle 2\rangle 4$.

$\langle 2\rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 5$ contradicts the assumption that $\alpha \models Q_{\mathrm{H}}$.

$\langle 1\rangle 7$. CASE: $\alpha' \models \neg\forall p : WF(receive\_pkt_{sr}(p))$

$\langle 2\rangle 1$. $\alpha' \models \exists p : \neg\, WF(receive\_pkt_{sr}(p))$

PROOF: Directly from Assumption $\langle 1\rangle$.

$\langle 2\rangle 2$. For some packet p (of the form $(\mathtt{send}, m, id)$),
$\alpha' \models \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle \wedge \Diamond\Box(p \in sr)$

PROOF: By $\langle 2\rangle 1$, Lemma 3.5 Part 8, the definition of WF and since $receive\_pkt_{sr}(p)$ is enabled when $p \in sr$.

$\langle 2\rangle 3$. $\alpha \models \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle \wedge \Diamond\Box(p \in sr)$

PROOF: By $\langle 2\rangle 2$, Lemmas 5.10 and 5.11, and the facts that $receive\_pkt_{sr}(p)$ is external, and if $(s, u) \in R_{\mathrm{HG}}$ and $u \models (p \in sr)$, then $s \models (p \in sr)$ (recall that p has the form $(\mathtt{send}, m, id)$).

$\langle 2\rangle 4$. $\alpha \models \neg\forall p : WF(receive\_pkt_{sr}(p))$

PROOF: Directly from $\langle 2\rangle 3$, Lemma 3.5 Part 7 and the definition of WF.

$\langle 2\rangle 5$. Q.E.D.

PROOF: $\langle 2\rangle 4$ contradicts the assumption that $\alpha \models Q_{\mathrm{H}}$.

$\langle 1\rangle 8$. CASE: $\alpha' \models \neg\forall p : (\Box\Diamond\langle send\_pkt_{rs}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{rs}(p)\rangle)$

PROOF: Similar to $\langle 1\rangle 6$.

$\langle 1\rangle 9$. CASE: $\alpha' \models \neg\forall p : WF(receive\_pkt_{rs}(p))$

PROOF: Similar to $\langle 1\rangle 7$.

$\langle 1\rangle 10$. Q.E.D.

PROOF: By $\langle 1\rangle 1$ and the exhaustive cases $\langle 1\rangle 2$–$\langle 1\rangle 9$.

■

With this result, the simulation result of the previous section, and Lemma 5.9 we can prove that $\mathrm{H}^{h'}$ correctly implements $\mathrm{G}^{\rho'}$.

Lemma 9.28

$\mathrm{H}^{h'} \sqsubseteq_{\mathrm{L}} \mathrm{G}^{\rho'}$

Proof

Immediate by Lemmas 9.13, 9.27, and 5.9.

■

And, finally, we can prove that H correctly implements G.

Theorem 9.29

$\mathrm{H} \sqsubseteq_{\mathrm{L}} \mathrm{G}$

Proof

By Lemma 9.28 and Lemma 5.15 we get
$\mathrm{H}' \sqsubseteq_{\mathrm{L}} \mathrm{G}^{\rho'}$
which by substitutivity (Lemma 2.16) implies
$\mathrm{H}' \setminus \mathcal{A}_{\mathrm{H}} \sqsubseteq_{\mathrm{L}} \mathrm{G}^{\rho'} \setminus \mathcal{A}_{\mathrm{H}}$
Then, by the definition of ρ, $\mathcal{A}_{\mathrm{H}}$, and $\mathcal{A}_{\mathrm{G}}$ we get
$\mathrm{H}' \setminus \mathcal{A}_{\mathrm{H}} \sqsubseteq_{\mathrm{L}} \mathrm{G}^{\rho'} \setminus \rho(\mathcal{A}_{\mathrm{G}})$
Now, since ρ only renames actions which are subsequently hidden, this implies
$\mathrm{H}' \setminus \mathcal{A}_{\mathrm{H}} \sqsubseteq_{\mathrm{L}} \mathrm{G}' \setminus \mathcal{A}_{\mathrm{G}}$
which finally, by definition, yields the result
$\mathrm{H} \sqsubseteq_{\mathrm{L}} \mathrm{G}$

■

Due to the fact that the correct implementation relation $\sqsubseteq_{\mathrm{L}}$ is a preorder, we get the overall result that H correctly implements S and thus solves the at-most-once message delivery problem.

Theorem 9.30

$\mathrm{H} \sqsubseteq_{\mathrm{L}} \mathrm{S}$

Proof

By Theorems 7.18, 8.19, and 9.29, and the fact that the subset relation, and thus the correct implementation relation (cf. Definition 2.15), is transitive.

■

We now move to the timed setting to consider the Clock-Based Protocol C.

Chapter 10

The Clock-Based Protocol C

The second and last low-level protocol we consider in this work is the Clock-Based Protocol of [LSW91], which in this work is denoted by C. As the name suggests the functionality of the protocol depends on the sender and receiver having access to certain clocks. Specifically, the sender and the receiver each has a local clock which is required to deviate from real time by at most some constant amount, called the *clock skew*. The C protocol thus consists of a sender, a receiver, two channels, and a special *clock subsystem* that guarantees that the local clocks are almost synchronized with real time. This structure is depicted in Figure 10.1. We model the clock subsystem as a live timed I/O automaton that issues *tick*s to the sender and the receiver. Exactly how to implement a clock subsystem in a distributed system falls outside the scope of this work [LMS85].

C is a timed protocol. Besides having the clock subsystem, we shall assume that channel delays and the maximum time difference between certain process steps are bounded. Thus, each component of C is specified as a live timed I/O automaton, and consequently C itself is a live timed I/O automaton.

The specification S is modeled as an (untimed) live I/O automaton since the problem statement did not mention time at all. In Section 2.3 we discussed what it means to implement an untimed specification by a timed implementation. The idea was to to consider the untimed specification as a timed system that allows tome to pass arbitrarily as long as possible liveness assumptions are satisfied. For this reason the operator *patient* on safe and live I/O automata was introduced.

We could have removed all liveness assumptions from C and used timing assumptions instead. However, then it would have been difficult to see which timing requirements were actually needed to guarantee the correctness of C and which were just additional timing requirements. Thus, we introduce the minimum timing requirements and otherwise use liveness to guarantee the progress of the system. This means that all external actions of C, which are subject to liveness requirements in S, will be given liveness requirements in C, whereas certain internal actions, like channel communication, will be given timing requirements. With this approach we cannot, of course, prove any maximum response time on, e.g., acknowledgements $ack(b)$ but if such a response time is important, it should have been specified in S. Instead S just assumes that the final implementation is "fast enough".

The rest of the chapter is organized as follows. First, in Section 10.1, we present the clock subsystem. In Section 10.2 we specify timed versions of the channels. Then, in Section 10.3, we specify the sender and receiver and furthermore intuitively describe how the C protocol works.

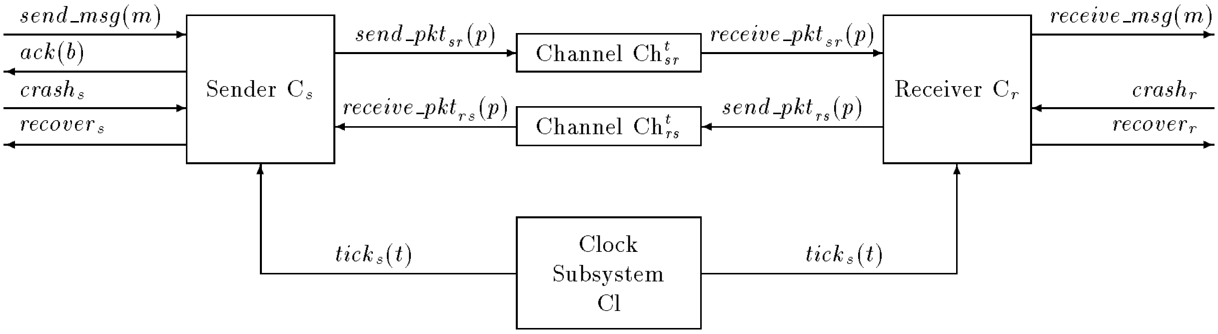


Figure 10.1

The Clock-Based Protocol C.

Section 10.4 shows how C is obtained from its subprocesses and Section 10.5 then considers the correctness of C. Section 10.6 discusses a "weak" version of C, where the timing assumptions are removed, and finally Section 10.7 considers a version of C that works for a single receiver but multiple senders.

10.1 The Clock Subsystem

The clock subsystem is specified as a live timed I/O automaton $\mathrm{Cl} = (A_{\mathrm{Cl}}, L_{\mathrm{Cl}})$. We use the explicit specification style (cf. Section 4.2.1) to specify A_{Cl} and specify L_{Cl} by an environment-free timed liveness formula Q_{Cl} for A_{Cl}.

10.1.1 States and Start States

A_{Cl} contains three state variables: *now* is as usual real time (ranging over T which equals the nonnegative real number), and $ctime_s$ and $ctime_s$ remember the last clock value sent to the sender and receiver, respectively.

Variable	Type	Initially	Description
now	T	0	Real time
$ctime_s$	T	0	Last clock value sent to the sender.
$ctime_r$	T	0	Last clock value sent to the receiver.

10.1.2 Actions

Input:
  none
Output:
  $tick_s(t)$, $t \in \mathsf{T}$
  $tick_r(t)$, $t \in \mathsf{T}$
Internal:
  none

Time-passage:
ν

10.1.3 Steps

The clock subsystem is responsible just for performing outputs of the form $tick_s(t)$ and $tick_r(t)$. This clock subsystem is constrained to produce ticks that have the property that, at any real time now, the most recent tick at either station has value within ϵ of now. Thus, ϵ, which is positive, denotes the clock skew. In addition, each local clock is nondecreasing, that is, successive $tick_s(t)$ events have nondecreasing values of t, and similarly for successive $tick_r(t)$ events.

$tick_s(t)$
Precondition:
$ctime_s \leq t \wedge$
$|t - now| \leq \epsilon$
Effect:
$ctime_s := t$

$tick_r(t)$
Precondition:
$ctime_r \leq t \wedge$
$|t - now| \leq \epsilon$
Effect:
$ctime_r := t$

ν (time-passage)
Precondition:
$now < t \wedge$
$|ctime_s - t| \leq \epsilon \wedge$
$|ctime_r - t| \leq \epsilon$
Effect:
$now := t$

It is easy to see that A_{Cl} is in fact a safe timed I/O automaton, i.e., that is satisfies the five axioms in Definition 2.17. Clearly **S1** is satisfied and since the $tick_s(t)$ and $tick_r(t)$ do not change the value of now, also **S2** is satisfied. **S3** is satisfied since the first conjunct in the precondition of the step rule for ν explicitly requires real time to increase in time-passage steps. Also clearly, if (s, ν, s') and (s', ν, s'') are steps, then (s, ν, s'') is a steps, so **S4** is satisfied. For the trajectory theorem **S5**, assume that (s, ν, s') is a step. Then $s.ctime_s = s'.ctime_s$ and $s.ctime_r = s'.ctime_r$. So, the mapping from the interval $[s.now, s'.now]$ to states, which to each time t returns the state $[now \mapsto t, ctime_s \mapsto s.ctime_s, ctime_r \mapsto s.ctime_r]$ is a trajectory from s to s'.

10.1.4 Liveness

We need no liveness restriction (other that normal admissibility). Thus, L_{Cl} should consist of all admissible timed executions of A_{Cl}. This is specified by an environment-free timed liveness formula Q_{Cl} for A_{Cl} as follows.

$$Q_{\mathrm{Cl}} \triangleq true$$

It is easy to see that $true$ actually induces the liveness condition consisting of all admissible timed executions of A_{Cl}. However, generally it is not the case that $true$ is an environment-free timed liveness formula for a safe timed I/O automaton. However, for the clock subsystem it is the case. The proof obligation is to show that there exists a (timed) strategy defined on A_{Cl} such that any outcome of the strategy can only consist of admissible and Zeno-tolerant timed executions. But this is clearly the case. First of all the clock subsystem has no inputs. So, the f function of the strategy should simply be defined to provide one $tick_s(t)$ step and one $tick_r(t)$ step every ϵ time units (remember that ϵ is positive). Then any outcome will consist of admissible timed executions only.

10.2 The Timed Channels

The channels we use to connect the sender and the receiver in C are basically the same as the channels we used in G and H. That is, an attempt to send a packet on a channel leads to zero or more copies (a finite number) of the packet being put into the channel. The channels we used in G and H furthermore had some liveness restrictions: if we made infinitely many attempts to send a packet, then infinitely many copies would get through.

Now, the C protocol needs certain timing assumptions about the channels. Not only should the channel delay—once a packet has been successfully placed in the channel—be bounded; it is also necessary to assume an upper bound on the number of attempts needed before a packet has been successfully placed in the channel. Thus, the timed channels should satisfy the following properties.

1. For each packet p_1, if k attempts (for some positive *channel retry number* k) are made to send p_1, then at least one copy of p_1 is put in the channel—even though the k attempts may be interspersed with attempts to send other packets p_2.
2. When a copy of a packet is successfully put in the channel, the copy will be delivered at the other end of the channel after at most the positive *channel delay time* d.

We give an explicit specification of the timed channel $\mathrm{Ch}^t_{sr} = (A_{\mathrm{Ch}^t_{sr}}, L_{\mathrm{Ch}^t_{sr}})$. The specification of the other channel $\mathrm{Ch}^t_{rs} = (A_{\mathrm{Ch}^t_{rs}}, L_{\mathrm{Ch}^t_{rs}})$ is similar (and obtained by replacing sr with rs).

10.2.1 States and Start States

The timed channel needs, as usual, a *now* variable to specify real time. As before the main state variable is a multiset sr. However, in order to specify that each packet must leave the channel at most time d after it entered the channel, we need to mark each packet with a *send time* (not to be confused with the identifier timestamp we associate with messages). Thus, the multiset contains elements of the form (p, t), where p is a packet and t is the real time when p entered the channel. Furthermore, to specify that after at most k attempts to send a packet, the packet has been successfully put into the channel, we have for each packet p a variable $count_{sr}(p)$ which counts the number of unsuccessful attempts to send p.

Variable	Type	Initially	Description
now	T	0	Real time
sr	$\mathcal{B}(P \times \mathsf{T})$	$\emptyset$	A multiset of packets together with the time when the packets were sent.
$count_{sr}(p)$	N	0	For each $p \in P$, $count_{sr}(p)$ contains the number of unsuccessful attempts to send p since last successful attempt.

Define $packets(sr)$ to be the *multiset* of packets in sr, i.e., the multiset obtained by removing all send times t' from all elements (p, t') in sr.

10.2.2 Actions

Input:
  $send\_pkt_{sr}(p)$, $p \in P$
Output:
  $receive\_pkt_{sr}(p)$, $p \in P$
Internal:
  none
Time-passage:
  ν

10.2.3 Steps

$send\_pkt_{sr}(p)$
  Effect:
    let ps be a finite multiset of (p, now) such that
      $ps \neq \emptyset$ if $count_{sr}(p) = k - 1$
    $sr := sr \cup ps$
    if $ps \neq \emptyset$ then
      $count_{sr}(p) := 0$
    else
      $count_{sr}(p) := count_{sr}(p) + 1$

$receive\_pkt_{sr}(p)$
  Precondition:
    $(p, t) \in sr$
  Effect:
    $sr := sr \setminus \{(p, t)\}$

ν (time-passage)
  Precondition:
    $t > now \wedge$
    $\forall (p, t') \in sr : (t \leq t' + d)$
  Effect:
    $now := t$

Note, that the operators $\cup$ in $send\_pkt_{sr}(p)$ and $\setminus$ in $receive\_pkt_{sr}(p)$ are operators on *multisets*, e.g., $sr \setminus \{(p, t)\}$ removes one copy of (p, t) from sr.

As for the clock subsystem it is easy to see that $A_{\mathrm{Ch}^t_{sr}}$ is in fact a safe timed I/O automaton.

10.2.4 Liveness

We need no liveness restriction (other that normal admissibility). Thus, $L_{\mathrm{Ch}^t_{sr}}$ should consist of all admissible timed executions of $A_{\mathrm{Ch}^t_{sr}}$. This is specified by an environment-free timed liveness formula $Q_{\mathrm{Ch}^t_{sr}}$ for $A_{\mathrm{Ch}^t_{sr}}$ as follows.

$$Q_{\mathrm{Ch}^t_{sr}} \triangleq true$$

$Q_{\mathrm{Ch}^t_{sr}}$ clearly is an environment-free timed liveness formula for $A_{\mathrm{Ch}^t_{sr}}$. The g function of a (timed) strategy could be defined to add one copy to sr every time $send\_pkt_{sr}(p)$ occurs. The f function of the strategy should then simply be defined to wait the maximum time (d) before outputting a packet again. In this way (since d is positive), if the environment provides Zeno input, the resulting outcome will be Zeno-tolerant. In all other cases the outcome will consist of admissible timed executions only. That suffices.

10.3 The Sender and the Receiver

Above we have specified the clock subsystem and the timed channels explicitly as live time I/O automata. To specify the sender and receiver processes in C, we use the implicit approach introduced in Section 4.2.1. That is, we describe the automaton part of both the sender and receiver live timed I/O automaton as MMT-specifications (cf. Definition 4.9) $A_{MMT,s}$ and $A_{MMT,r}$, respectively.

When formally defining $steps(A_{MMT,s})$ and $steps(A_{MMT,r})$ below, we furthermore provide an intuitive description of the functionality of C.

10.3.1 States and Start States

Sender

The identifiers used to tag messages at the C level are taken from the sender's local clock and are thus also called *timestamps*. Thus, the domain of the variable $last_s$, which contains the current timestamp, is T. The sender's local clock is contained in $time_s$. This variable must be stable, i.e., it must survive a crash.

Variable		Type	Initially	Description
$mode_s$		{`idle`, `send`, `rec`}	`idle`	The mode of the sender. Compared to G, the sender does not need a special `needid` mode. Instead the sender enters `send` mode directly from `idle` mode.
buf_s		Msg^*	ε	The list of messages at the sender side. Same as at the G level.
$time_s$	S	T	0	The sender's local clock.
$current\text{-}msg_s$		$Msg \cup \{$`nil`$\}$	`nil`	The message about to be sent to the receiver. Same as at the G level.
$last_s$		T	0	The timestamp chosen for the current message. Same as at the G level.
$current\text{-}ack_s$		Bool	*false*	Acknowledgement from the receiver. Same as at the G level.
S = Stable				

Receiver

The receiver's local clock is called $time_r$ and as for the sender's local clock, it must be stable. The receiver also contains the variables $lower_r$ and $upper_r$, both ranging over T. The role of these variables is to delimit the interval of timestamps that the receiver will accept. The variable $upper_r$, which is stable, is initialized to the special timing constant β. Exactly how $lower_r$ and $upper_r$ are manipulated and what the properties of β must be will be described below. The final new variable is $rm\text{-}time_r$. This variable holds the timestamp of the last message delivered to the user and is used to calculate when the receiver can safely clean up its state. This mechanism is also described below.

Variable		Type	Initially	Description
$mode_r$		{`idle`, `rcvd`, `ack`, `rec`}	`idle`	The mode of the receiver. Same as at the G level.
buf_r		Msg^*	ε	The list of messages accepted. Same as at the G level.
$time_r$	S	T	0	The receiver's local clock.
$last_r$		T	0	The timestamp of the last message accepted.
$lower_r$		T	0	A lower bound on the timestamp of a new message that can be accepted.
$upper_r$	S	T	β	An upper bound on such a timestamp
$rm\text{-}time_r$		$\mathsf{T} \cup \{\infty\}$	∞	Remembers the value of the local clock when the last message accepted was delivered to the user. Is used for clean-up purposes.
$nack\text{-}buf_r$		T^*	ε	The list of timestamps for which the receiver will issue a negative acknowledgement.
S = Stable				

10.3.2 Actions

Sender

Input:
- $send\_msg(m)$, $m \in Msg$
- $crash_s$
- $receive\_pkt_{rs}(t, b)$, $t \in \mathsf{T}$, $b \in \mathsf{Bool}$
- $tick_s(t)$, $t \in \mathsf{T}$

Output:
- $ack(b)$, $b \in \mathsf{Bool}$
- $recover_s$
- $send\_pkt_{sr}(m, t)$, $m \in Msg$, $t \in \mathsf{T}$

Internal:
- $choose\_id(t)$, $t \in \mathsf{T}$

Receiver

Input:
- $crash_r$
- $receive\_pkt_{sr}(m, t)$, $m \in Msg$, $t \in \mathsf{T}$
- $tick_r(t)$, $t \in \mathsf{T}$

Output:
- $receive\_msg(m)$, $m \in Msg$
- $recover_r$
- $send\_pkt(t, b)$, $t \in \mathsf{T}$, $b \in \mathsf{Bool}$

Internal:
- $increase\text{-}lower_r(t)$, $t \in \mathsf{T}$
- $increase\text{-}upper_r(t)$, $t \in \mathsf{T}$
- $cleanup_r$

10.3.3 Steps

We now provide the formal definition of the steps of the underlying automata in the MMT-specifications of the sender and receiver. As always we list the definition of the steps of the sender in the left column and the definition of the steps of the receiver in the right column. However, first we provide the intuition behind the functionality of C.

Informally C works as follows during normal mode of operation. The sender associates in a $choose\_id(t)$ step the timestamp t with the next message it wishes to transmit. The timestamp is obtained from the sender's local clock $time_s$, so the precondition for $choose\_id(t)$ guarantees that the local clock has advanced since the last time a timestamp was chosen ($last_s$). The sender is now in `send` mode and starts to transmit repeatedly the current packet over the channel to the receiver. The time between every retry, as we shall see formally in Section 10.3.6, is at most the constant l_s. Based on this constant and the channel characteristics, it is possible to derive the maximum delay before the current packet is received.

The receiver now uses the associated timestamp to decide whether or not to accept a received message—roughly, it will accept a message provided that the associated timestamp is greater than the timestamp of the last message that was accepted, which is kept in $last_r$. However, the receiver does not always remember the timestamp of the last accepted message: it might forget this information because of a crash, or simply because a long time has elapsed since the last message was accepted and it is no longer efficient to remember it (see below). Therefore, the receiver uses safe time estimates determined from its own local clock ($time_r$) to decide when to accept a message. The estimates are kept in $lower_r$ and $upper_r$; the receiver accepts if the message's timestamp is in the interval $(lower_r, upper_r]$.

The $lower_r$ bound is designed to be at least as big as the time of the last message accepted. It can be bigger, however, but in this case is must be sufficiently less than the receiver's local time (at least a maximum one-way message delay (plus a double clock skew) less). This is because the receiver should not accidentally fail to accept a valid message that takes the maximum time to arrive. We note that the reason why we do not want to remember just the last timestamp is that we envision using this protocol in parallel for many users, and a single $lower_r$ bound could be used for all users that have not sent messages for a long while. The special timing constant ρ signifies the amount by which $lower_r$ must be kept smaller than $time_r$ when incremented in $increase\text{-}lower_r(t)$ steps. In Section 10.3.6 we show how ρ should be related to the other timing constants of the system.

The $upper_r$ bound is chosen to be big enough so that the receiver still accepts the most recent messages, even if they arrive very fast. That is, it should be somewhat larger than the current time (at least a double clock skew larger). But this bound is kept in stable storage, and therefore should not be updated very often. Thus, it will generally be set to be a good deal larger than the current local time. When we present the timing constraints in Section 10.3.4 below, we show that at most some time l'_r elapses between every time $upper_r$ is increased (in an $increase\text{-}upper_r(t)$ step). The timing constant β, which occurs in the definition of $increase\text{-}upper_r(t)$ below, then has to be properly related to l'_r in order to guarantee that $upper_r$ is always big enough.

Unlike the H protocol, C will not continuously issue positive acknowledgements for the last packet successfully received. Instead it only issues one positive acknowledgement and returns to `idle` mode (cf. the definition of the $send\_pkt_{rs}(t, true)$ steps below). If this packet is lost in the channel, eventually the receiver will receive another copy of the current packet; this will change $mode_r$ to `ack` and a new positive acknowledgement will be issued. After at most k retries, $(t, true)$ is successfully placed in the buffer and after at most d time units thereafter, the sender will receive the acknowledgement. Once $send\_pkt_{rs}(t, true)$ is enabled, it must occur within l_r time units unless it is disabled in the meantime. This upper bound will be important in order to specify when the receiver is allowed to clean up its state.

This completes a normal cycle of the sender and receiver. After the formal definition of the steps, we return to the description of the special $cleanup_r$ action and what can happen due to crashes and recoveries.

$send\_msg(m)$
  Effect:
    if $mode_s \neq \texttt{rec}$ then
      $buf_s := buf_s \,\hat{}\, m$

$choose\_id(t)$
  Precondition:
    $mode_s = \texttt{idle} \wedge$
    $buf_s \neq \varepsilon \wedge$
    $time_s = t \wedge$
    $t > last_s$
  Effect:
    $mode_s := \texttt{send}$
    $last_s := t$
    $current\text{-}msg_s := head(buf_s)$
    $buf_s := tail(buf_s)$

$send\_pkt_{sr}(m, t)$
  Precondition:
    $mode_s = \texttt{send} \wedge$
    $current\text{-}msg_s = m \wedge$
    $last_s = t$
  Effect:
    none

$receive\_pkt_{sr}(m, t)$
  Effect:
    if $mode_r \neq \texttt{rec}$ then
      if $lower_r < t \leq upper_r$ then
        $mode_r := \texttt{rcvd}$
        $buf_r := buf_r \,\hat{}\, m$
        $last_r := t$
        $rm\text{-}time_r := \infty$
        $lower_r := t$
      else if $last_r < t \leq lower_r$ then
        $nack\text{-}buf_r := nack\text{-}buf_r \,\hat{}\, t$
      else if $mode_r = \texttt{idle} \wedge last_r = t$ then
        $mode_r := \texttt{ack}$

$receive\_msg(m)$
  Precondition:
    $mode_r = \texttt{rcvd} \wedge$
    $buf_r \neq \varepsilon \wedge$
    $head(buf_r) = m$
  Effect:
    $buf_r := tail(buf_r)$
    if $buf_r = \varepsilon$ then
      $mode_r := \texttt{ack}$
      $rm\text{-}time_r := time_r$

$receive\_pkt_{rs}(t, b)$
Effect:
if $mode_s = \texttt{send} \wedge last_s = t$ then
$mode_s := \texttt{idle}$
$current\text{-}ack_s := b$
$current\text{-}msg_s := \texttt{nil}$

$ack(b)$
Precondition:
$mode_s = \texttt{idle} \wedge$
$buf_s = \varepsilon$
$current\text{-}ack_s = b$
Effect:
none

$crash_s$
Effect:
$mode_s := \texttt{rec}$

$recover_s$
Precondition:
$mode_s = \texttt{rec}$
Effect:
$mode_s := \texttt{idle}$
$last_s := time_s$
$buf_s := \varepsilon$
$current\text{-}msg_s := \texttt{nil}$
$current\text{-}ack_s := false$

$tick_s(t)$
Effect:
$time_s := t$

$send\_pkt_{rs}(t, true)$
Precondition:
$mode_r = \texttt{ack} \wedge$
$last_r = t$
Effect:
$mode_r := \texttt{idle}$

$send\_pkt_{rs}(t, false)$
Precondition:
$mode_r \neq \texttt{rec} \wedge$
$nack\text{-}buf_r \neq \varepsilon \wedge$
$head(nack\text{-}buf_r) = t$
Effect:
$nack\text{-}buf_r := tail(nack\text{-}buf_r)$

$crash_r$
Effect:
$mode_r := \texttt{rec}$

$recover_r$
Precondition:
$mode_r = \texttt{rec} \wedge$
$upper_r + 2\epsilon < time_r$
Effect:
$mode_r := \texttt{idle}$
$last_r := 0$
$rm\text{-}time_r := \infty$
$buf_r := \varepsilon$
$lower_r := upper_r$
$upper_r := time_r + \beta$
$nack\text{-}buf_r := \varepsilon$

$increase\text{-}lower_r(t)$
Precondition:
$mode_r \neq \texttt{rec} \wedge$
$lower_r \leq t < time_r - \rho$
Effect:
$lower_r := t$

$increase\text{-}upper_r(t)$
Precondition:
$mode_r \neq \texttt{rec} \wedge$
$upper_r \leq t = time_r + \beta$
Effect:
$upper_r := t$

$cleanup_r$
Precondition:
$mode_r \in \{\texttt{idle}, \texttt{ack}\} \wedge$
$time_r > rm\text{-}time_r + \alpha$
Effect:
$mode_r := \texttt{idle}$
$last_r := 0$
$rm\text{-}time_r := \infty$

$tick_r(t)$
Effect:
$time_r := t$

All that needs to be kept in stable storage is just the local clocks $time_s$ and $time_r$, plus the one variable $upper_r$ of the receiver. When the receiver side crashes and recovers again (cf. the definition of $recover_r$ above), it resets its $lower_r$ bound to the old $upper_r$ bound, to be sure that it will not accept, and thus deliver, any message twice. This explains why we cannot just set $upper_r$ to infinity. It also explains another detail: the precondition for the $recover_r$ steps requires the local clock to grow beyond $upper_r + 2\epsilon$ before recovery can take place. This is because otherwise the new $lower_r$ bound would be too big compared to $time_r$ which could lead to the rejection of a very fast message sent to the system after the recovery of the receiver. If we were to allow such a rejection, C would not correctly (or even safely) implement S since S only allows the loss of messages which are in the system between crash and recovery.

The way the receiver informs the sender that the sender is in a bad `send` state is similar to the way this is done at the G level: when the receiver receives a packet (m, t) where t is not between $lower_r$ and $upper_r$, it should issue a negative acknowledgement for t. However, if $t < last_r$, the receiver has already successfully received a message with a later timestamp, so (m, t) cannot be the current packet of the sender. In this situation the receiver does not issue the negative acknowledgement. (Note, that due to crashes or clean-ups (see below), the receiver may forget $last_r$. However, in this case $last_r = 0$, and the receiver will issue negative acknowledgements for all "bad" timestamps and, in particular, the current one.)

Finally we consider the clean-up mechanism of the sender. When a long time has elapsed since the receiver started to issue positive acknowledgements for the last packet accepted, it can be sure that the sender has received the acknowledgement, and is thus allowed to forget $last_r$ and move to `idle` mode. This is specified in the definition of $cleanup_r$ above. Section 10.3.6 describes how large the timing constant α occurring in the precondition should be.

10.3.4 Timing Constraints

We can now specify $sets(A_{MMT,s})$, $boundmap(A_{MMT,s})$, $sets(A_{MMT,r})$, and $boundmap(A_{MMT,r})$ and thus complete the MMT-specifications of the sender and the receiver.

Sender

The correctness of C depends on an upper bound on the $send\_pkt_{sr}(m, t)$ actions of the sender. Thus, $sets(A_{MMT,s})$ contains only one set of locally-controlled actions and $boundmap(A_{MMT,s})$ then associates a lower and upper bound on this set. Formally we have

$$C^t_{\mathrm{C},s} \triangleq \{send\_pkt_{sr}(m, t) \mid m \in Msg \wedge t \in \mathsf{T}\}$$

and

$$\begin{aligned} b_l(C^t_{\mathrm{C},s}) &\triangleq 0 \\ b_u(C^t_{\mathrm{C},s}) &\triangleq l_s \end{aligned}$$

where l_s is a positive real.

Receiver

Similarly, as mentioned above we put bounds on two sets of locally-controlled actions of the receiver. The two constants l_r and l'_r are both positive reals.

$$\begin{aligned} C^t_{\mathrm{C},r1} &\triangleq \{send\_pkt_{rs}(id, true) \mid id \in ID\} \\ C^t_{\mathrm{C},r2} &\triangleq \{increase\text{-}upper_r(t) \mid t \in \mathsf{T}\} \end{aligned}$$

and

$$\begin{array}{rcl} b_l(C^t_{\mathrm{C},r1}) & = & 0 \\ b_u(C^t_{\mathrm{C},r1}) & = & l_r \\ b_l(C^t_{\mathrm{C},r2}) & = & 0 \\ b_u(C^t_{\mathrm{C},r2}) & = & l'_r \end{array}$$

10.3.5 The Sender and Receiver Safe Timed I/O Automata

The safe timed I/O automata of the sender and receiver processes in C are now given by (cf. Definition 4.10)

$$\begin{array}{rcl} A_{\mathrm{C},s} & \triangleq & time(A_{MMT,s}) \\ A_{\mathrm{C},r} & \triangleq & time(A_{MMT,r}) \end{array}$$

10.3.6 Derived Timing Constants

Before we specify the liveness requirements for the sender and receiver processes of C, we return to the three timing constants β, ρ, and α occurring in the definition of the steps of the sender and receiver, and show how they should be related to the other timing constants. We give the intuition behind the constants, and in the proofs in Section 10.5 we show that the properties of the constants actually guarantee correctness. We first repeat the other timing constants, which are all positive reals:

ϵ The maximum clock skew from real time (at both the sender and receiver side).

l_s An upper time bound between retransmissions of message packets (m, t) from the sender.

l_r An upper time bound between retransmissions of positive acknowledgement packets $(t, true)$ from the receiver.

l'_r An upper bound between *increase-upper*$_r(t)$ steps of the receiver. (This upper bound will usually be bigger than l_r since *increase-upper*$_r(t)$ writes to *stable* storage.)

d An upper bound on channel delay.

Furthermore, the channel retry number k is a fixed positive integer, which represents the number of retries that will guarantee delivery of a packet.

We consider β, ρ, and α one by one.

The Timing Constant β

The timing constant β occurs in the definition of the *increase-upper*$_r(t)$ steps above and indicate the amount by which *upper*$_r$ should be set bigger than *time*$_r$. Assume that the sender's local time is ϵ ahead of real time and the receiver's time is ϵ behind. If the sender picks a timestamp for the current message and this message arrives very fast (in fact arbitrarily fast since we have no lower bounds in the system) at the receiver, the timestamp of this message will be 2ϵ larger than the receiver's local time. Since the message must be accepted, *upper*$_r$ must be at least 2ϵ

larger than $time_r$ at any moment (where the receiver is not crashed). When *increase-upper*$_r(t)$ has occurred, it will recur before l'_r time units. Thus, β should satisfy

$$\beta \geq 2\epsilon + l'_r$$

Note, the smaller β is, the more often *increase-upper*$_r(t)$ steps (and thus writes to stable storage) are required to happen. On the other hand, if β is chosen too big, recovery will be delayed (cf. the definition of $recover_r$).

The Timing Constant ρ

The timing constant ρ occurs in the definition of the *increase-lower*$_r(t)$ steps above and indicate the amount by which $lower_r$ must be smaller than $time_r$. The ρ bound should guarantee that very slow messages from the sender will still be accepted. Assume the sender's local time is ϵ behind real time and the receiver's local time is ϵ ahead. By the time the sender associates a new timestamp t with the current message, $t = time_r - 2\epsilon$. Now, the sender will succeed in placing the current packet in the channel after at most k retries and the delay between each retry is at most l_s. Thus, after kl_s time units, from the time the timestamp was chosen, the current packet must have been placed in the channel, and after at most d time units the packet will be received. Thus, during the time of transmission, the receiver's local time has increased by at most $kl_s + d$ time units (it cannot have increased by more since it was already the maximum amount ahead of real time). We finally get that the timestamp t will be $time_r - kl_s + d + 2\epsilon$ at the time of receipt in this worst case. Thus,

$$\rho \geq kl_s + d + 2\epsilon$$

The Timing Constant α

We finally consider α which occurs in the definition of $cleanup_r$. Clearly, α is the most complicated of the timing constants.

There is no bound on how fast new packets can arrive at the receiver, nor are there bounds on how fast the receiver delivers accepted messages to the user. The α bound has to indicate the first time by which it is no longer necessary to remember $last_r$. This bound thus has to be calculated from the time the last message accepted (i.e., the message for which $last_r$ gives the timestamp) is delivered.

We consider a situation where neither the sender nor the receiver crashes.

Let now_{rm} be a real time when *receive_msg*(m) occurs and buf_r becomes empty, and let $time_{r,rm}$ be the corresponding value of $time_r$. Also, let $now_{send\text{-}ack,i}$ denote the real time when the receiver performs its ith *send_pkt*$_{rs}(t, true)$ step for the current timestamp t (contained in $last_r$). We have,

$$now_{send\text{-}ack,1} \leq now_{rm} + l_r$$

The maximum delay until the receiver receives (m, t) again is $kl_s + d$. (Just before the receiver performed *send_pkt*$_{rs}(t, true)$ the sender might have succeeded in putting a copy of (m, t) into the channel, and this copy could be fast such that it arrives with no delay at the receiver, i.e., just before *send_pkt*$_{rs}(t, true)$. Since such copies are not buffered by the receiver, the receiver has to wait for the next copy which arrives after at most $kl_s + d$ time units.) Thus,

$$\begin{aligned} now_{send\text{-}ack,2} &\leq now_{send\text{-}ack,1} + (kl_s + d + l_r) \\ &= now_{rm} + l_r + (kl_s + d + l_r) \end{aligned}$$

And for the kth $send\_pkt_{rs}(t, true)$,

$$\begin{aligned} now_{send\text{-}ack,k} &\leq now_{send\text{-}ack,k-1} + (kl_s + d + l_r) \\ &= now_{send\text{-}ack,k-2} + 2(kl_s + d + l_r) \\ &= \ldots \\ &= now_{send\text{-}ack,1} + (k-1)(kl_s + d + l_r) \\ &= now_{rm} + l_r + (k-1)(kl_s + d + l_r) \end{aligned}$$

Now, let $now_{ack\text{-}rcvd}$ be the real time when $(t, true)$ is received by the sender and let $time_{r,ack\text{-}rcvd}$ be the corresponding value of $time_r$.

$$\begin{aligned} now_{ack\text{-}rcvd} &\leq now_{send\text{-}ack,k} + d \\ &= now_{rm} + l_r + (k-1)(kl_s + d + l_r) + d \\ &= now_{rm} + k(l_r + d) + (k-1)kl_s \end{aligned}$$

Since $time_r - \epsilon \leq now$ and $time_r + \epsilon \geq now$, we have

$$\begin{aligned} time_{r,ack\text{-}rcvd} - \epsilon &\leq now_{ack\text{-}rcvd} \\ &\leq now_{rm} + k(l_r + d) + (k-1)kl_s \\ &\leq time_{r,rm} + \epsilon + k(l_r + d) + (k-1)kl_s \end{aligned}$$

Thus,

$$time_{r,ack\text{-}rcvd} \leq time_{r,rm} + k(l_r + d) + (k-1)kl_s + 2\epsilon$$

Since the state variable $rm\text{-}time_r$ of the receiver is set to $time_{r,rm}$ at the time of the last $receive\_msg(m)$ step, we see from the definition of $cleanup_r$ that α should satisfy

$$\alpha \geq k(l_r + d) + (k-1)kl_s + 2\epsilon$$

Note that

- α depends on k^2 (but fortunately not on $k^2 d$).
- the 2ϵ in α is actually not obtained as the maximum difference between sender and receiver clocks but as two times the maximum receiver clock skew.

10.3.7 Liveness

The liveness requirements to the sender and receiver processes of C are weak fairness to sets of locally-controlled actions.

Sender

Let

$$\begin{aligned} C_{\mathrm{C},s} \triangleq\ & \{ack(true), ack(false), recover_s\} \cup \\ & \{choose\_id(t) \mid t \in \mathsf{T}\} \cup \\ & \{send\_pkt_{sr}(m, id) \mid m \in Msg \wedge id \in ID\} \end{aligned}$$

Then the liveness condition $L_{\mathrm{C},s}$ is induced by

$$Q_{\mathrm{C},s} \quad \triangleq \quad WF(C_{\mathrm{C},s})$$

Note, that it is actually not necessary to add the $send\_pkt_{sr}(m, id)$ actions to $C_{\mathrm{C},s}$ since these actions are already constrained by the stronger timing requirements.

In the untimed setting weak fairness to locally-controlled actions is trivially environment-free. This is not necessarily the case in the timed setting. The problem is that even with the simple weak fairness requirements, the system might still collaborate with a Zeno environment and generate outcome timed executions that are not Zeno-tolerant. However, $Q_{\mathrm{C},s}$ is environment-free for $A_{\mathrm{C},s}$. Intuitively, consider a strategy that for actions in $C^t_{\mathrm{C},s}$ always waits the maximum delay l_s before performing an action in $C^t_{\mathrm{C},s}$. The actions in $C_{\mathrm{C},s}$ should then be handled similarly with some arbitrary positive real number as bound. If the sets $C^t_{\mathrm{C},s}$ and $C_{\mathrm{C},s}$ becomes disabled, there are no requirements so the strategy should just let time pass forever. With this strategy, if the environment is not Zeno, each outcome timed execution will be in $L_{\mathrm{C},s}$, and if the environment is Zeno, each outcome timed execution will be Zeno-tolerant.

Finally note that, by Proposition 3.4, $Q_{\mathrm{C},s}$ is stuttering-insensitive.

Receiver

Similarly, let

$$\begin{aligned} C_{\mathrm{C},r1} \quad &\triangleq \quad \{recover_r\} \cup \{receive\_msg(m) \mid m \in Msg\} \cup \\ &\quad\quad \{send\_pkt_{rs}(id, true) \mid id \in ID\} \\ C_{\mathrm{C},r2} \quad &\triangleq \quad \{send\_pkt_{rs}(t, false) \mid t \in \mathsf{T}\} \end{aligned}$$

Then $L_{\mathrm{C},r}$ is induced by

$$\begin{aligned} Q_{\mathrm{C},r} \quad &\triangleq \quad WF(C_{\mathrm{C},r1}) \wedge \\ &\quad\quad WF(C_{\mathrm{C},r2}) \end{aligned}$$

As for the sender, $Q_{\mathrm{C},r}$ is stuttering-insensitive and environment-free for $A_{\mathrm{C},r}$.

10.4 The Specification of C

C is the parallel composition of sender, receiver, two channels, and clock subsystem. First define $\mathrm{C}'' = (A''_{\mathrm{C}}, L''_{\mathrm{C}})$ as,

$$\mathrm{C}'' \quad \triangleq \quad \mathrm{C}_s \| \mathrm{C}_r \| \mathrm{Ch}^t_{sr} \| \mathrm{Ch}^t_{rs} \| \mathrm{Cl}$$

By Proposition 4.17, L''_{C} is induced by Q_{C}, which is defined as

$$Q_{\mathrm{C}} \quad \triangleq \quad Q_{\mathrm{C},s} \wedge Q_{\mathrm{C},r} \wedge Q_{\mathrm{Ch}^t_{sr}} \wedge Q_{\mathrm{Ch}^t_{rs}} \wedge Q_{\mathrm{Cl}}$$

C'' has channel communication as well as ticks from the clock subsystem as external (output) actions. To obtain a specification where the ticks are hidden, define

$$\mathcal{A}'_{\mathrm{C}} \quad \triangleq \quad \{tick_s(t) \mid t \in \mathsf{T}\} \cup \{tick_s(t) \mid t \in \mathsf{T}\}$$

Then $\mathrm{C}' = (A'_{\mathrm{C}}, L'_{\mathrm{C}})$ is defined as

$$\mathrm{C}' \quad \triangleq \quad \mathrm{C}'' \setminus \mathcal{A}'_{\mathrm{C}}$$

By Proposition 4.18, L'_{C} is induced by Q_{C}.

Finally, to get C, we hide the channel actions. First define

$$\begin{array}{rcl} \mathcal{A}_{\mathrm{C}} & \triangleq & \{\mathit{send\_pkt}_{sr}(m,t) \mid m \in \mathit{Msg} \wedge t \in \mathsf{T}\} \cup \\ & & \{\mathit{receive\_pkt}_{sr}(m,t) \mid m \in \mathit{Msg} \wedge t \in \mathsf{T}\} \cup \\ & & \{\mathit{send\_pkt}_{rs}(t,b) \mid t \in \mathsf{T} \wedge b \in \mathsf{Bool}\} \cup \\ & & \{\mathit{receive\_pkt}_{rs}(t,b) \mid t \in \mathsf{T} \wedge b \in \mathsf{Bool}\} \end{array}$$

Then the specification of $\mathrm{C} = (A_{\mathrm{C}}, L_{\mathrm{C}})$ is given by

$$\mathrm{C} \quad \triangleq \quad \mathrm{C}' \setminus \mathcal{A}_{\mathrm{C}}$$

Again, by Proposition 4.18, L_{C} is induced by Q_{C}.

We now turn to proving the correctness of C. This involves, among other things, use of the Embedding Theorem of Section 2.3.

10.5 Correctness of C

The objective of this section is to prove correctness of C—not with respect to G but with respect to the *patient* version of G. Then the Embedding Theorem of Chapter 2 will allow us to conclude that C correctly implements *patient*(S).

First, recall that the G protocol uses a set *ID* of identifiers that has to satisfy certain conditions (cf. Section 8.1). We instantiate this set with the time domain T, which clearly satisfies the conditions. Thus, we set $\mathit{ID} = \mathsf{T}$ in the proofs below.

Next, recall from Section 9.4 that we first proved that H′ correctly implements G′, where H′ and G′ are the versions of H and G with channel communication as external actions. This was because the Execution Correspondence Theorem gives a stronger result the more *external* actions the systems have in common. The same motivation leads us first to consider the proof that C′ correctly implements *patient*(*G*′). Thus, let $\mathrm{G}^{p\prime} = (A_{\mathrm{G}}^{p\prime}, L_{\mathrm{G}}^{p\prime})$ be defined as

$$\mathrm{G}^{p\prime} \triangleq \mathit{patient}(\mathrm{G}')$$

By Proposition 4.22, $L_{\mathrm{G}}^{p\prime}$ is induced by Q_{G} and Q_{G} is minimal.

In order to prove that C′ correctly implements $\mathrm{G}^{p\prime}$, we first enhance C′ with history variables and thereby obtain $\mathrm{C}^{h\prime} = (A_{\mathrm{C}}^{h\prime}, L_{\mathrm{C}}^{h\prime})$. We then prove several invariants of $A_{\mathrm{C}}^{h\prime}$ and show the existence of a timed refinement mapping from $A_{\mathrm{C}}^{h\prime}$ to $A_{\mathrm{G}}^{p\prime}$. Finally, this refinement result is used to prove that $\mathrm{C}^{h\prime}$ correctly implements $\mathrm{G}^{p\prime}$ and, in turn, that C correctly implements *patient*(S).

10.5.1 Adding History Variables

We add two history variables to C′ and denote the resulting live timed I/O automaton by $\mathrm{C}^{h\prime} = (A_{\mathrm{C}}^{h\prime}, L_{\mathrm{C}}^{h\prime})$.

Variable		**Type**	**Initially**	**Description**
used_s	H	T^*	ε	The list of timestamps used by the sender. Same as at the G level.
deadline	H	$\mathsf{T} \cup \{\infty\}$	∞	An estimated deadline on arrival of the current packet.
H = History				

We now show how the history variables should be updated (cf. Section 9.4.1 where history variables are added at the H level). We refer to Section 5.2.5 for a description on how we are allowed to manipulate the history variables.

choose_id(t)
  Precondition:
    (* Precondition from C_s *)
    ...
  Effect:
    (* Effect clause from C_s *)
    ...
    $used_s := used_s \,\hat{}\, t$
    if $mode_r \neq \texttt{rec}$ then
      $deadline := now + kl_s + d$

receive_pkt$_{sr}(m, t)$
  Precondition:
    (* Precondition from Ch^t_{sr} *)
    ...
  Effect:
    (* Effect clause from Ch^t_{sr} *)
    ...
    (* Effect clause from C_r *)
    if $mode_r \neq \texttt{rec}$ then
      if $lower_r < t \leq upper_r$ then
        ...
        if $t = last_s \wedge mode_s = \texttt{send}$ then
          $deadline := \infty$
      else if $last_r < t \leq lower_r$ then
        ...
      else if $mode_r = \texttt{idle} \wedge last_r = t$ then
        ...

crash$_s$
  Effect:
    (* Effect clause from C_s *)
    ...
    $deadline := \infty$

crash$_r$
  Effect:
    (* Effect clause from C_r *)
    ...
    $deadline := \infty$

By Lemma 5.32, $L_{\mathrm{C}}^{h\,\prime}$ is induced by Q_{C}.

10.5.2 Invariants

In this section we state the invariants of $A_{\mathrm{C}}^{h\,\prime}$ we need below. The proofs are deferred to Appendix C.

The first invariant deals with the local clocks of the sender and receiver in $A_{\mathrm{C}}^{h\,\prime}$ and states that the maximal clock skew for these is ϵ, which then implies that $time_s$ and $time_r$ can differ by at most 2ϵ.

Invariant 10.1

1. $time_s = ctime_s$

2. $time_r = ctime_r$

3. $|time_s - now| \leq \epsilon$

4. $|time_r - now| \leq \epsilon$

5. $|time_s - time_r| \leq 2\epsilon$

■

When the receiver is not in recovery mode, $upper_r$ is updated regularly to ensure that timestamps chosen by the sender are never "too big". This is expressed by the following invariant.

Invariant 10.2

1. If $mode_r \neq \texttt{rec}$ then $upper_r \geq now + \epsilon$

2. If $mode_r \neq \texttt{rec}$ then $upper_r \geq time_s$

3. If $mode_r \neq \texttt{rec}$ then $upper_r \geq time_r$

■

The following invariant deals with $last_s$. Since the local clock $time_s$ can never decrease and due to the facts that the current timestamp is taken from $time_s$, and $last_s$ gets reset to $time_s$ after a crash, it is the case that $last_s$ is always greater than or equal to $time_s$. Furthermore, the current timestamp (i.e., the value of $last_s$ when $mode_s = \texttt{send}$) can never be 0.

Invariant 10.3

1. $last_s \leq time_s$

2. If $mode_s = \texttt{send}$ then $last_s > 0$

■

The state variable $last_r$ contains the timestamp of the last message accepted by the receiver (or 0 right after recovery or cleanup). The next invariant states that the value of $last_r$ can never be considered a good timestamp by the receiver. (Otherwise the receiver could accidentally accept the same packet twice). Specifically, $last_r$ is always less than or equal to $lower_r$. Furthermore, $lower_r$ is always less than or equal to $upper_r$.

Invariant 10.4

1. $last_r \leq lower_r$

2. $lower_r \leq upper_r$

■

The next invariant states that the number of unsuccessful attempts (since the last successful attempt) to send a packet (m, t), where $t > last_s$, is always 0. Actually, no attempts can ever have been made to transmit (m, t) since the sender cannot yet have issued the timestamp t. Furthermore, the number of unsuccessful attempts (since last successful attempt) to send any packet can never be greater than or equal to k (the channel retry number).

Invariant 10.5

1. If $t > last_s$ then $count_{sr}(m,t) = 0$
2. $count_{sr}(m,t) \leq k - 1$

■

The following invariant is a key invariant and states properties of timestamps associated with messages and acknowledgements in the channels.

Invariant 10.6

1. If $(m,t) \in packets(sr)$ then $t \leq last_s$
2. If $(m, last_s) \in packets(sr) \wedge mode_s = \texttt{send}$ then $m = current\text{-}msg_s$
3. $last_r \leq last_s$
4. If $(t, true) \in packets(rs)$ then $t \leq last_s$
5. If $t \in nack\text{-}buf_r$ then $t \leq lower_r$
6. If $(t,b) \in packets(rs)$ then $t \leq lower_r$

■

Properties of the relationship between $lower_r$ and $last_s$ are stated in the following invariant.

Invariant 10.7

1. $lower_r \leq time_s$
2. If $last_s < time_s$ then $lower_r < time_s$

■

The sender chooses increasing timestamps as indicated by the next invariant.

Invariant 10.8

1. If t precedes t' in $used_s$ then $t < t'$

■

Due to the way the channels deal with the maximum channel delay d, the following invariant holds.

Invariant 10.9

1. If $((m,t),t') \in sr$ then $t' \leq now + d$

■

To state the next invariant, we need a few definitions. Define the function *mintime* with the following signature

$$mintime : P \times (\mathcal{B}(P \times \mathsf{T})) \to \mathsf{T}$$

in the following way

$$mintime(p, ch) \triangleq \begin{cases} t & \text{if } (p,t) \in ch \wedge \forall (p,t') \in ch : (t' \geq t) \\ 0 & \text{otherwise} \end{cases}$$

Thus, $mintime(p, ch)$ gives the minimal send time associated with the packet p in ch (and defaults to 0 if $p \notin packets(ch)$). Remember from the way we model the channels sr and rs that each element in the channels has two times associated with it: one is a timestamp chosen by the sender; the other represents the real time when the element was put into the channel and is called the send time of the packet. The function *mintime* returns send times.

For any state s of $A_{\mathrm{C}}^{h\,\prime}$ we define $s.bound$ in the following way, where we use m and t as shorthands for $s.current\text{-}msg_s$ and $s.last_s$, respectively.

$$s.bound \triangleq \begin{cases} \infty & \text{if } s.mode_s \neq \texttt{send} \\ d + mintime((m,t), s.sr) & \text{if } s.mode_s = \texttt{send} \wedge \\ & \quad (m,t) \in packets(s.sr) \\ s.last(C^t_{\mathrm{C},s}) + (k - 1 - s.count_{sr}(m,t))l_s + d & \text{if } s.mode_s = \texttt{send} \wedge \\ & \quad (m,t) \notin packets(s.sr) \end{cases}$$

Thus, $s.bound$ represents an estimated time of arrival for the current packet. With this definition we can prove very important properties of the history variable *deadline*.

Invariant 10.10

1. $bound \leq deadline$
2. $now \leq bound$
3. $now \leq deadline$
4. If $deadline \neq \infty$ then $deadline \leq last_s + \epsilon + kl_s + d$
5. If $deadline \neq \infty$ then $now \leq last_s + \epsilon + kl_s + d$
6. If $deadline \neq \infty$ then $last_s > lower_r$
7. If $deadline \neq \infty$ then $mode_s = \texttt{send} \wedge mode_r \neq \texttt{rec}$

■

The receiver is allowed to clean up its state, i.e., to forget the timestamp of the last message accepted and move to `idle` mode, when a sufficiently long time has elapsed since the message was delivered to the user. This is because by then the receiver can be certain that the sender has received a positive acknowledgement packet for the current packet. In the specification of the receiver, α indicates how long time the receiver must wait before cleaning up. The following invariant captures the fact that α is properly defined. We do not prove the invariant but note that it can be proved in a fashion similar to the proof of Invariant 10.10.

Invariant 10.11

1. *If* $mode_s = \mathtt{send} \wedge mode_r \neq \mathtt{rec} \wedge time_r > rm\text{-}time_r + \alpha$ *then* $last_s \neq last_r$

The final two invariants are trivial and state that any timestamps occurring in the channels are positive.

Invariant 10.12

1. If $(m, t) \in packets(sr)$ then $t > 0$

■

Invariant 10.13

1. If $(t, b) \in packets(rs)$ then $t > 0$

■

We refer to the conjunction of the invariants above by I_{C^h}.

10.5.3 Safety

We now define a function from $states(A_{\mathrm{C}}^{h\,\prime})$ to $states(A_{\mathrm{G}}^{p\,\prime})$. Below, in Lemma 10.15, this function is proved to be a timed refinement mapping from $A_{\mathrm{C}}^{h\,\prime}$ to $A_{\mathrm{G}}^{p\,\prime}$ with respect to I_{C^h} and I_{G}. (Note, that the invariant I_{G} of A_{G} is clearly also an invariant of $A_{\mathrm{G}}^{p\,\prime}$.)

Below we use the notation $(t_1, t_2]$ to denote both the left-open interval from a to b and the set $\{t \mid t_1 < t \leq t_2\}$. Similar notation is used for the other kinds of intervals.

Definition 10.14 (Refinement Mapping from $A_{\mathrm{C}}^{h\,\prime}$ to $A_{\mathrm{G}}^{p\,\prime}$)

If $s \in states(A_{\mathrm{C}}^{h\,\prime})$ then define $R_{\mathrm{CG}}(s)$ to be the state $u \in states(A_{\mathrm{G}}^{p\,\prime})$ such that

1. $$\begin{array}{lcl} u.now &=& s.now \\ u.mode_s &=& s.mode_s \\ u.buf_s &=& s.buf_s \\ u.current\text{-}msg_s &=& s.current\text{-}msg_s \\ u.current\text{-}ack_s &=& s.current\text{-}ack_s \\ u.used_s &=& s.used_s \\ u.mode_r &=& s.mode_r \\ u.buf_r &=& s.buf_r \\ u.nack\text{-}buf_r &=& s.nack\text{-}buf_r \end{array}$$

2. $$\begin{array}{lcl} u.last_s &=& (\text{if } s.last_s = 0 \text{ then } \mathtt{nil} \text{ else } s.last_s) \\ u.last_r &=& (\text{if } s.last_r = 0 \text{ then } \mathtt{nil} \text{ else } s.last_r) \end{array}$$

3. $u.good_s = \{s.time_s\} \setminus \{s.last_s\}$

4. $u.good_r = (s.lower_r, s.upper_r]$

5. $u.issued_r = (0, s.upper_r]$

6. $u.current\text{-}ok = (s.deadline \neq \infty)$

7. $u.sr \quad = \quad packets(s.sr)$
   $u.rs \quad = \quad packets(s.rs)$

■

Note how the values of most variables at the G level correspond directly to the value of the same variables at the C level as expressed by Part 1. Part 2 gives the trivial correspondence for the $last_s$ and $last_r$ variables. Parts 3–5 contain the interesting aspects of the mapping: $good_s$—the timestamps the sender can associate to messages—consists of the value of $time_s$, but only if the clock has increased since the last timestamp was chosen; otherwise $good_s$ is empty; $good_r$ is, as expected, the left-open interval from $lower_r$ to $upper_r$; finally, the receiver has *issued* all timestamps up to and including $upper_r$. The correspondence in Part 6 between *current-ok* at the G level and *deadline* at the C level is obvious. Finally, Part 7 states that each channel at the G level is obtained from the corresponding channel at the C level by removing the send time components of all elements.

We now prove that R_{CG} is in fact a timed refinement mapping from $A^{h}_{\mathrm{C}}{}'$ to $A^{p}_{\mathrm{G}}{}'$ (with respect to I_{C^h} and I_{G}).

Lemma 10.15

$A^{h}_{\mathrm{C}}{}' \leq_{tR} A^{p}_{\mathrm{G}}{}'$ *via* R_{CG}.

Proof

We prove that R_{CG} is a timed refinement mapping from $A^{h}_{\mathrm{C}}{}'$ to $A^{p}_{\mathrm{G}}{}'$ with respect to I_{C^h} and I_{G}. We check the three conditions (which we call real time correspondence, base case, and inductive case, respectively) of Definition 5.18.

Real Time Correspondence

From the definition of R_{CG} we see that for all states s of C, $R_{\mathrm{CG}}(s).now = s.now$ as required.

Base Case

For the initial condition, let s be the start state of C. Then it is easy to check that $R_{\mathrm{CG}}(s)$ is a start state of $A^{p}_{\mathrm{G}}{}'$.

Inductive Case

Assume $(s, a, s') \in steps(A^{h}_{\mathrm{C}}{}')$ such that s and s' satisfy I_{C^h} and $R_{\mathrm{HG}}(s)$ satisfies I_{G}. Below we consider cases based on a (and sometimes subcases of each case) and for each (sub)case we define a finite execution fragment α of $A^{p}_{\mathrm{G}}{}'$ of the form $(R_{\mathrm{CG}}(s), a', u'', a'', u''', \ldots, R_{\mathrm{CG}}(s'))$ with $vis\text{-}trace(\alpha) = vis\text{-}trace(a)$. For brevity we let u denote $R_{\mathrm{HG}}(s)$ and u' denote $R_{\mathrm{HG}}(s')$.

$\underline{a = \nu}$

Then $(u, \nu, u') \in steps(A^{p}_{\mathrm{G}}{}')$: the only change in going from s to s' is that the *now* variable increases, thus, by definition of R_{CG}, the only difference between u and u' is that the *now* variable of $A^{p}_{\mathrm{G}}{}'$ increases and all such changes are allowed in $A^{p}_{\mathrm{G}}{}'$.

$a \in \{send\_msg(m), receive\_msg(m), ack(b)\}$

Then it is easy to see that $(u, a, u') \in steps(A_{\mathrm{G}}^{p\,\prime})$. This step (and finite execution fragment) clearly has the right visible trace.

$a \in \{crash_s, crash_r\}$

Then it is easy to see that $(u, a, u') \in steps(A_{\mathrm{G}}^{p\,\prime})$. This step (and finite execution fragment) clearly has the right visible trace.

The only thing to note here is the handling of *deadline*. The step of $A_{\mathrm{C}}^{h\,\prime}$ changes *deadline* to ∞ but this corresponds, according to the definition of R_{CG}, to changing *current-ok* to *false* in $A_{\mathrm{G}}^{p\,\prime}$ as required by the definition of the *crash* actions in $A_{\mathrm{G}}^{p\,\prime}$.

$a = recover_s$

We show that $(u, recover_s, u'', shrink\_good_s(s.time_s), u')$, where u'' is defined below, is a finite execution fragment of $A_{\mathrm{G}}^{p\,\prime}$ by showing that $(u, recover_s, u'')$ and $(u'', shrink\_good_s(s.time_s), u')$ are steps of $A_{\mathrm{G}}^{p\,\prime}$. Clearly the execution fragment has the right visible trace.

$$
\begin{array}{llll}
\text{Define } & u''.mode_s & = & \texttt{idle} \\
& u''.last_s & = & s.time_s \\
& u''.buf_s & = & \varepsilon \\
& u''.current\text{-}msg_s & = & \texttt{nil} \\
& u''.current\text{-}ack_s & = & false \\
& u''.x & = & u.x \quad \text{for the remaining state variables } x
\end{array}
$$

First, consider $(u, recover_s, u'')$. From the definition of $recover_s$ in $A_{\mathrm{C}}^{h\,\prime}$ we have that $s.mode_s =$ `rec` which implies, by the definition of R_{CG}, that also $u.mode_s =$ `rec`. Thus, $recover_s$ is enabled in u. Then, by definition of u'' and $recover_s$ in $A_{\mathrm{G}}^{p\,\prime}$, clearly $(u, recover_s, u'') \in steps(A_{\mathrm{G}}^{p\,\prime})$.

Next, consider $(u'', shrink\_good_s(s.time_s), u')$. The definition of $shrink\_good_s$ in $A_{\mathrm{G}}^{p\,\prime}$ has no precondition, so $shrink\_good_s(s.time_s)$ is enabled in u''. From the definitions of u'' and R_{CG} we have that $u''.good_s = u.good_s \subseteq \{s.time_s\}$.

We must show that the differences between u'' and u' are allowed by the definition of the $shrink\_good_s(s.time_s)$ steps in $A_{\mathrm{G}}^{p\,\prime}$. This amounts, by the definition of $shrink\_good_s(s.time_s)$ in $A_{\mathrm{G}}^{p\,\prime}$, to showing that $u'.good_s = u''.good_s \setminus \{s.time_s\}$ and that all other state variables of $A_{\mathrm{G}}^{p\,\prime}$ have the same values in u'' and u'.

For $good_s$ we have that $u'.good_s = \emptyset$ (since $s'.time_s = s'.last_s$), but from above we have $u''.good_s \subseteq \{s.time_s\}$, so $u'.good_s = u''.good_s \setminus \{s.time_s\}$ as required.

It is easy to check that the rest of the state variables of $A_{\mathrm{G}}^{p\,\prime}$ have the same values in u'' and u'.

$\pi = recover_r$

We show that
$(u, shrink\_good_r((s.lower_r, s.upper_r]), u'', grow\_good_r((s.upper_r, s.time_r + \beta]), u''', recover_r, u')$, where u'' and u''' are defined below, is a finite execution fragment of $A_{\mathrm{G}}^{p\,\prime}$ by showing that $(u, shrink\_good_r((s.lower_r, s.upper_r]), u'')$, $(u'', grow\_good_r((s.upper_r, s.time_r + \beta]), u''')$, and $(u''', recover_r, u')$ are steps of $A_{\mathrm{G}}^{p\,\prime}$. The execution fragment clearly has the right visible trace.

Define $u''.good_r = \emptyset$
$u''.x = u.x$ for the remaining state variables x

First, consider $(u, shrink\_good_r((s.lower_r, s.upper_r]), u'')$. From the precondition of the $recover_r$ steps in ${A^h_{\mathrm{C}}}'$ and the definition of R_{CG} we have that $u.mode_r = s.mode_r = \mathtt{rec}$. Then Invariant 8.6 Part 2 implies that $u.current\text{-}ok = false$, thus, $shrink\_good_r((s.lower_r, s.upper_r])$ is enabled in u. Since the definition of R_{CG} implies that $u.good_r = (s.lower_r, s.upper_r]$, it is easy to see that $(u, shrink\_good_r((s.lower_r, s.upper_r]), u'') \in steps({A^p_{\mathrm{G}}}')$.

Define $u'''.issued_r = (0, s.time_r + \beta]$
$u'''.good_r = (s.upper_r, s.time_r + \beta]$
$u'''.x = u''.x$ for the remaining state variables x

Next, consider $(u'', grow\_good_r((s.upper_r, s.time_r + \beta]), u''')$. By definition of u'' and R_{CG} we have that $u''.issued_r = u.issued_r = (0, s.upper_r]$. So, $(s.upper_r, s.time_r + \beta]$ and $u''.issued_r$ do not intersect. Also, by adding $(s.upper_r, s.time_r + \beta]$ to $issued_r$ we still have infinitely many unused timestamps left in T. Thus, $grow\_good_r((s.upper_r, s.time_r + \beta])$ is enabled in u''. Since $u''.good_r = \emptyset$ by definition, it is easy to see that the change in $good_r$ is as required by the definition of the $grow\_good_r((s.upper_r, s.time_r + \beta])$ steps in ${A^p_{\mathrm{G}}}'$. To show that also $issued_r$ is handled correctly, we must show that $u'''.issued_r = u''.issued_r \cup (s.upper_r, s.time_r + \beta]$, i.e., we must show that $(0, s.time_r + \beta] = (0, s.upper_r] \cup (s.upper_r, s.time_r + \beta]$. A sufficient condition for this to hold is that $s.time_r + \beta \geq s.upper_r$, but this is implied by the precondition of the $recover_r$ step in ${A^h_{\mathrm{C}}}'$. To leave all other state variables unchanged is also as required by the definition of $grow\_good_r((s.upper_r, s.time_r + \beta])$ in ${A^p_{\mathrm{G}}}'$.

Finally, consider $(u''', recover_r, u')$. We have $u'''.mode_r = u.mode_r = s.mode_r = \mathtt{rec}$, so $recover_r$ is enabled in u'''. We show that all state variables are handled according to the definition of $recover_r$ in ${A^p_{\mathrm{G}}}'$. The only interesting cases are $issued_r$ and $good_r$.

For $issued_r$ we have $u'''.issued_r = (0, s.time_r + \beta]$ by definition of u''' and furthermore $u'.issued_r = (0, s'.upper_r] = (0, s.time_r + \beta]$ by definition of R_{CG} and the $recover_r$ step in ${A^h_{\mathrm{C}}}'$. Thus, $u'''.issued_r = u'.issued_r$ and this is allowed by the definition of $recover_r$ in ${A^p_{\mathrm{G}}}'$ if $|\mathsf{T} \setminus s'.issued_r| = \infty$ which is clearly satisfied and if $u'.issued_r$ includes a) $u'''.issued_r$, b) $u'''.used_s$, and c) $u'''.good_s$. Case a) is clearly satisfied. For b) we have $u'''.used_s = u.used_s = (0, s.last_s]$. Thus, we must show that $s.last_s \leq s.time_r + \beta$, but this follows from $s.last_s \leq s.time_s \leq s.time_r + 2\epsilon \leq s.time_r + \beta$, where the first inequality follows from Invariant 10.3 Part 1, the second inequality follows from Invariant 10.1 Part 5, and the third inequality follows from the definition of β. For c) we have $u'''.good_s = u.good_s = \{s.time_s\} \setminus \{s.last_s\}$. It suffices to show that $s'.time_s \leq s'.upper_r$ (since $s'.time_s = s.time_s$ and $s'.upper_r = s.time_r + \beta$), but that follows from Invariant 10.2 Part 2. Thus, $issued_r$ is handled correctly.

For $good_r$ we have $u'''.good_r = (s.upper_r, s.time_r + \beta]$ and $u'.good_r = (s'.lower_r, s'.upper_r]$ but since $s'.lower_r = s.upper_r$ and $s'.upper_r = s.time_r + \beta$, by definition of the $recover_r$ step in ${A^h_{\mathrm{C}}}'$, we have that $u'''.good_r = u'.good_r$ as required by the definition of $recover_r$ in ${A^p_{\mathrm{G}}}'$.

$\underline{a \in \{send\_pkt_{sr}(m, t), send\_pkt_{rs}(t, true), send\_pkt_{rs}(t, false)\}}$

It is straightforward to show that $(u, a, u') \in steps({A^p_{\mathrm{G}}}')$. This step (and finite execution fragment) clearly has the right visible trace.

$\underline{a = receive\_pkt_{sr}(m, t)}$

We consider cases.

1. $s.mode_r \neq \mathtt{rec}$ and $s.lower_r < t \leq s.upper_r$.

   We show that $(u, receive\_pkt_{sr}(m,t), u'', shrink\_good_r((s.lower_r, t]), u')$, where u'' is defined below, is a finite execution fragment of $A_{\mathrm{G}}^{p\,\prime}$ by showing that $(u, receive\_pkt_{sr}(m,t), u'')$ and $(u'', shrink\_good_r((s.lower, t]), u')$ are steps of $A_{\mathrm{G}}^{p\,\prime}$. Clearly the execution fragment has the right visible trace.

   $$\begin{array}{lcll} \text{Define } u''.good_r & = & u.good_r \setminus \{t' \mid t' \leq_u t\} & \\ u''.x & = & u'.x & \text{for the remaining state variables } x \end{array}$$

   First, consider $(u, receive\_pkt_{sr}(m,t), u'')$. By the case assumption and the definition of R_{CG}, we have $u.mode_r \neq \mathtt{rec}$ and $t \in u.good_r$. Then, by definition of $receive\_pkt_{sr}(m,t)$ in $A_{\mathrm{G}}^{p\,\prime}$ and u'' it is easy to see that $(u, receive\_pkt_{sr}(m,t), u'') \in steps(A_{\mathrm{G}}^{p\,\prime})$.

   Then consider $(u'', shrink\_good_r((s.lower_r, t]), u')$. We show that $shrink\_good_r((s.lower_r, t])$ is enabled in u''. Assume $u''.current\text{-}ok = true$ (otherwise $shrink\_good_r((s.lower_r, t])$ is trivially enabled). Then, by definition of $receive\_pkt_{sr}(m,t)$ in $A_{\mathrm{G}}^{p\,\prime}$ we have $u''.last_s \neq t$ or $u''.mode_s \neq \mathtt{send}$. By the precondition of $shrink\_good_r((s.lower_r, t])$, we must show two conditions.

   1) First, since $mode_s$ ranges over $\{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}$ in $A_{\mathrm{C}}^{h\,\prime}$, we have $u.mode_s (= u''.mode_s) \neq$ $\mathtt{needid}$. Thus, the first condition is satisfied.

   2) Second, assume $u''.mode_s = \mathtt{send}$. We must show that $u''.last_s \notin (s.lower_r, t]$. From above we have $u''.last_s \neq t$. Then since $s'.last_r = u.last_r = u''.last_r = t$, Invariant 10.6 Part 3 implies $t < u''.last_s$. That suffices.

   Thus, $shrink\_good_r((s.lower_r, t])$ is enabled in u''.

   We must show that all state variables of $A_{\mathrm{G}}^{p\,\prime}$ are handled correctly. This is easy for all variables other than $good_r$ by explicit definition of u''.

   For $good_r$ we must show that $u'.good_r = u''.good_r \setminus (s.lower_r, t]$. Since $s'.lower_r = t$ and $s'.upper_r = s.upper_r$, the definitions of R_{CG} and u'' imply $u''.good_r = (s.lower_r, s'.upper_r] \setminus \{t' \mid t' \leq_u t\}$ and $u'.good_r = (t, s'.upper_r)$. Thus, it suffices to show that if $t' \leq_u t$, then $t' \leq t$, but that follows directly from Invariant 10.8 Part 1. That suffices.

2. $s.mode_r = \mathtt{rec}$ or $\neg(s.lower_r < t \leq s.upper_r)$

   We show that $(u, receive\_pkt_{sr}(m,t), u') \in steps(A_{\mathrm{G}}^{p\,\prime})$. This step (and execution fragment) clearly has the right trace.

   We consider subcases.

   (a) $mode_r = \mathtt{rec}$.

   In this case the only difference between s and s' is that $s'.sr$ is missing one element $((m,t), t'')$ compared to $s.sr$. Thus, the only difference between u and u' is, by definition of R_{CG}, that $u'.sr$ is missing one packet (m,t) compared to $u.sr$.

   Since $s.mode_r = \mathtt{rec}$ we have $u.mode_r = \mathtt{rec}$, so in this case it is easy to see that $(u, receive\_pkt_{sr}(m,t), u') \in steps(A_{\mathrm{G}}^{p\,\prime})$.

   (b) $mode_r \neq \mathtt{rec}$, $\neg(s.lower_r < t \leq s.upper_r)$, and $last_r < t \leq lower_r$.

   In this case the only difference between s and s' is that $s'.nack\text{-}buf_r = s.nack\text{-}buf_r \,\hat{}\, t$ and $s'.sr$ is missing one element $((m,t), t'')$ compared to $s.sr$. Then the definition of R_{CG} implies that u' and u are the same except that $u'.nack\text{-}buf_r = u.nack\text{-}buf_r \,\hat{}\, t$ and $u'.sr$ is missing one packet (m,t) compared to $u.sr$.

   Now, the definition of R_{CG} implies that $u.mode_r \neq \mathtt{rec}$ and $t \notin u.good_r$, and since $s.last_r < t$, $u.last_r \neq t$. Thus, by definition of $receive\_pkt_{sr}(m,t)$ in $A_{\mathrm{G}}^{p\,\prime}$, it is easy to see that $(u, receive\_pkt_{sr}(m,t), u') \in steps(A_{\mathrm{G}}^{p\,\prime})$.

(c) $mode_r \neq \texttt{rec}$, $\neg(s.lower_r < t \leq s.upper_r)$, $\neg(last_r < t \leq lower_r)$, $mode_r = \texttt{idle}$, and $last_r = t$.
In this case the only difference between s and s' is that $s'.mode_r = \texttt{ack}$ and $s'.sr$ is missing one element $((m,t),t'')$ compared to $s.sr$. Then the definition of R_{CG} implies that u' and u are the same except that $u.mode_r = \texttt{idle}$, $s.mode_r = \texttt{ack}$ and $u'.sr$ is missing one packet (m,t) compared to $s.sr$.
We have, by definition of R_{CG} that $u.mode_r = \texttt{idle}$ and $t \notin u.good_r$. Furthermore, the case assumption and Invariant 10.12 imply that $s.last_r > 0$, so, by the definition of R_{CG}, $u.last_r = s.last_r = t$. Then, by definition of $receive\_pkt_{sr}(m,t)$ in $A_{\mathrm{G}}^{p\,\prime}$, it is easy to see that $(u, receive\_pkt_{sr}(m,t), u') \in steps(A_{\mathrm{G}}^{p\,\prime})$.

(d) $mode_r \neq \texttt{rec}$, $\neg(s.lower_r < t \leq s.upper_r)$, $\neg(last_r < t \leq lower_r)$, and $(mode_r \neq \texttt{idle}$ or $last_r \neq t)$.
In this case the only difference between s and s' is that $s'.sr$ is missing one element $((m,t),t'')$ compared to $s.sr$. Thus, the only difference between u and u' is, by definition of R_{CG}, that $u'.sr$ is missing one packet (m,t) compared to $u.sr$.
We must show that the definition of $receive\_pkt_{sr}(m,t)$ in $A_{\mathrm{G}}^{p\,\prime}$ allows all state variables except sr to be unchanged. (The change to sr is as required by $receive\_pkt_{sr}(m,t)$.)
As in the previous case we have $u.mode_r \neq \texttt{rec}$ and $t \notin u.good_r$. Thus, according to the definition of $receive\_pkt_{sr}(m,t)$ for the receiver of $A_{\mathrm{G}}^{p\,\prime}$, the required changes to the state variables are not given by the first alternative in the embedded if-statement.
Now assume $t \neq s.last_r$ (cf. the case assumption). Then also $t \neq u.last_r$. Then, by definition of $receive\_pkt_{sr}(m,t)$ in $A_{\mathrm{G}}^{p\,\prime}$, we see that in order for $A_{\mathrm{G}}^{p\,\prime}$ to allow $u'.nack\text{-}buf_r = u.nack\text{-}buf_r$ it suffices to show that $t \neq u.last_s$. By the case assumption and Invariant 10.2 Part 2, Invariant 10.3 Part 1, and Invariant 10.6 Part 1, $t < s.last_r$. Thus, $u.last_r = s.last_r > t$. That suffices.
Finally, assume that $t = s.last_r$ and $mode_r \neq \texttt{idle}$. Then it is clearly the case that $(u, receive\_pkt_{sr}(m,t), u') \in steps(A_{\mathrm{G}}^{p\,\prime})$.

$a = receive\_pkt_{rs}(t,b)$

We show that $(u, receive\_pkt_{rs}(t,b), u') \in steps(A_{\mathrm{G}}^{p\,\prime})$. This step (and finite execution fragment) clearly has the right visible trace.

Since $(t,b) \in packets(s.rs)$, the definition of R_{CG} gives $(t,b) \in u.rs$. Thus, $receive\_pkt_{rs}(t,b)$ is enabled in u.

We consider cases based on the if-statement in the definition of $receive\_pkt_{sr}(t,b)$ of the sender in $A_{\mathrm{C}}^{h\,\prime}$. In both cases a $((t,b),t')$ element of $s.rs$ gets removed and this corresponds, by the definition of R_{CG}, to removing a (t,b) element from $u.rs$, but this is as required by the definition of $receive\_pkt_{rs}(t,b)$ in $A_{\mathrm{G}}^{p\,\prime}$. Below we consider the remaining state variables of $A_{\mathrm{G}}^{p\,\prime}$.

Assume $s.mode_s \neq \texttt{send}$ or $s.last_s \neq t$. Then the only difference between s and s' is the change in the channel rs as described above, so the only difference between u' and u is the corresponding change in sr (according to R_{CG}). Now, the definition of R_{CG} implies that $u.mode_s \neq \texttt{send}$ or $u.last_s \neq t$ so we see, from the definition of $receive\_pkt_{sr}(t,b)$ in $A_{\mathrm{G}}^{p\,\prime}$, that $(u, receive\_pkt_{rs}(t,b), u') \in steps(A_{\mathrm{G}}^{p\,\prime})$.

Then, assume $s.mode_s = \texttt{send}$ and $s.last_s = t$. From Invariant 10.13 we have $t > 0$, so the definition of R_{CG} implies that $u.mode_s = \texttt{send}$ and $u.last_s = t$. Thus, the condition of the if-statement in $A_{\mathrm{G}}^{p\,\prime}$ is satisfied. It is now easy to see that the changes made by $A_{\mathrm{C}}^{h\,\prime}$ correspond to allowed changes in ${A^p_{\mathrm{G}}}'$. (Note that $u.last_s = u'.last_s$ but this is allowed by the definition of $receive\_pkt_{rs}(t, b)$ in ${A^p_{\mathrm{G}}}'$).

$\underline{a = choose\_id(t)}$

We show that $(u, prepare, u'', grow\_good_s(t), u'''choose\_id(t), u'''', shrink\_good_s(t), u')$, where u'', u''', and u'''' are defined below, is an execution fragment of ${A^p_{\mathrm{G}}}'$ by showing that $(u, prepare, u'')$, $(u'', grow\_good_s(t), u''')$, $(u''', choose\_id(t), u'''')$, and $(u'''', shrink\_good_s(t), u')$ are steps of ${A^p_{\mathrm{G}}}'$. Clearly the execution fragment has the right visible trace.

$$\begin{aligned} \text{Define } u''.mode_s &= \texttt{needid} \\ u''.good_s &= \emptyset \\ u''.current\text{-}msg_s &= head(u.buf_s) \\ u''.buf_s &= tail(u.buf_s) \\ u''.current\text{-}ok &= (\text{if } u.rec_r \neq \texttt{rec} \text{ then } true \text{ else } u.current\text{-}ok) \\ u''.x &= u.x \quad \text{for the remaining state variables } x \end{aligned}$$

We first consider $(u, prepare, u'')$. From the precondition of the $choose\_id(t)$ steps in ${A^h_{\mathrm{C}}}'$ we have that $s.mode_s = \texttt{idle}$ and $s.buf_s \neq \varepsilon$. This implies, by the definition of R_{CG}, that $u.mode_s = \texttt{idle}$ and $u.buf_s = s.buf_s \neq \varepsilon$. Thus,$prepare$ is enabled in u (and furthermore the definition of u'' is well-defined). Now, by definition of u'', clearly $(u, prepare, u'') \in steps({A^p_{\mathrm{G}}}')$.

$$\begin{aligned} \text{Define } u'''.good_s &= \{t\} \\ u'''.x &= u''.x \quad \text{for the remaining state variables } x \end{aligned}$$

Next, consider $(u'', grow\_good_s(t), u''')$. We have, from the definition of u'', that $u''.mode_s = \texttt{needid}$, so from the definition of $grow\_good_s(t)$ in ${A^p_{\mathrm{G}}}'$ we have to show three conditions in order to show that $grow\_good_s(t)$ is enabled in u''. First, assume $u''.mode_r \neq \texttt{rec}$. We must show $t \in u''.issued_r$. We have $u''.issued_r = u.issued_r = (0, s.upper_r]$ (by definition of u'' and R_{CG}) and $t = s.time_s > s.last_s$ (from the precondition of $choose\_id(t)$ in ${A^h_{\mathrm{C}}}'$), so we must show that $s.time_s \leq s.upper_r$ but that follows from Invariant 10.2 Part 2. Second, assume $u''.current\text{-}ok = true$. We must show $t \in u''.good_r$, thus since $u''.good = u.good_r$, we must show $time_s \in (s.lower_r, s.upper_r]$. The lower bound follows from Invariant 10.7 Part 2 since the precondition of the $choose\_id(t)$ step in ${A^h_{\mathrm{C}}}'$ implies that $s.last_s < s.time_s$. The upper bound is already shown in the treatment of the first part of the precondition above. Third, we must show that $t \notin u''.used_s$, thus we must show that $s.time_s \notin (0, s.last_s]$ but that follows from the precondition of the $choose\_id(t)$ steps in ${A^h_{\mathrm{C}}}'$. Thus, we have shown that $grow\_good_s(t)$ is enabled in u''. Now, by definition of u''' and since $u''.good_s = \emptyset$, obviously $(u'', grow\_good_s(t), u''') \in steps({A^p_{\mathrm{G}}}')$.

$$\begin{aligned} \text{Define } u''''.mode_s &= \texttt{send} \\ u''''.last_s &= t \\ u''''.used_s &= u'''.used_s \,\hat{}\, t \\ u''''.x &= u'''.x \quad \text{for the remaining state variables } x \end{aligned}$$

Next, consider $(u''', choose\_id(t), u'''')$. By the definitions of u'', u''', and R_{CG} we have that $u'''.mode_s = \texttt{needid}$ and $t \in u'''.good_s (= \{t\})$. Thus, $choose\_id(t)$ is enabled in u'''. By definition of u'''' and $choose\_id(t)$, clearly $(u''', choose\_id(t), u'''') \in steps({A^p_{\mathrm{G}}}')$.

Finally, consider $(u'''', shrink\_good_s(t), u')$. From the definition of $shrink\_good_s(t)$ in ${A^p_{\mathrm{G}}}'$ we see that we must show that u'''' and u' are the same except that $u'.good_s = u''''.good_s \setminus \{t\}$. From the definition of R_{CG} and the $choose\_id(t)$ step of ${A^h_{\mathrm{C}}}'$ we have $u'.good_s = \{s'.time_s\} \setminus \{s'.last_s\} = \emptyset$.

Thus, since $u''''.good_s = u'''.good_s = \{t\}$, the condition on $good_s$ is satisfied. It is trivial to check that all other state variables of $A_{\mathrm{G}}^{p\,\prime}$ are handled correctly.

$\underline{a = \textit{increase-lower}_r(t)}$

We show that $(u, \textit{shrink\_good}_r((0,t]), u') \in \textit{steps}(A_{\mathrm{G}}^{p\,\prime})$. This step (and finite execution fragment) clearly has the right visible trace.

From the precondition of $\textit{increase-lower}_r(t)$ in $A_{\mathrm{C}}^{h\,\prime}$ we have $s.mode_r \neq \mathtt{rec}$ and $s.lower_r \leq t < s.time_r - \rho$.

We first show that $\textit{shrink\_good}_r((0,t])$ is enabled in u. If $u.\textit{current-ok} = \textit{false}$ then this is obvious. So assume $u.\textit{current-ok} = \textit{true}$. We must check two conditions. First assume $u.mode_s = \mathtt{needid}$. Then we must show that $(0,t] \cap u.good_s = \emptyset$ which, by definition of R_{CG}, amounts to showing $(0,t] \cap (\{s.time_s\} \setminus \{s.last_s\}) = \emptyset$. Thus, it suffices to show $t < s.time_s$ which, by definition of $\textit{increase-lower}_r(t)$ in $A_{\mathrm{C}}^{h\,\prime}$, is the same as showing $s'.lower_r < s'.time_s$, but this is implied by Invariant 10.3 Part 1 and Invariant 10.10 Part 6, where the latter invariant applies since $u.\textit{current-ok} = \textit{true}$ implies $s.deadline \neq \infty$ which again, by definition of $\textit{increase-lower}_r(t)$, implies $s.deadline \neq \infty$. For the second condition in the precondition we must show, under the assumption that $u.mode_s = \mathtt{send}$, that $u.last_s \neq t$, which is implied by proving $s'.last_s \neq s'.lower_r$. Again, Invariant 10.10 Part 6 gives the result.
Thus, $\textit{shrink\_good}_r((0,t])$ is enabled in u.

To show that $(u, \textit{shrink\_good}_r((0,t]), u') \in \textit{steps}(A_{\mathrm{G}}^{p\,\prime})$ we must finally show that $u'.good_r = u.good_r \setminus \{t\}$ and that all other state variables in $A_{\mathrm{G}}^{p\,\prime}$ have the same values in u and u'. By definition of R_{CG} and $\textit{increase-lower}_r(t)$ we have $u.good_r = (s.lower, s.upper_r] = (s.lower_r, s'.upper_r]$ and $u'.good_r = (t, s'.upper_r]$, so since $t \geq s.lower_r$, by the precondition of $\textit{increase-lower}_r(t)$, it is easy to see that the condition for $good_r$ is satisfied. Since the $\textit{increase-lower}_r(t)$ step of $A_{\mathrm{C}}^{h\,\prime}$ only changes $lower_r$ and $lower_r$ is only used in the definition of R_{CG} to define $good_r$, it is obvious that all state variables, but $good_r$, of $A_{\mathrm{G}}^{p\,\prime}$ have the same values in u and u'.

$\underline{a = \textit{increase-upper}_r(t)}$

We show that then $(u, \textit{grow\_good}_r((s.upper_r, t]), u') \in \textit{steps}(A_{\mathrm{G}}^{p\,\prime})$. This step (and finite execution fragment) clearly has the right visible trace.

Since, by definition of R_{CG}, $u.issued_r = (0, s.upper_r]$, it is obvious that $u.issued_r \cap (s.upper_r, t] = \emptyset$ and that $|\mathsf{T} \setminus (u.issued_r \cup (s.upper_r, t])| = \infty$. Thus, a $\textit{grow\_good}_r((s.upper_r, t])$ step is enabled in u.

Now we first show that $u'.issued_r = u.issued_r \cup (s.upper_r, t]$ and $u'.good_r = u.good_r \cup (s.upper_r, t]$, as required by the definition of $\textit{grow\_good}_r((s.upper_r, t])$ in $A_{\mathrm{G}}^{p\,\prime}$. For $issued_r$ we have $u.issued_r = (0, s.upper_r]$ and $u'.issued_r = (0, s'.upper_r] = (0,t]$. Now, since $t \geq s.upper_r$, by the precondition of $\textit{increase-upper}_r(t)$, the condition for $issued_r$ is clearly satisfied. For $good_r$ we similarly have $u.good_s = (s.lower_r, s.upper_r]$ and $u'.good_r = (s'.lower_r, s'.upper_r] = (s.lower_r, t]$. Thus, the condition for $good_r$ is also satisfied.

We must finally show that all other state variables in $A_{\mathrm{G}}^{p\,\prime}$ have the same values in u and u', but this is obvious since the $\textit{increase-upper}_r(t)$ step of $A_{\mathrm{C}}^{h\,\prime}$ only changes $upper_r$, and $upper_r$ is only used in R_{CG} to define $good_r$ and $issued_r$.

$\underline{a = cleanup_r}$

We show that $(u, cleanup_r, u') \in steps(A_G^{p\,\prime})$. This step (and finite execution fagment) clearly has the right visible trace.

By the precondition of $cleanup_r$ we have $s.mode_r \in \{\texttt{idle}, \texttt{ack}\}$ and $s.time_r > s.rm\text{-}time_r + \alpha$. By the definition of R_{CG} and Invariant 10.10, we have $u.mode_r \in \{\texttt{idle}, \texttt{ack}\}$ and $u.mode_s \Longrightarrow u.last_s \neq u.last_r$. Thus, $cleanup_r$ is enabled in u.

It is now easy to see that the variable changes specified by the $cleanup_r$ step of $A_C^{h\,\prime}$ correspond to the required variable changes of the $cleanup_r$ step of $A_G^{p\,\prime}$. (The change of $rm\text{-}time_r$ in $A_C^{h\,\prime}$ does not affect any of the variables of $A_G^{p\,\prime}$). Thus, $(u, cleanup_r, u') \in steps(A_G^{p\,\prime})$.

$\underline{a = tick_s}$

We consider cases.

1. $s'.time_s = s.time_s$
   In this case clearly $s' = s$ and thus $u' = u$. Then the finite execution fragment u of $A_G^{p\,\prime}$ has the right properties.
2. $s'.time_s \neq s.time_s$
   We show that $(u, shrink\_good_s(s.time_s), u'', grow\_good_s(s'.time), u')$, where u'' is defined below, is a finite execution fragment of $A_G^{p\,\prime}$ by showing that $(u, shrink\_good_s(s.time_s), u'')$ and $(u'', grow\_good_s(s'.time), u')$ are steps of $A_G^{p\,\prime}$. Clearly this execution fragment has the right visible trace.

   Define $u''.good_s = \emptyset$
   $u''.x = u.x$ for the remaining state variables x

   First, consider $(u, shrink\_good_s(s.time_s), u'')$. Note that trivially $shrink\_good_s(s.time_s)$ is enabled in u. We check that all state variables of $A_G^{p\,\prime}$ are handled correctly. By the definition of R_{CG} we have $u.good_s \subseteq \{s.time_s\}$. Then, since $u''.good_s = \emptyset$, $good_s$ is handled correctly. By definition all other variables of $A_G^{p\,\prime}$ have the same values in u and u'', which is also as required by the definition of $shrink\_good_s(s.time_s)$ in $A_G^{p\,\prime}$.

   Then, consider $(u'', grow\_good_s(s'.time), u')$. By definition of R_{CG} (and the fact that $mode_s$ ranges over $\{\texttt{idle}, \texttt{send}, \texttt{rec}\}$ in $A_C^{h\,\prime}$), we have $u.mode_s \neq \texttt{needid}$ and consequently, by definition of u'', $u''.mode_s \neq \texttt{needid}$. This shows that $grow\_good_s(s'.time)$ is enabled in u''.

   By Invariant 10.3 Part 1, $s.last_s \leq s.time_s$. The Case Assumption together with the precondition of the $tick_s$ steps of the clock subsystem implies that $s'.time_s > s.time_s$. Then since $s'.last_s = s.last_s$, we have $s'.time_s \neq s'.last_s$. This implies, by definition of R_{CG} that $u'.good_s = \{s'.time_s\}$. Thus, $good_s$ is handled as required by the definition of $grow\_good_s(s'.time)$ in $A_G^{p\,\prime}$. It is easy to see that all the remaining variables of $A_G^{p\,\prime}$ have the same values in u'' and u' which is also as required by the definition of $grow\_good_s(s'.time)$ in $A_G^{p\,\prime}$. That suffices.

$\underline{a = tick_r}$

We show that $u' = u$. Then the finite execution fragment u clearly has the right properties.

Now, clearly $u' = u$ since the $tick_r$ step of $A_C^{h\,\prime}$ only changes $time_r$ and $ctime_r$, and these variables are not mentioned in the definition of R_{CG}.

This concludes the simulation proof.

■

This simulation result allows us to prove that $A_{\mathrm{C}}^{h\,\prime}$ safely implements $A_{\mathrm{G}}^{p\,\prime}$, and, in turn, that A_{C} safely implements G^p.

Lemma 10.16

$A_{\mathrm{C}}^{h\,\prime} \sqsubseteq_{\mathrm{St}} A_{\mathrm{G}}^{p\,\prime}$

Proof

Immediate by Lemmas 10.15 and 5.23.

■

Theorem 10.17

$A_{\mathrm{C}} \sqsubseteq_{\mathrm{St}} \mathit{patient}(A_{\mathrm{G}})$

Proof

By Lemma 10.16 and Lemma 5.29 we get

$A'_{\mathrm{C}} \sqsubseteq_{\mathrm{St}} \mathit{patient}(A'_{\mathrm{G}})$

which by substitutivity (Lemma 2.33) implies

$A'_{\mathrm{C}} \setminus \mathcal{A}_{\mathrm{C}} \sqsubseteq_{\mathrm{St}} \mathit{patient}(A'_{\mathrm{G}}) \setminus \mathcal{A}_{\mathrm{C}}$

which, by definition of $\mathcal{A}_{\mathrm{G}}$ and $\mathcal{A}_{\mathrm{C}}$, gives

$A'_{\mathrm{C}} \setminus \mathcal{A}_{\mathrm{C}} \sqsubseteq_{\mathrm{St}} \mathit{patient}(A'_{\mathrm{G}}) \setminus \mathcal{A}_{\mathrm{G}}$

By Proposition 2.38 we then get

$A'_{\mathrm{C}} \setminus \mathcal{A}_{\mathrm{C}} \sqsubseteq_{\mathrm{St}} \mathit{patient}(A'_{\mathrm{G}} \setminus \mathcal{A}_{\mathrm{G}})$

which finally, by definition of A_{C} and A_{G}, gives the result

$A_{\mathrm{C}} \sqsubseteq_{\mathrm{St}} \mathit{patient}(A_{\mathrm{G}})$

■

10.5.4 Correctness

The liveness proof presented in this section is significantly simpler than the liveness proof in the proof of correctness of H. The reason is that the sender and receiver processes are very similar in C and G, and that the packets sent to the channels at the two levels are of the same type. Recall that at the H level, additional packet types (`needid`, `accept`, and `done`) made the liveness proof very complex.

Actually, the only preliminary lemmas we need, express the fact that the timing requirements of the timed channels are sufficient to guarantee the liveness requirements specified for the untimed channels used at the G level.

Lemma 10.18

1. $\mathit{exec}^{\infty}(A_{\mathrm{C}}^{h\,\prime}) \models \forall p : (\Box\Diamond\langle \mathit{send\_pkt}_{sr}(p)\rangle \Longrightarrow \Box\Diamond\langle \mathit{receive\_pkt}_{sr}(p)\rangle)$
2. $\mathit{exec}^{\infty}(A_{\mathrm{C}}^{h\,\prime}) \models \forall p : \mathit{WF}(\mathit{receive\_pkt}_{sr}(p))$

Proof

We only sketch the proofs.

1. Consider any packet p and assume α is an admissible execution of ${A_C^h}'$ such that $\alpha \models \Box\Diamond\langle \mathit{send\_pkt}_{sr}(p)\rangle$, thus, $\mathit{send\_pkt}_{sr}(p)$ occurs infinitely often in α. For every k occurrences of $\mathit{send\_pkt}_{sr}(p)$ at least one element of the form (p,t), where t is the send time for p, is placed in sr. By the maximum channel delay d, we have that not later than real time $t+d$ a $\mathit{receive\_pkt}_{sr}(p)$ action occurs. Then, since α is admissible, for every k occurrences of $\mathit{send\_pkt}_{sr}(p)$ in α there is at least one occurrence of $\mathit{receive\_pkt}_{sr}(p)$. Thus, since there are infinitely many occurrences of $\mathit{send\_pkt}_{sr}(p)$, there are infinitely many occurrences of $\mathit{receive\_pkt}_{sr}(p)$, i.e, $\alpha \models \Box\Diamond\langle \mathit{receive\_pkt}_{sr}(p)\rangle$. That suffices.

2. Consider any packet p and assume α is an admissible execution of ${A_C^h}'$ such that for some suffix α_1 of α, $\alpha_1 \models \Box(p \in \mathit{packets}(sr))$ (the enabling condition for $\mathit{receive\_pkt}_{sr}(p)$ is $(p \in \mathit{packets}(sr))$). Then, for any time t, a $\mathit{receive\_pkt}_{sr}(p)$ action occurs not later than time $t+d$ since all packets much have left the channel after at most the channel delay time d. Then, since α is admissible, infinitely many occurrences of $\mathit{receive\_pkt}_{sr}(p)$ occur in α_1. Thus, $\alpha_1 \models \Box\Diamond\langle \mathit{receive\_pkt}_{sr}(p)\rangle$. That suffices by definition of WF.

■

Lemma 10.19

1. $\mathit{exec}^\infty({A_C^h}') \models \forall p : (\Box\Diamond\langle \mathit{send\_pkt}_{rs}(p)\rangle \Longrightarrow \Box\Diamond\langle \mathit{receive\_pkt}_{rs}(p)\rangle)$
2. $\mathit{exec}^\infty({A_C^h}') \models \forall p : \mathit{WF}(\mathit{receive\_pkt}_{rs}(p))$

Proof

Similar to the proof of Lemma 10.18.

■

We can now show the main part of the liveness proof, namely, if α is a live execution of $\mathrm{C}^{h'}$ and α' is an execution of $\mathrm{G}^{p'}$ such that $(\alpha,\alpha') \in R_{\mathrm{CG}}$, then α' is live. As usual, we prove this result by contradiction. Thus, we assume that α' is not live and then derive a contradiction with the fact that α is live.

Lemma 10.20

Let $\alpha \in \mathit{exec}^\infty({A_C^h}')$ *and* $\alpha' \in \mathit{exec}^\infty({A_G^p}')$ *be arbitrary admissible executions of* ${A_C^h}'$ *and* ${A_G^p}'$*, respectively, with* $(\alpha,\alpha') \in R_{\mathrm{CG}}$*. Assume* $\alpha \models Q_{\mathrm{C}}$*. Then* $\alpha' \models Q_{\mathrm{G}}$*.*

Proof

We prove the conjecture by contradiction. Thus,

ASSUME: $\alpha' \not\models Q_{\mathrm{G}}$
PROVE: False

$\langle 1 \rangle 1.$ $\alpha' \models \neg\, WF(C_{\mathrm{G},s/r1}) \vee$
$\neg\Box(\Box(mode_s = \mathtt{needid} \wedge mode_r \neq \mathtt{rec}) \Longrightarrow \Diamond\langle C_{\mathrm{G},s/r2}\rangle) \vee$
$\neg\, WF(C_{\mathrm{G},s/r3}) \vee$
$\neg\, WF(C_{\mathrm{G},s/r4}) \vee$
$\neg\forall p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{sr}(p)\rangle) \vee$
$\neg\forall p :\ WF(receive\_pkt_{sr}(p)) \vee$
$\neg\forall p : (\Box\Diamond\langle send\_pkt_{rs}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{rs}(p)\rangle) \vee$
$\neg\forall p :\ WF(receive\_pkt_{sr}(p))$

PROOF: Immediate by the Assumption, the definition of Q_{G}, and the Boolean operators.

$\langle 1 \rangle 2.$ CASE: $\alpha' \models \neg\, WF(C_{\mathrm{G},s/r1})$

$\langle 2 \rangle 1.$ $\alpha' \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}) \wedge \Diamond\Box\neg\langle C_{\mathrm{G},s/r1}\rangle$

PROOF: From Case Hypothesis $\langle 1 \rangle$ by noting that $enabled(C_{\mathrm{G},s/r1}) = (mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\})$ and by expanding WF.

$\langle 2 \rangle 2.$ $\alpha \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}) \wedge \Diamond\Box\neg\langle C_{\mathrm{G},s/r1} \setminus \{prepare\}\rangle$

PROOF: From $\langle 2 \rangle 1$ by definition of R_{CG} and by Lemmas 5.25 and 5.26.

$\langle 2 \rangle 3.$ $\alpha \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}) \wedge$
$\Diamond\Box\neg\langle C_{\mathrm{G},s/r1} \setminus \{prepare\}\rangle \wedge$
$\Diamond\Box\neg\langle\{choose\_id(t) \mid t \in \mathsf{T}\}\rangle$

PROOF: By $\langle 2 \rangle 2$ and the definition of $A_{\mathrm{C}}^{h'}$. Consider a suffix α_1 of α that satisfies $\alpha_1 \models \Box\neg\langle C_{\mathrm{G},s/r1} \setminus \{prepare\}\rangle$. Then if $mode_s$ is `send` it will stay `send` unless a crash occurs, in which case $mode_s$ changes to `rec`. However, once in mode `rec`, the sender will stay there since no $recover_s$ occurs in α_1. Now, $choose\_id(t)$ actions can only occur if $mode_s = \mathtt{idle}$. However, then the sender never returns to mode `idle` again, as we have just seen. Thus, there is at most one occurrence of a $choose\_id(t)$ action in α_1. This gives the result.

$\langle 2 \rangle 4.$ $\alpha \models \Diamond\Box(mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}) \wedge \Diamond\Box\neg\langle C_{\mathrm{C},s}\rangle$

PROOF: By $\langle 2 \rangle 3$ and the definition of $C_{\mathrm{C},s}$.

$\langle 2 \rangle 5.$ $\alpha \models \neg\, WF(C_{\mathrm{C},s})$

PROOF: From $\langle 2 \rangle 4$ by using the definitions of WF and $C_{\mathrm{C},s}$.

$\langle 2 \rangle 6.$ Q.E.D.

PROOF: $\langle 2 \rangle 5$ contradicts the assumption that $\alpha \models Q_{\mathrm{C}}$.

$\langle 1 \rangle 3.$ CASE: $\alpha' \models \neg\Box(\Box(mode_s = \mathtt{needid} \wedge mode_r \neq \mathtt{rec}) \Longrightarrow \Diamond\langle C_{\mathrm{G},s/r2}\rangle)$

$\langle 2 \rangle 1.$ $\alpha' \models \Diamond\Box(mode_s = \mathtt{needid} \wedge mode_r \neq \mathtt{rec}) \wedge \Diamond\Box\neg\langle C_{\mathrm{G},s/r2}\rangle$

PROOF: Directly by Assumption $\langle 1 \rangle$.

$\langle 2 \rangle 2.$ $\alpha \models \Diamond\Box(mode_s \notin \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\})$

PROOF: By $\langle 2 \rangle 1$, the definition of R_{CG}, and Lemma 5.26.

$\langle 2 \rangle 3.$ Q.E.D.

PROOF: $\langle 2 \rangle 2$ contradicts the fact that always $mode_s \in \{\mathtt{idle}, \mathtt{send}, \mathtt{rec}\}$ at the C level.

$\langle 1\rangle 4.$ CASE: $\alpha' \models \neg\, WF(C_{\mathrm{G},s/r3})$

$\langle 2\rangle 1.$ $\alpha' \models \Diamond\Box(mode_r = \mathtt{rec} \vee (mode_r = \mathtt{rcvd} \wedge buf_r \neq \varepsilon) \vee mode_r = \mathtt{ack}) \wedge \Diamond\Box\neg\langle C_{\mathrm{G},s/r3}\rangle$

PROOF: By Assumption $\langle 1\rangle$ and the definitions of WF and $enabled(C_{\mathrm{G},s/r3})$.

$\langle 2\rangle 2.$ $\alpha \models \Diamond\Box(mode_r = \mathtt{rec} \vee (mode_r = \mathtt{rcvd} \wedge buf_r \neq \varepsilon) \vee mode_r = \mathtt{ack}) \wedge \Diamond\Box\neg\langle C_{\mathrm{G},s/r3}\rangle$

PROOF: From $\langle 2\rangle 1$ by definition of R_{CG}, the fact that $C_{\mathrm{G},s/r3}$ contains external actions only, and Lemmas 5.25 and 5.26.

$\langle 2\rangle 3.$ $\alpha \models \neg\, WP(C_{\mathrm{C},r1})$

PROOF: By $\langle 2\rangle 2$ using the definition of WF, the fact that $C_{\mathrm{C},r1} = C_{\mathrm{G},s/r3}$, and the definition of $enabled(C_{\mathrm{C},r1})$.

$\langle 2\rangle 4.$ Q.E.D.

PROOF: $\langle 2\rangle 3$ contradicts the assumption that $\alpha \models Q_{\mathrm{C}}$.

$\langle 1\rangle 5.$ CASE: $\alpha' \models \neg\, WF(C_{\mathrm{G},s/r4})$

$\langle 2\rangle 1.$ Q.E.D.

PROOF: Similar to Case $\langle 1\rangle 4$ we get $\alpha \models \neg\, WF(C_{\mathrm{C},r2})$, which contradicts the assumption that $\alpha \models Q_{\mathrm{C}}$.

$\langle 1\rangle 6.$ CASE: $\alpha' \models \neg\forall p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \implies \Box\Diamond\langle receive\_pkt_{sr}(p)\rangle)$

$\langle 2\rangle 1.$ $\alpha' \models \exists p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle)$

PROOF: Directly from Assumption $\langle 1\rangle$.

$\langle 2\rangle 2.$ $\alpha \models \exists p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle)$

PROOF: By $\langle 2\rangle 2$, Lemma 3.5 Parts 7 and 8, and Lemma 5.25.

$\langle 2\rangle 3.$ $\alpha \models \neg\forall p : (\Box\Diamond\langle send\_pkt_{sr}(p)\rangle \implies \Box\Diamond\langle receive\_pkt_{sr}(p)\rangle)$

PROOF: Directly from $\langle 2\rangle 2$.

$\langle 2\rangle 4.$ Q.E.D.

PROOF: $\langle 2\rangle 3$ contradicts Lemma 10.18 Part 1.

$\langle 1\rangle 7.$ CASE: $\alpha' \models \neg\forall p : WF(receive\_pkt_{sr}(p))$

$\langle 2\rangle 1.$ $\alpha' \models \exists p : \neg\, WF(receive\_pkt_{sr}(p))$

PROOF: Directly from Assumption $\langle 1\rangle$.

$\langle 2\rangle 2.$ $\alpha' \models \exists p : \Diamond\Box(p \in sr) \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle$

PROOF: By $\langle 2\rangle 1$ and the definition of WF.

$\langle 2\rangle 3.$ $\alpha \models \exists p : \Diamond\Box(p \in packets(sr)) \wedge \Diamond\Box\neg\langle receive\_pkt_{sr}(p)\rangle$

PROOF: By $\langle 2\rangle 2$, Lemma 3.5 Parts 7 and 8, the definition of R_{CG}, and Lemmas 5.25 and 5.26.

$\langle 2\rangle 4.$ $\alpha \models \neg\forall p : WF(receive\_pkt_{sr}(p))$

PROOF: Directly from $\langle 2\rangle 3$ and the definition of WF.

$\langle 2\rangle 5$. Q.E.D.

PROOF: $\langle 2\rangle 4$ contradicts Lemma 10.18 Part 2.

$\langle 1\rangle 8$. CASE: $\alpha' \models \neg\forall p : (\Box\Diamond\langle send\_pkt_{rs}(p)\rangle \Longrightarrow \Box\Diamond\langle receive\_pkt_{rs}(p)\rangle)$

PROOF: Similar to $\langle 1\rangle 6$ using Lemma 10.19 Part 1.

$\langle 1\rangle 9$. CASE: $\alpha' \models \neg\forall p : WF(receive\_pkt_{rs}(p))$

PROOF: Similar to $\langle 1\rangle 7$ using Lemma 10.19 Part 2.

$\langle 1\rangle 10$. Q.E.D.

PROOF: By $\langle 1\rangle 1$ and the exhaustive cases $\langle 1\rangle 2$–$\langle 1\rangle 9$.

■

With this result, the timed refinement mapping result of the previous section, and Lemma 5.24 we can prove that $\mathrm{C}^{h'}$ correctly implements $\mathrm{G}^{p'}$.

Lemma 10.21

$\mathrm{C}^{h'} \sqsubseteq_{\mathrm{Lt}} \mathrm{G}^{p'}$

Proof

Immediate by Lemmas 10.15, 10.20, and 5.24.

■

This lemma allows us to prove that H correctly implements *patient*(G).

Theorem 10.22

$\mathrm{C} \sqsubseteq_{\mathrm{Lt}} patient(\mathrm{G})$

Proof

By Lemma 10.21 and Lemma 5.30 we get
$\mathrm{C}' \sqsubseteq_{\mathrm{Lt}} patient(\mathrm{G}')$
which by substitutivity (Lemma 2.33) implies
$\mathrm{C}' \setminus \mathcal{A}_{\mathrm{C}} \sqsubseteq_{\mathrm{Lt}} patient(\mathrm{G}') \setminus \mathcal{A}_{\mathrm{C}}$
which, by definition of $\mathcal{A}_{\mathrm{G}}$ and $\mathcal{A}_{\mathrm{C}}$, gives
$\mathrm{C}' \setminus \mathcal{A}_{\mathrm{C}} \sqsubseteq_{\mathrm{Lt}} patient(\mathrm{G}') \setminus \mathcal{A}_{\mathrm{G}}$
By Proposition 2.38 we then get
$\mathrm{C}' \setminus \mathcal{A}_{\mathrm{C}} \sqsubseteq_{\mathrm{Lt}} patient(\mathrm{G}' \setminus \mathcal{A}_{\mathrm{G}})$
which finally, by definition of C and G, gives the result
$\mathrm{C} \sqsubseteq_{\mathrm{Lt}} patient(\mathrm{G})$

■

Finally, we can state and prove the main result, namely that C correctly implements *patient*(S).

Theorem 10.23

$\mathrm{C} \sqsubseteq_{\mathrm{Lt}} patient(\mathrm{S})$

Proof

By Theorems 7.18 and 8.19 and the fact that $\sqsubseteq_{\mathrm{L}}$ is transitive, we have G $\sqsubseteq_{\mathrm{L}}$ S. Then the Embedding Theorem (Theorem 2.37) implies *patient*(G) $\sqsubseteq_{\mathrm{Lt}}$ *patient*(S). This, Theorem 10.22, and the fact that $\sqsubseteq_{\mathrm{Lt}}$ is transitive finally give the result.

■

10.6 A "Weak" Clock-Based Protocol

In the previous section we have considered the Clock-Based Protocol C and shown that it correctly implements the *patient* version of the specification S. In the specification of C we have made some timing assumptions. Specifically, we have assumed a certain *channel retry number* k and a *maximum channel delay* d. Now, what if these assumptions are somehow violated in a physical implementation of the C protocol? What if a communication wire is damaged during some construction work and rerouting leads to a transmission delay greater than d for some packet p? Could the C protocol then suddenly reorder or duplicate messages? The answer is "no". C is in [LSW91] designed to guarantee ordered at-most-once delivery even if all the timing assumptions are violated. However, in case of timing violation the system might lose messages even if no crashes occur, but message loss is generally considered less damaging than duplication.

We suspect that this scenario is general for timing-based communication protocols: without timing assumptions the protocols satisfy some *minimal* requirements (like at-most-once message delivery), and with timing assumptions the protocols satisfy additional properties (like exactly-once message delivery in the absence of crashes).

Our proofs above do not indicate that C guarantees at-most-once delivery even if the timing assumptions are violated. A formal proof of this property would show that a "weak" version of C with no timing assumptions safely implements a "weak" version of S that allows messages to be lost at any time. Note, that the reason why we only need to prove *safe* implementation as opposed to correct implementation is that "at-most-once message delivery" is a safety property.

In order not to have to redo many of the proofs above when performing the proof between the weak versions of the protocols, we think that the proofs should be structured as follows: first prove that the weak version of C safely implements the weak version of S. Then add the additional assumptions, prove additional invariants, and extend the first proof to prove correct implementation.

In a temporal logic setting, like TLA [Lam91], "additional assumptions" are added as new conjuncts to the specifications. Proof of safe implementation, which is expressed as implication in the logic, should then use the new conjuncts of the specification to prove the new conjuncts of the implementation. Exactly how this should be performed in our setting is left for future research.

10.7 The Clock-Based Protocol With One Receiver and Multiple Senders

Consider the situation depicted in Figure 10.2. The picture shows a situation where several receivers—each interacting with a single sender—are placed on the same node. Thus, n copies of the sender, receiver, and channels from above are put in parallel. Instead of implementing n identical copies of the receiver on the receiver node, a single optimized process can be designed

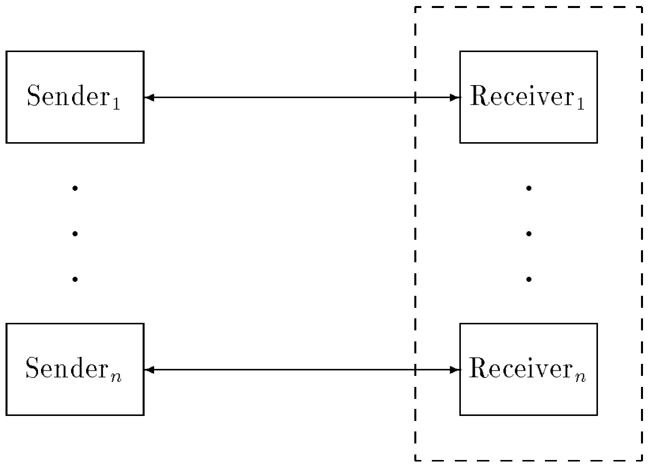


Figure 10.2

The Clock-Based Protocol with several receivers on the same node.

that implements the parallel composition of the receivers. Then, due to the substitutivity results for live timed I/O automata (Proposition 2.33), such a multiple-sender receiver senders (called the ms-receiver) will work in concert with the n senders. Below, we let ss-receiver denote the single-sender receiver from above.

In [LSW91], the receiver of the Clock-Based Protocol is in fact designed to handle multiple senders. This receiver has a structure very similar to the ss-receiver. However, it is optimized so that only one single $upper_r$ variable is needed. This is important since $upper_r$ variables must be kept stable and stable updates are expensive. Furthermore, "old" $lower_r$ variables, i.e., $lower_r$ variables for senders that have not sent messages for a long time, can be cleaned up such that sufficient information about these old variables can be kept in a single common $lower_r$ variable.

This section discribes the design of the ms-receiver of [LSW91] and sketches the proof that it implements the parallel composition of n ss-receiver. It turns out that because of the similarities between the ms-receiver and the ss-receiver, the proof is very simple.

Figure 10.3 shows the visible actions of the ms-receiver. There are n versions of the channel actions, receive message actions, and recovery actions but only one of both $crash_r$ and $tick_r$. This user interface is then the same as one would get by composing n copies of the ss-receiver in parallel after indexing all locally-controlled actions with the index of the ss-receiver. It may seem strange to have a recovery action for each index; however, since the ms-receiver should implement and, thus, have the same user interface as the parallel composition of n (renamed) ss-receivers, and since live timed I/O automata cannot synchronize on output actions (like recovery), it is inevitable that the ms-receiver has n recovery actions. One should, thus, think of the ms-receiver as offering recovery of its n parts, one by one.

Let $\mathrm{C}_{ms,r}$ be a live timed I/O automaton modeling the ms-receiver. It should, then, be proved that

$$\mathrm{C}_{ms,r} \sqsubseteq_{\mathrm{Lt}} \mathrm{C}_{r,1} \| \cdots \| \mathrm{C}_{r,n}$$

where $\mathrm{C}_{r,i} \triangleq \rho_i(\mathrm{C}_r)$ and the function ρ_i maps each locally-controlled action of C_r to an indexed

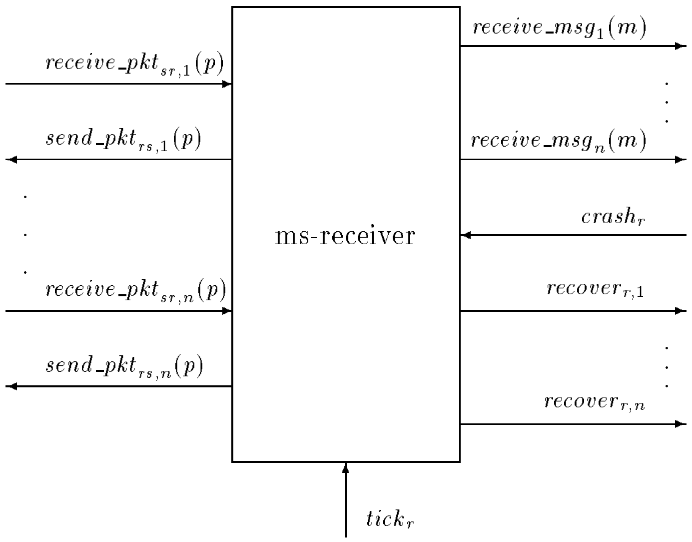


Figure 10.3

The visible actions of the ms-receiver.

version of the same action, and is the identity mapping for the remaining actions. For instance, ρ_i maps *receive_pkt*$_{sr}(p)$ to *receive_pkt*$_{sr,i}(p)$. (Actually, the processes $\mathrm{C}_{r,1}, \ldots, \mathrm{C}_{r,n}$ are not compatible in the strong sense where the ordinary state variable names of different processes are required to be non-overlapping. So, for present purposes, assume that all state variables of $\mathrm{C}_{r,i}$ (except *now*) are indexed with i.)

We do not define $\mathrm{C}_{ms,r}$ completely formally but sketch how it works. First, recall that in C_r, *lower*$_r$ indicates a lower bound on timestamps that the receiver will accept. Every time a new message is accepted, *lower*$_r$ is advanced to the timestamp of that message. Furthermore, special *increase-lower*$_r$ steps are in C_r allowed to increase *lower*$_r$ as long as it is kept small enough to allow very slow messages from the sender to be accepted.

$\mathrm{C}_{ms,r}$ contains n versions (*lower*$_{r,1}, \ldots,$ *lower*$_{r,n}$) of *lower*$_r$—one for each sender—and each variable *lower*$_{r,i}$ remembers the last timestamp received from the ith sender in order to ensure that only messages with later timestamps will be accepted from that sender in the future. In $\mathrm{C}_{ms,r}$, *lower*$_{r,i}$ is only advanced when packets are accepted from the ith sender, i.e., in *receive_pkt*$_{sr,i}(p)$ steps.

Now, $\mathrm{C}_{ms,r}$ furthermore contains a *common-lower*$_r$ variable. This variable is increased in special *increase-common-lower*$_r$ steps, and whenever it advances past the value of a *lower*$_{r,i}$ variable, this *lower*$_{r,i}$ variable is changed to `nil`, i.e., is cleaned up. Thus, *common-lower*$_r$ captures all relevant information about the timestamps that must be accepted from senders that have not sent for a while, as long as *common-lower*$_r$ is kept sufficiently small.

Also, $\mathrm{C}_{ms,r}$ only needs a single *upper*$_r$ variable, which gives the upper bound on timestamps that can be accepted from any sender.

Figure 10.4 shows how an *increase-common-lower*$_r$ step changes a *lower*$_{r,i}$ variable to `nil`. In situation a), $\mathrm{C}_{ms,r}$ will accept timestamps in the interval (*common-lower*$_r$, *upper*$_r$] from sender

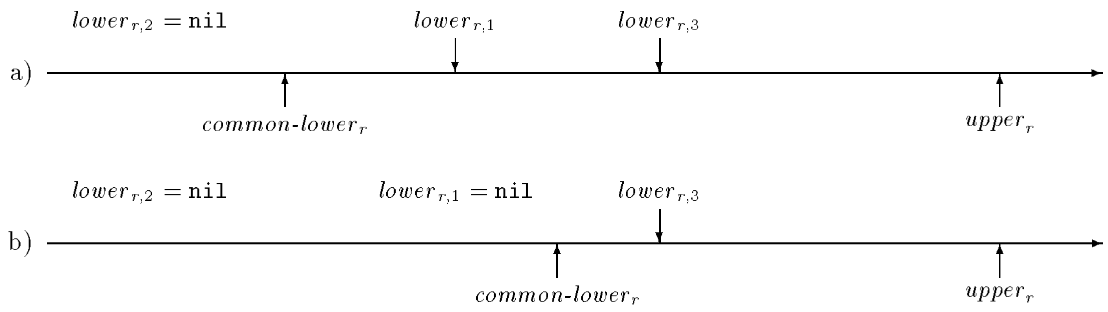


Figure 10.4

The difference between situation a) and b) is that an *increase-common-lower*$_r$ step of $C_{ms,r}$ has advanced *common-lower*$_r$ and thereby has cleaned up $lower_{r,3}$ (by changing it to `nil`).

2 and timestamps in the interval $(lower_{r,i}, upper_r]$ from sender $i \in \{1,3\}$. In situation b), $lower_{r,1}$ has been cleaned up and $C_{ms,r}$ will consequently now only accept timestamps in the interval $(common\text{-}lower_r, upper_r]$ from sender 1. However, this is safe since *common-lower*$_r$ is kept sufficiently small (in the same way the $lower_r$ variable is kept sufficiently small in C_r).

All other variables of C_r, except $time_r$, have n versions in $C_{ms,r}$. For instance, $C_{ms,r}$ has the n buffers $buf_{r,1}, \ldots, buf_{r,n}$. However, of course, only one local receiver clock $time_r$ is needed.

We only specify the most interesting steps of $C_{ms,r}$. These are the steps labeled with $receive\_pkt_{sr,i}(m,t)$ or $increase\text{-}common\text{-}lower_r(t)$ actions.

$receive\_pkt_{sr,i}(m,t)$
  Effect:
    if $mode_{r,i} \neq \texttt{rec}$ then
      if $(lower_{r,i} \neq \texttt{nil} \wedge lower_{r,i} < t \leq upper_r) \vee$
        $(lower_{r,i} = \texttt{nil} \wedge common\text{-}lower_r < t \leq upper_r)$ then
        $mode_{r,i} := \texttt{rcvd}$
        $buf_{r,i} := buf_{r,i} \,\hat{}\, m$
        $last_{r,i} := t$
        $rm\text{-}time_{r,i} := \infty$
        $lower_{r,i} := t$
      else if $(lower_{r,i} \neq \texttt{nil} \wedge last_{r,i} < t \leq lower_{r,i}) \vee$
          $(lower_{r,i} = \texttt{nil} \wedge last_{r,i} < t \leq common\text{-}lower_r)$ then
        $nack\text{-}buf_{r,i} := nack\text{-}buf_{r,i} \,\hat{}\, t$
      else if $mode_{r,i} = \texttt{idle} \wedge last_{r,i} = t$ then
        $mode_{r,i} := \texttt{ack}$

$increase\text{-}common\text{-}lower_r(t)$
  Precondition:
    $\forall i : (mode_{r,i} \neq \texttt{rec}) \wedge$
    $common\text{-}lower_r \leq t < time_r - \rho$
  Effect:
    $common\text{-}lower_r := t$
    for all i with $lower_{r,i} \neq \texttt{nil}$:
      if $common\text{-}lower_r \geq lower_{r,i}$ then
        $lower_{r,i} := \texttt{nil}$

Note, that the timing constant ρ, which occurs in the definition of *increase-common-lower*$_r$ steps, is the same constant as for the ss-receiver above.

Steps labeled by $crash_r$ should in $C_{ms,r}$ change *all* $mode_{r,i}$ variables to `rec`.

It requires a timed refinement mapping to verify that $C_{ms,r}$ correctly implements $C_{r,1} \| \cdots \| C_{r,n}$. This refinement mapping R_{ms} maps most variables one-to-one. Let s be any state of $C_{ms,r}$. Then $R_{ms}(s)$ is the state u that for all i satisfies

- $u.upper_{r,i} = s.upper_r$.
- $u.time_{r,i} = s.time_r$.
- $u.lower_{r,i} =$ (if $s.lower_{r,i} \neq \texttt{nil}$ then $s.lower_{r,i}$ else $s.common\text{-}lower_r$).
- $u.x = s.x$ for the remaining variables x.

It is fairly straightforward to verify that R_{ms} actually is a timed refinement mapping. The way $lower_{r,i}$ is defined in the mapping implies that a $receive\_pkt_{sr,i}(m,t)$ step of $C_{ms,r}$ directly corresponds to a $receive\_pkt_{sr,i}(m,t)$ step of $C_{r,1} \| \cdots \| C_{r,n}$. In fact, there is the same one-to-one correspondence for all other actions, except for $increase\text{-}common\text{-}lower_r(t)$ and $increase\text{-}upper_r(t)$.

A $increase\text{-}common\text{-}lower_r(t)$ step of $C_{ms,r}$ may change several $lower_{r,i}$ variables to `nil`. This corresponds at the abstract level to these $lower_{r,i}$ variables being advanced. Thus, an $increase\text{-}common\text{-}lower_r(t)$ step of $C_{ms,r}$ corresponds to a series of $increase\text{-}lower_{r,i}(t)$—one for each process identifier i for which $lower_{r,i} = \texttt{nil}$ in $C_{ms,r}$ after the $increase\text{-}common\text{-}lower_r(t)$ step.

An $increase\text{-}upper_r(t)$ step simply corresponds a sequence of steps labeled $increase\text{-}upper_{r,1}(t)$, $\ldots$, $increase\text{-}upper_{r,n}(t)$.

We do not complete the modeling of $C_{ms,r}$ in this report but leave this and the complete simulation and liveness proofs for future work.

Chapter 11

Conclusion

11.1 Summary

This report contains two parts. Part I describes the formal models of [GSSL93] for timed and untimed systems, and the associated simulation-based proof techniques. Also, an extended temporal logic is developed, in which temporal formulas evaluate over executions of alternating states and actions and, thus, are well-suited for describing and reasoning about liveness conditions—in the timed setting via sampling characterizations of timed executions. It is furthermore shown how application of the semantic operators of parallel composition, action hiding, and action renaming is reflected in the syntax.

The proof techniques are used to prove that one system correctly implements a more abstract system. A proof generally consists of three parts. First, several invariants of the systems are proved. Then, secondly, a relation is defined and proved to be a simulation relation from the concrete to the abstract system. During this process, one generally has to go back and prove additional invariants. Finally, a liveness proof builds on top of the simulation result.

Part II presents a case study intended to check the adequacy of the formal framework on large examples. In particular, two practical protocols for solving the at-most-once message delivery problem on channels that may delete, duplicate, and reorder packets are considered. One protocol is the Five-Packet Handshake Protocol of [Bel76], which is the standard protocol for setting up network connections, used in TCP, ISO TP-4, and many other transport protocols. The other protocol is the Clock-Based Protocol of [LSW91], which relies on certain timing assumptions. Both protocols are sufficiently complicated that it seems that formal proof is the only means by which their correctness can be verified.

Both the specification S of the at-most-once message delivery problem and the Five-Packet Handshake Protocol, which we call H, are formalized as live I/O automata, however at very different levels of abstraction. The specification S corresponds closely the the informal description of the at-most-once message delivery problem, and is easily checked to have the desirable behavior. H is expressed as the parallel composition of several components.

The Clock-Based Protocol, which we call C, is formalized as a live timed I/O automaton. A special MMT-specification style is used to specify the sender and receiver in a clear way since the timing restrictions on these components are of the simple form: if a set of actions becomes enabled (or stays enabled after being executed), then an action from the set must be executed after some lower time bound and before some upper time bound, unless the set is disabled in the meantime. C is formalized in the timed model and S in the untimed model. It is argued that in this case correctness of C should be expressed with respect to the *patient* version of S, i.e., the object of the timed model that behaves just like S, except that it allows arbitrary passage of time.

Instead of proving directly that H and C correctly implement S and *patient*(S), respectively, the correctness proof is split into smaller parts by introducing intermediate levels of abstraction. In particular, both H and C can be seen as implementations of an (untimed) Generic Protocol G. By introducing intermediate levels of abstraction, not only do we get the advantage of splitting complicated proofs into smaller parts, we also avoid that proofs of similar parts will have to be repeated in the correctness proofs for both H and C; instead these similar parts are captured in G and in the proof that G correctly implements S. In fact, we believe that G is sufficiently general so that other practical protocols can be proved to be correct implementations of G.

A direct proof that G correctly implements S is still very complicated since it involves a backward simulation, and backward simulations seem to be inherently difficult. Thus, to limit the backward simulation to a development step as small as possible, the Delayed-Decision Specification D was defined. In this way the correctness proof for D requires a backward simulation, whereas the correctness proofs for lower levels of abstraction only require the use of the simpler (timed) refinements (plus the use of history variables).

The report contains full proof of correctness for the protocols. However, some of the proofs are only sketched, when similar formal proofs are found elsewhere in the report.

11.2 Evaluation

The operational models of live (timed) I/O automata, the syntax for describing these, and the proof techniques have proved to provide a powerful formal framework within which both untimed and timed distributed systems can be formalized and proved correct. The abstract specification is close to the informal problem statement and the formalism offers a clear, intuitive, and modular approach to the description of the low-level protocols. In particular, for timed systems, where the only timing restrictions are lower and upper time bounds on progress, the MMT-style offers a clear notation.

It should be noted, however, that the example presented in this report only proves correctness of a timed protocol with respect to the *patient* version of an (untimed) specification. This means that the timing assumptions of the timed protocol are only used to prove certain invariants, whereas the handling of time the simulation proofs is almost trivial. [LA91] deals with timed simulation proofs (with non-*patient* specifications) for MMT-style systems.

Some aspects of performing the correctness proofs are intellectually challenging. In particular, defining simulation relations involves a lot of insight and intuition about the systems, and also finding the sequence of abstract steps that corresponds to a given concrete step requires key intuition. In fact these two aspects of the proofs provide important documentation of the functionality of systems and can be used to convey intuition about these.

However, in a simulation proof one must prove that the sequence of abstract steps has the right properties. This involves checking that the steps are in fact steps of the abstract system, which, in turn, amounts to checking that each variable is handled according to the abstract transition relation. This part of the proof involves a lot of tedious details, and forms a quite sizable part of the total proof. Because of the details, the proof is very difficult to maintain; sometimes, during a proof attempt, one has to go back and change either the abstract or the concrete specification, which may lead to a need to change part of the proof already done. Unless extreme care is taken, such changes are likely to introduce inconsistencies in the proof.

Apart from this, simulation proof techniques scale well to large examples and impose a nice case structure on the proof.

Liveness proofs are also challenging. They, too, require insight into the way the protocols work. The temporal logic offers an expressive way formalize liveness conditions and an ad hoc set of rules. Our liveness proofs are not proofs of validity of temporal formulas, but instead proofs of satisfaction, i.e., that certain executions satisfy the temporal formulas. In the proof steps temporal rules, which have the form of valid implications, meta rules, and semantic reasoning are used. This seems to provide a straightforward way of performing careful liveness proofs by hand.

Live (timed) I/O automata, temporal logic, and simulation-based proof techniques are good tools for formally specifying and verifying timed and untimed communication protocols.

The embedding results of the model tie the untimed and timed models together in a very general and useful coordinated framework that allows proving that a timed system correctly implements an untimed specification.

11.3 Further Work

There is a considerable amount of further work remaining. We have already begun the work of automating simulation proofs in the untimed model, by proving the equivalence of versions of S and D using the Larch Prover [SGG+93, GG91]. We have been pleased with the preliminary results: the prover has not only been able to check our hand proofs, but in fact has been able to fill in many of the details. Current research tries to use the same approach on a timed forward simulation. Future research should consider automation of more complicated simulation proofs.

Second, if the timing assumptions on C are weakened or removed, the resulting algorithm still will not deliver any message more than once; however, it may lose messages even in the absence of a crash. It remains to formulate the weaker specification and prove that the weaker version of C satisfies it.

Third, there are other algorithms that solve the at-most-once message delivery problem, for example, using bounded identifier spaces or cryptographic assumptions. We would like also to verify these, preferably reusing as much of our proofs as possible.

Finally, future research should deal with the extended temporal logic developed in this work, and try to find a basic set of rules that is adequate for the liveness proofs of typical distributed systems. The rules presented in this report, which are specifically tailored for the case study, seem to be a good starting point for such an investigation.

11.4 Conclusions

We can draw several conclusions:

- Live (timed) I/O automata, temporal logic, and simulation-based proof techniques provide a powerful coordinated framework for formally specifying and verifying timed and untimed communication protocols.

- The proof techniques, especially simulation proofs, scale well and are not too difficult to use. It is challenging and requires insight and key intuition to find, e.g., the right simulation relations, and a lot of detailed work to verify these choices. For large proofs, computer assistance is essential to help with the details; however, the insight will always be required.

- Backward simulation proofs are much harder to do than refinement mapping and forward simulation proofs but are necessary in certain situations. It seems to be worthwhile to try to limit the use of backward simulations to as small a development step as possible.
- Many practical protocols can be treated as implementations of a common abstract protocol.
- Verifying a coordinated collection of protocols, rather than just a single isolated protocol, is extremely valuable. It leads to the discovery of useful abstractions, and tends to make the proofs more elegant.
- Doing proofs for realistic communication protocols is feasible now. We predict that it will become more so, and will be of considerable practical importance.

Bibliography

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AL92a] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J. W. de Dakker, C. Huizing, and G. Rozenberg, editors, *Proceedings of REX Workshop "Real-Time: Theory in Practice", Mook, The Netherlands, June 1991*, number 600 in Lecture Notes in Computer Science, pages 1–27. Springer-Verlag, 1992.

[AL92b] M. Abadi and L. Lamport. An old-fashioned recipe for real time. Technical report, DEC, Systems Research Center, October 12 1992.

[Bel76] D. Belsnes. Single message communication. *IEEE Transactions on Communications*, Com-24(2), February 1976.

[GG91] S.J. Garland and J.V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, DEC, Systems Research Center, December 1991.

[GSSL93] R. Gawlick, R. Segala, J. Søgaard-Andersen, and N. Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, MIT, Laboratory for Computer Science, December 1993.

[Jon91] B. Jonsson. Simulations between specifications of distributed systems. In J. C. M. Baeten and J. F. Groote, editors, *Proceedings of CONCUR '91. 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 1991*, number 527 in Lecture Notes in Computer Science, pages 346–360. Springer-Verlag, 1991.

[LA91] N. Lynch and H. Attiya. Using mappings to prove timing properties. Technical Report MIT/LCS/TR-412.d, MIT, Laboratory for Computer Science, October 1991.

[Lam91] L. Lamport. The temporal logic of actions. Research Report 79, DEC, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA, December 1991.

[Lis91] B. Liskov. Practical uses of synchronized clocks in distributed systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–10, 1991.

[LMS85] L. Lamport and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[LSW91] B. Liskov, L. Shrira, and J. Wroclawski. Efficient at-most-once messages based on synchronized clocks. *ACM Transactions on Computer Systems*, 9(2):125–142, May 1991.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT, Laboratory for Computer Science, Cambridge, MA, 02139, April 1987.

[LT89] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[LV92] N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In J. W. de Dakker, C. Huizing, and G. Rozenberg, editors, *Proceedings of REX Workshop "Real-Time: Theory in Practice", Mook, The Netherlands, June 1991*, number 600 in Lecture Notes in Computer Science, pages 397–446. Springer-Verlag, 1992.

[LV93a] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – Part I: Untimed systems, 1993. Submitted for publication. Also, Laboratory for Computer Science, Massachusetts Institute of Technology Technical Memo MIT/LCS/TM-486.

[LV93b] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – Part II: Timing-based systems, 1993. Submitted for publication. Also, Laboratory for Computer Science, Massachusetts Institute of Technology Technical Memo MIT/LCS/TM-487.

[MMT91] M. Merritt, F. Modugno, and M. Tuttle. Time-constrained automata. In *Proceedings of CONCUR'91. 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 1991*, number 527 in Lecture Notes in Computer Science, pages 408–423. Springer-Verlag, 1991.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[SGG+93] J.F. Søgaard-Andersen, S.J. Garland, J.V. Guttag, N.A. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer Aided Verification. 5th International Conference, CAV '93. Elounda, Greece, June/July 1993*, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

Appendix A

Basic Definitions

This appendix gives basic definitions used in this report.

A.1 Record Notation

If a variable or value is of tuple type, e.g., $X \times Y \times Z$, we will use the normal record notation to extract the sub-values. For example if d has type $X \times Y \times Z$, $d.x$ will extract the first component of the tuple, etc.

A.2 Sets

We use standard notation for sets. A set consisting of the elements $e_1, e_2, \ldots$ we write as

$$\{e_1, e_2, \ldots\}$$

and a notation like

$$\{f(i) \mid i \in \mathsf{N} \wedge g(i) = 4\}$$

is used to denote the set of all elements $f(i)$, where i is a natural number such that $g(i) = 4$.

A singleton set with the element e is sometimes written e instead of $\{e\}$. As usual we use $\in$ to express set membership, and $\subset$ and $\subseteq$ to express the proper subset and subset relations, respectively. The empty set is denoted by $\emptyset$. Furthermore we use the normal operators on sets

$\cup$	Union
$\cap$	Intersection
$\overline{}$	Complement (with respect to some given set)
$\setminus$	Set minus

Set Type

For any set S, denote by $\mathcal{P}(S)$ the set of all (finite or infinite) subsets of S.

Cardinality

The cardinality of a set S, written $|S|$, is defined as

$$|S| \;\triangleq\; \begin{cases} n & \text{if } S \text{ has } n \text{ elements} \\ \infty & \text{if } S \text{ has infinitely many elements} \end{cases}$$

A.3 Bags (Multisets)

For bags we use the following operators from the previous section:

$$|s|, \cap, \cup, \in$$

$|s|$ counts the total number of elements (including duplicates) of s.

Bag Type

For any set S, denote by $\mathcal{B}(S)$ the set of all (finite or infinite) bags with elements from S.

A.4 Lists and Sequences

In this report we use the terms "sequence", "list", and "queues" synonymously.

A list l consisting of the elements $e_1, e_2, \ldots$ we will write in one of the ways

$$\begin{array}{lcl} l & = & \langle e_0, e_1, \ldots \rangle \\ l & = & e_0, e_1, \ldots \\ l & = & e_0 e_1 \ldots \end{array}$$

We denote by ε the empty list.

List Type

For any set S, denote by S^* the set of all *finite* lists of elements in S.

Length

The length of a list $l = \langle e_0, e_1, \ldots \rangle$, written $|l|$, is defined as

$$|l| \triangleq \begin{cases} n & \text{if } l \text{ is finite and ends in } e_{n-1} \\ \infty & \text{if } l \text{ is infinite} \end{cases}$$

Head, Tail, Last, and Init

If $l = \langle e_0, e_1, e_2, \ldots \rangle$ is nonempty, define

$$\begin{array}{lcl} head(l) & \triangleq & e_0 \\ tail(l) & \triangleq & \langle e_1, e_2, \ldots \rangle \end{array}$$

If furthermore l is finite and ends in e_{n-1}, then define

$$\begin{array}{lcl} last(l) & \triangleq & e_{n-1} \\ init(l) & \triangleq & \langle e_0, e_1, \ldots, e_{n-2} \rangle \end{array}$$

Concatenation

Concatenation of two lists l_1 and l_2, written $l_1 \,\hat{}\, l_2$ or sometimes $l_1 l_2$, is defined when l_1 is finite. If $l_1 = \langle e_0, \ldots, e_{n-1} \rangle$ and $l_2 = \langle e_n, e_{n+1}, \ldots \rangle$, then define

$$l_1 \,\hat{}\, l_2 \triangleq \langle e_0, \ldots, e_{n-1}, e_n, e_{n+1}, \ldots \rangle$$

List Construction

Let $I = \{i_1, i_2, \ldots\}$ be a set of totally ordered elements with $i_1 < i_2 < \cdots$. Then define

$$\langle f(i) \mid i \in I \wedge P(i) \rangle \quad \triangleq \quad e_{i_1} \,\hat{}\, e_{i_2} \,\hat{}\, \cdots$$

where f is a function, P is a predicate, and

$$e_{i_k} = \begin{cases} f(i_k) & \text{if } P(i_k) \\ \varepsilon & \text{otherwise} \end{cases}$$

Indexing

If $l = \langle e_0, e_1, \ldots \rangle$, then define for all i with $0 \leq i < |l|$

$$l[i] \triangleq e_i$$

We let $dom(l)$ denote the set of indices of any list l. Thus,

$$dom(l) \triangleq \{i \mid 0 \leq i < |l|\}$$

We also let $elems(l)$ be the set of elements in l. Thus,

$$elems(l) \triangleq \{l[i] \mid i \in dom(l)\}$$

If l is nonempty, we denote by $maxidx(l)$ the maximum index in l. Thus,

$$maxidx(l) \triangleq |l| - 1$$

Restriction

If l is a list and S is a set, we let $l \restriction S$ denote the restriction of l to S. For example, $\langle 1, 3, 2, 5, 4 \rangle \restriction \{2, 3, 4, 7\} = \langle 3, 2, 4 \rangle$. Formally,

$$l \restriction S \quad \triangleq \quad \langle l[i] \mid i \in dom(l) \wedge l[i] \in S \rangle$$

Set Operations on Lists

As notational convention we allow set operators like $\in$, $\subseteq$, etc., to operate on lists l. This should just be thought of as a shorthand notation for the same operators operating on $elems(l)$. For instance, $e \in l$ means $e \in elems(l)$ and $l \subseteq S$ means $elems(l) \subseteq S$ for some set S.

A.5 Functions and Mappings

We use the terms "function" and "mapping" synonymously. We use standard notation for function definition and application. When explicitly defining the mapping from elements to elements we use notation like

$$\begin{array}{ll} [& 1 \mapsto 1, \\ & 2 \mapsto 4, \\ & 3 \mapsto 9, \\ & \ldots \\ & 9 \mapsto 81 \quad] \end{array}$$

or equivalently

$$[i \mapsto i^2 \mid 1 \leq i \leq 9]$$

Function Type

A function f mapping elements from S_1 to S_2 has the type

$$S_1 \to S_2$$

We shall only deal with total functions, i.e., $f(s)$ is defined for all elements $s \in S_1$. S_1 is referred to as the *domain* of f and S_2 as the *codomain* of f.

Domain and Range

For any function f, $dom(f)$ denotes the domain of f. The *range* (or *image*) of f is defined as

$$rng(f) \;\triangleq\; \{f(e) \mid e \in dom(f)\}$$

Operations on Functions

For function $f : A \to B$ and $g : C \to D$ with $B \subseteq C$, define the *composition* $f \circ g : A \to D$ such that for all $a \in A$,

$$(f \circ g)(a) = f(g(a))$$

For any function $f : A \to B$ and set S, denote by $f \setminus S$ the function with type $(A \setminus S) \to B$ such that for all $a \in A \setminus S$,

$$(f \setminus S)(a) = f(a)$$

Similarly $f \restriction S$ denotes the function of type $(A \cap S) \to B$ such that for all $a \in A \cap S$

$$(f \restriction S)(a) = f(a)$$

For functions $f_i : A_i \to B_i$, $1 \leq i \leq k$, with disjoint domains, denote by $f_1 \cup \cdots \cup f_k$ the function of type $(A_1 \cup \cdots \cup A_k) \to (B_1 \cup \cdots \cup B_k)$ such that for all $a \in (A_1 \cup \cdots \cup A_k)$

$$(f_1 \cup \cdots \cup f_k)(a) = f_i(a) \qquad \text{if } a \in A_i$$

Appendix B

Proofs from Part I

B.1 Proofs in Chapter 3

Proof of Lemma 3.1:

Let α be an arbitrary execution over $(\mathcal{V}, \mathcal{A})$.

If α is infinite, then $\widehat{\alpha} = \alpha$ and the result trivially follows.

Now, assume α is finite and let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots a_n s_n$. Furthermore, let $j \geq 0$ an arbitrary natural number. Let $a_i = \zeta$ and $s_i = s_n$ for all $i > n$. Then $\widehat{\alpha} = s_0 a_1 s_1 a_2 s_2 \cdots$. We prove the lemma by structural induction over P.

<u>Base Case:</u> P is a step formula

$(\alpha, j) \models P$
iff (by definition)
$(0 \leq j < n$ and $(s_j, a_{j+1}, s_{j+1}) \models P)$ or
$(j \geq n$ and $(s_n, \zeta, s_n) \models P)$
iff (by definition of s_i and a_i for $i > n$)
$0 \leq j$ and $(s_j, a_{j+1}, s_{j+1}) \models P$
iff (by definition)
$(\widehat{\alpha}, j) \models P$

<u>Inductive Step:</u>

Assume as induction hypothesis that Q is a temporal formula over $(\mathcal{V}_Q, \mathcal{A}_Q)$ such that for all α_Q over $(\mathcal{V}_Q, \mathcal{A}_Q)$ and all $j_Q \leq 0$

$$(\alpha_Q, j_Q) \models Q \qquad \text{iff} \qquad (\widehat{\alpha_Q}, j_Q) \models Q$$

Assume a similar induction hypothesis for R. We consider the different possibilities for P (cf. Section 3.5).

- $P = \bigcirc Q$

  $(\alpha, j) \models \bigcirc Q$
  iff (by definition)
  $(\alpha, j+1) \models Q$

iff (by the induction hypothesis)
$(\widehat{\alpha}, j+1) \models Q$
iff (by definition)
$(\widehat{\alpha}, j) \models \bigcirc Q$

- $P = Q \mathcal{W} R$

  Similar to case $P = \bigcirc Q$.

- $P = \forall x : Q$

  Since P is a temporal formula over $(\mathcal{V}, \mathcal{A})$, Q is a temporal formula over $(\mathcal{V} \cup \{x\}, \mathcal{A})$.

  $(\alpha, j) \models \forall x : Q$
  iff (by definition)
  for all values v, $(\alpha_v^x, j) \models Q$
  iff (by the induction hypothesis)
  for all values v, $(\widehat{\alpha_v^x}, j) \models Q$
  iff (by definition of $\widehat{}$ and α_v^x)
  for all values v, $(\widehat{\alpha}_v^x, j) \models Q$
  iff (by definition)
  $(\widehat{\alpha}, j) \models \forall x : Q$

- $P = \exists x : Q$

  Similar to case $P = \forall x : Q$.

- $P = Q \Longrightarrow R$

  Similar to case $P = \bigcirc Q$.

- $P = \neg Q$

  Similar to case $P = \bigcirc Q$.

■

Proof of Lemma 3.2:

This lemma holds for our temporal logic since we do not have any *past* operators, i.e., operators that can reference previous positions in an executions. For instance, some temporal logics (see, e.g., [MP92]) have a *previous* operator, which is dual to our *next* operator $\bigcirc$ and is defined such that *previous* P holds at position j in an execution if P holds at position $j-1$ in that execution. Since our logic lacks this possibility of referencing previous positions, the question whether P holds at position j in α only depends on the suffix $s_j a_{j+1} s_{j+1} \cdots$ of α, i.e., ${}_j|\alpha$. Similarly, the question whether P holds at position i in ${}_{j-i}|\alpha$ only depends on ${}_i|({}_{j-i}|\alpha)$, and since ${}_i|({}_{j-i}|\alpha) = {}_j|\alpha$, the result follows.

Formally, the result can be proven by structural induction over P.

■

Proof of Lemma 3.3:

Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ and $\alpha' = s'_0 a'_1 s'_1 a'_2 s'_2 \cdots$. We define inductively a nondecreasing mapping $m : \mathsf{N} \rightarrow \mathsf{N}$ such that $_k|\alpha \simeq {}_{m(k)}|\alpha'$. Furthermore, for each k we define a mapping m_k : $\{0, \ldots, m(k) - 1\} \rightarrow \{0, \ldots, k - 1\}$, such that for all $0 \leq i' < m(k)$, $_{m_k(i')}|\alpha \simeq {}_{i'}|\alpha'$. This inductive definition is clearly sufficient to prove the lemma.

<u>Base Case:</u> $k = 0$

Define $m(0) = 0$. Then, by assumption, $_0|\alpha = \alpha \simeq \alpha' = {}_{m(0)}|\alpha'$, as required.

Let m_0 be the empty mapping. Then, vacuously, for all $0 \leq i' < m(0)$, $_{m_0(i')}|\alpha \simeq {}_{i'}|\alpha'$

<u>Inductive Step:</u>

Assume as induction hypothesis that $_k|\alpha \simeq {}_{m(k)}|\alpha'$ and that, for all $0 \leq i' < m(k)$, $_{m_k(i')}|\alpha \simeq {}_{i'}|\alpha'$. We consider cases.

- $a_{k+1} = \zeta$.

  Define $m(k+1) = m(k)$. Then, clearly, $_{k+1}|\alpha \simeq {}_k|\alpha \simeq {}_{m(k)}|\alpha' = {}_{m(k+1)}|\alpha'$.

  Define $m_{k+1} = m_k$. By the induction hypothesis and the definition of $m(k+1)$ and m_{k+1}, for all $0 \leq i' < m(k+1)$, $_{m_{k+1}(i')}|\alpha \simeq {}_{i'}|\alpha'$.

- $a_{k+1} = a \neq \zeta$.

  Then, since $_k|\alpha \simeq {}_{m(k)}|\alpha'$ (induction hypothesis), there must be a unique number $k' > m(k)$ such that $s'_{m(k)} a'_{m(k)+1} s'_{m(k)+1} \cdots a'_{k'} s'_{k'} = s'_{m(k)} \zeta s'_{m(k)} \cdots a s'_{k'}$. Thus, the first non-stuttering action in α' after position $m(k)$ must be a.

  Define $m(k+1) = k'$. Then the induction hypothesis, the definition of k', and the case assumption imply $_{k+1}|\alpha \simeq {}_k|\alpha \simeq {}_{m(k)}|\alpha' = {}_{m(k+1)}|\alpha'$.

  Define m_{k+1} to coincide with m_k for all $0 \leq i' < m(k)$, and define $m_{k+1}(i') = k$, for all $m(k) \leq i' < m(k+1)$. Then the induction hypothesis and the definition of m_{k+1} give, for all $0 \leq i' < m(k)$, $_{m_{k+1}(i')}|\alpha \simeq {}_{i'}|\alpha'$. For $m(k) \leq i' < m(k+1)$ we have, $_{m_{k+1}(i')}|\alpha = {}_k|\alpha \simeq {}_{m(k)}|\alpha' \simeq {}_{i'}|\alpha'$, where the last stuttering-equivalence follows from the fact that $_{i'}|\alpha'$ only differs from $_{m(k)}|\alpha'$ by having less stuttering in the start.

This concludes the proof.

■

Proof of Proposition 3.4:

Let $\alpha_1 = s_{1,0} a_{1,1} s_{1,1} a_{1,2} s_{1,2} \cdots$ and $\alpha_2 = s_{2,0} a_{2,1} s_{2,1} a_{2,2} s_{2,2} \cdots$ be arbitrary executions such that $\alpha_1 \simeq \alpha_2$.

1. Let P be a state predicate.

   $\alpha_1 \models P$
   iff (by definition)
   $s_{1,0} \models P$

iff (since $\alpha_1 \simeq \alpha_2$ implies $s_{1,0} = s_{2,0}$)
  $s_{2,0} \models P$
iff (by definition)
  $\alpha_2 \models P$

This proves that P is stuttering-insensitive.

2. Let P be a state transition predicate, and assume that $(s, \zeta, s) \models P$ (which implies $(s, s)[\![P]\!] = \mathit{true}$) for all state s.

  $\alpha_1 \models P$
iff (by definition)
  $(s_{1,0}, s_{1,1})[\![P]\!] = \mathit{true}$
implies (since $\alpha_1 \simeq \alpha_2$ implies either $(s_{1,0}, s_{1,1}) = (s_{2,0}, s_{2,1})$ or $(s_{1,0}, s_{1,1}) = (s_{2,0}, s_{2,0})$)
  $(s_{2,0}, s_{2,1})[\![P]\!] = \mathit{true}$ or $(s_{2,0}, s_{2,0})[\![P]\!] = \mathit{true}$
iff (since $(s_{2,0}, s_{2,0})[\![P]\!] = \mathit{true}$ by assumption)
  $(s_{2,0}, s_{2,1})[\![P]\!] = \mathit{true}$
iff (by definition)
  $\alpha_2 \models P$

A symmetric argument gives the implication in the other direction. This proves that P is stuttering-insensitive.

3. Let f be an action function.

  $\alpha_1 \models \Diamond\langle f \rangle$
iff (by definition)
  there is a step $(s_{1,i}, a_{1,i+1}, s_{1,i+1})$ in α_1 such that $a_{1,i+1} \in (s_{1,i}, s_{1,i+1})[\![f]\!]$
iff (since ζ can never be in the range of an action function)
  there is a step $(s_{1,i}, a_{1,i+1}, s_{1,i+1})$ in α_1 such that $a_{1,i+1} \neq \zeta$ and $a_{1,i+1} \in (s_{1,i}, s_{1,i+1})[\![f]\!]$
implies (by definition of $\simeq$)
  there is a step $(s_{2,j}, a_{2,j+1}, s_{2,i+1}) = (s_{1,i}, a_{1,i+1}, s_{1,i+1})$ in α_2 such that $a_{2,j+1} \in (s_{2,j}, s_{2,j+1})[\![f]\!]$
iff (by definition)
  $\alpha_2 \models \Diamond\langle f \rangle$

A symmetric argument gives the implication in the other direction. This proves that $\Diamond\langle f \rangle$ is stuttering-insensitive.

4. Assume that P and Q are stuttering-insensitive temporal formulas.

 (a) $P \mathcal{W} Q$

  $\alpha_1 \models P \mathcal{W} Q$
 iff (by definition)
  there exists a $k \geq 0$ such that $(\alpha_1, k) \models Q$ and for every $0 \leq i < k$, $(\alpha_1, i) \models P$, or else, for all $i \geq 0$, $(\alpha_1, i) \models P$
 iff (by Lemma 3.1)

there exists a $k \geq 0$ such that $(\widehat{\alpha_1}, k) \models Q$ and for every $0 \leq i < k$, $(\widehat{\alpha_1}, i) \models P$,
or else, for all $i \geq 0$, $(\widehat{\alpha_1}, i) \models P$
iff (by Lemma 3.2)
there exists a $k \geq 0$ such that ${}_k|\widehat{\alpha_1} \models Q$ and for every $0 \leq i < k$, ${}_i|\widehat{\alpha_1} \models P$,
or else, for all $i \geq 0$, ${}_i|\widehat{\alpha_1} \models P$
implies (by Lemma 3.3 and the fact that P and Q are stuttering-insensitive)
there exists a $k' \geq 0$ such that ${}_{k'}|\widehat{\alpha_2} \models Q$ and for every $0 \leq i' < k'$, ${}_{i'}|\widehat{\alpha_2} \models P$,
or else, for all $i' \geq 0$, ${}_{i'}|\widehat{\alpha_2} \models P$
iff (by Lemma 3.2)
there exists a $k' \geq 0$ such that $(\widehat{\alpha_2}, k') \models Q$ and for every $0 \leq i' < k'$, $(\widehat{\alpha_2}, i') \models P$,
or else, for all $i' \geq 0$, $(\widehat{\alpha_2}, i') \models P$
iff (by Lemma 3.1)
there exists a $k' \geq 0$ such that $(\alpha_2, k') \models Q$ and for every $0 \leq i' < k'$, $(\alpha_2, i') \models P$,
or else, for all $i' \geq 0$, $(\alpha_2, i') \models P$
iff (by definition)
$\alpha_2 \models P \mathcal{W} Q$

A symmetric argument gives the implication in the other direction. This proves that $P \mathcal{W} Q$ is stuttering-insensitive.

(b) $\forall x : P$

Since $\alpha_1 \simeq \alpha_2$, we have, for all values v, $(\alpha_1)^x_v \simeq (\alpha_2)^x_v$.

$\alpha_1 \models \forall x : P$
iff (by definition)
for all values v, $(\alpha_1)^x_v \models P$
iff (since P is stuttering-insensitive and $(\alpha_1)^x_v \simeq (\alpha_2)^x_v$)
for all values v, $(\alpha_2)^x_v \models P$
iff (by definition)
$\alpha_2 \models \forall x : P$

This proves that $\forall x : P$ is stuttering-insensitive.

(c) $\exists x : P$

Similar to case $\forall x : P$.

(d) $\neg P$

$\alpha_1 \models \neg P$
iff (by definition)
$\alpha_1 \not\models P$
iff (by the fact that P is stuttering-insensitive)
$\alpha_2 \not\models P$
iff (by definition)
$\alpha_2 \models \neg P$

This proves that $\neg P$ is stuttering-insensitive.

(e) $P \Longrightarrow Q$

Similar to case $\neg P$.

■

B.2 Proofs in Chapter 4

B.2.1 Untimed Systems

Proof of Lemma 4.1:

Let $(\mathcal{V}, \mathcal{A})$ be an arbitrary pair with $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{A}' \subseteq \mathcal{A}$ and let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be an arbitrary execution over $(\mathcal{V}, \mathcal{A})$. Furthermore, let $\alpha' = \alpha \restriction (\mathcal{V}', \mathcal{A}') = s'_0 a'_1 s'_1 a'_2 s'_2 \cdots$. Then $s'_k = s_k \lceil \mathcal{V}'$ and $a'_k = \begin{cases} a_k & \text{if } a_k \in \mathcal{A}' \\ \zeta & \text{otherwise} \end{cases}$

We prove the lemma by structural induction on P.

<u>Base Case:</u>

In the base case P is a step formula over $(\mathcal{V}', \mathcal{A}')$. We consider the two kinds of step formulas:

- $P = \langle f \rangle$, where f is an action function over $(\mathcal{V}', \mathcal{A}')$.

  $(\alpha', j) \models \langle f \rangle$
  iff (by definition)
  $(0 \le j < |\alpha'|$ and $(s'_j, a'_{j+1}, s'_{j+1}) \models \langle f \rangle)$ or
  $(j \ge |\alpha'|$ and $(s'_{|\alpha'|}, \zeta, s'_{|\alpha'|}) \models \langle f \rangle)$
  iff (by definition and the fact that ζ can never be in the range of an action function)
  $(0 \le j < |\alpha'|$ and $a'_{j+1} \in (s'_j, s'_{j+1})[\![f]\!])$ or
  $(j \ge |\alpha'|$ and *false*$)$
  iff
  $(0 \le j < |\alpha'|$ and $a'_{j+1} \in (s'_j, s'_{j+1})[\![f]\!])$
  iff (step 4; see below)
  $(0 \le j < |\alpha|$ and $a_{j+1} \in (s_j, s_{j+1})[\![f]\!])$
  iff
  $(0 \le j < |\alpha|$ and $a_{j+1} \in (s_j, s_{j+1})[\![f]\!])$ or
  $(j \ge |\alpha|$ and *false*$)$
  iff (since ζ can never be in the range of an action function)
  $(0 \le j < |\alpha|$ and $a_{j+1} \in (s_j, s_{j+1})[\![f]\!])$ or
  $(j \ge |\alpha|$ and $\zeta \in (s_{|\alpha|}, s_{|\alpha|})[\![f]\!])$
  iff (by definition)
  $(0 \le j < |\alpha|$ and $(s_j, a_{j+1}, s_{j+1}) \models \langle f \rangle)$ or
  $(j \ge |\alpha|$ and $(s_{|\alpha|}, \zeta, s_{|\alpha|}) \models \langle f \rangle)$
  iff (by definition)
  $(\alpha, j) \models \langle f \rangle$

  Step 4 above is justified as follows: first, $|\alpha'| = |\alpha|$ by definition of $\restriction$. Next, since $s'_j = s_j \lceil (\mathcal{V}', \mathcal{A}')$, $s'_{j+1} = s_{j+1} \lceil (\mathcal{V}', \mathcal{A}')$, and f is an action function over $(\mathcal{V}', \mathcal{A}')$, we have that $(s'_j, s'_{j+1})[\![f]\!] = (s_j, s_{j+1})[\![f]\!]$. Finally, if $a'_{j+1} = \zeta$, then $a_{j+1} \notin \mathcal{A}'$ by definition of $\restriction$, and since f is an action function over $(\mathcal{V}', \mathcal{A}')$, we have $a'_{j+1} \in (s'_j, s'_{j+1})[\![f]\!])$ iff $a_{j+1} \in (s_j, s_{j+1})[\![f]\!])$. If $a'_{j+1} \ne \zeta$, then $a'_{j+1} = a_{j+1}$. That suffices.

- P is a state transition predicate over $(\mathcal{V}', \mathcal{A}')$.

$(\alpha', j) \models P$
iff (by definition)
$(0 \leq j < |\alpha'|$ and $(s'_j, a'_{j+1}, s'_{j+1}) \models P)$ or
$(j \geq |\alpha'|$ and $(s'_{|\alpha'|}, \zeta, s'_{|\alpha'|}) \models P)$
iff (by definition)
$(0 \leq j < |\alpha'|$ and $(s'_j, s'_{j+1})[\![P]\!] = \mathit{true})$ or
$(j \geq |\alpha'|$ and $(s'_{|\alpha'|}, s'_{|\alpha'|})[\![P]\!] = \mathit{true})$
iff (step 3; see below)
$(0 \leq j < |\alpha|$ and $(s_j, s_{j+1})[\![P]\!] = \mathit{true})$ or
$(j \geq |\alpha|$ and $(s_{|\alpha|}, s_{|\alpha|})[\![P]\!] = \mathit{true})$
iff (by definition)
$(0 \leq j < |\alpha|$ and $(s_j, a_{j+1}, s_{j+1}) \models P)$ or
$(j \geq |\alpha|$ and $(s_{|\alpha|}, \zeta, s_{|\alpha|}) \models P)$
iff (by definition)
$(\alpha, j) \models P$

Step 3 is justified as follows: first, $|\alpha'| = |\alpha|$, by definition of $\upharpoonright$. Then, since P is a state transition predicate over $(\mathcal{V}', \mathcal{A}')$ and $s'_k = s_k\lceil(\mathcal{V}', \mathcal{A}')$ for all k, the result directly follows.

Inductive Step:

Let Q be an arbitrary temporal formula over $(\mathcal{V}'_Q, \mathcal{A}'_Q)$ and assume as induction hypothesis that for all pairs $(\mathcal{V}_Q, \mathcal{A}_Q)$ with $\mathcal{V}'_Q \subseteq \mathcal{V}_Q$ and $\mathcal{A}'_Q \subseteq \mathcal{A}_Q$, all executions α_Q over $(\mathcal{V}_Q, \mathcal{A}_Q)$, and all $j_Q \geq 0$,

$$(\alpha_Q \upharpoonright (\mathcal{V}'_Q, \mathcal{A}'_Q), j_Q) \models Q \qquad \text{iff} \qquad (\alpha_Q, j_Q) \models Q$$

Assume a similar induction hypothesis for the temporal formula R over $(\mathcal{V}'_R, \mathcal{A}'_R)$. We consider the different possibilities for P (cf. Section 3.5).

- $P = \bigcirc Q$

  $(\alpha \upharpoonright (\mathcal{V}', \mathcal{A}'), j) \models \bigcirc Q$
  iff (by definition)
  $(\alpha \upharpoonright (\mathcal{V}', \mathcal{A}'), j+1) \models Q$
  iff (by the induction hypothesis)
  $(\alpha, j+1) \models Q$
  iff (by definition)
  $(\alpha, j) \models \bigcirc Q$

- $P = Q \; \mathcal{W} \; R$

  Similar to case $P = \bigcirc Q$.

- $P = \forall x : Q$

  $(\alpha \upharpoonright (\mathcal{V}', \mathcal{A}'), j) \models \forall x : Q$
  iff (by definition)
  for all values v, $((\alpha \upharpoonright (\mathcal{V}', \mathcal{A}'))^x_v, j) \models Q$

iff (by definition of $\restriction$ and substitution)
for all values v, $(\alpha_v^x \restriction (\mathcal{V}' \cup \{x\}, \mathcal{A}'), j) \models Q$
iff (by the induction hypothesis)
for all values v, $(\alpha_v^x, j) \models Q$
iff (by definition)
$(\alpha, j) \models \forall x : Q$

- $P = \exists x : Q$

  Similar to case $P = \forall x : Q$.

- $P = (Q \Longrightarrow R)$

  Similar to case $P = \bigcirc Q$.

- $P = \neg Q$

  Similar to case $P = \bigcirc Q$.

■

Proof of Lemma 4.3:

$\Longrightarrow$: Assume $\alpha\lceil A_i \models Q_i$ for all i. Then since $\alpha\lceil A_i \simeq \alpha \restriction A_i$ and Q_i is stuttering-insensitive, we have $\alpha \restriction A_i \models Q_i$, for all i. Then by Lemma 4.2, $\alpha \models Q_i$, for all i, and thus $\alpha \models Q_1 \wedge \ldots \wedge Q_N$.

$\Longleftarrow$: Assume $\alpha \models Q_1 \wedge \ldots \wedge Q_N$. Then $\alpha \models Q_i$, for all i, and Lemma 4.2 implies that $\alpha \restriction A_i \models Q_i$, for all i. Again, since $\alpha\lceil A_i \simeq \alpha \restriction A_i$ and Q_i is stuttering-insensitive, it follows that $\alpha\lceil A_i \models Q_i$, for all i.

■

Proof of Proposition 4.4:

By Definition 2.9 we have $L = \{\alpha \in exec(A) \mid \alpha\lceil A_1 \in L_1, \ldots, \alpha\lceil A_N \in L_N\}$. By definition of $\lceil$ we know that if $\alpha \in exec(A)$, then $\alpha\lceil A_i \in exec(A_i)$, for all i. Thus, since L_i is induced by Q_i, we get $L = \{\alpha \in exec(A) \mid \alpha\lceil A_1 \models Q_1, \ldots, \alpha\lceil A_N \models Q_N\}$. By Lemma 4.3 we finally get $L = \{\alpha \in exec(A) \mid \alpha \models Q_1 \wedge \ldots \wedge Q_N\}$ which proves that L is induced by $Q_1 \wedge \ldots \wedge Q_N$.

■

Proof of Proposition 4.5:

Let $(A_{\mathcal{A}}, L_{\mathcal{A}}) = (A, L) \setminus \mathcal{A}$. The proof is trivial since, by Definitions 2.3 and 2.10, $exec(A_{\mathcal{A}}) = exec(A)$ and $L_{\mathcal{A}} = L$.

■

Proof of Proposition 4.6:

Let $(A_\rho, L_\rho) = \rho((A, L))$. By Definition 2.11 we have $(A_\rho, L_\rho) = (\rho(A), \{\rho(\alpha) \mid \alpha \in L\})$.

First note that since Q is a temporal formula over A, Definition 2.4 implies that $\rho(Q)$ is a temporal formula over A_ρ.

Now, it is clear that $\alpha \models Q$ iff $\rho(\alpha) \models \rho(Q)$. Since also $exec(A_\rho) = \{\rho(\alpha) \mid \alpha \in exec(A)\}$, it follows that $L_\rho = \{\alpha \in exec(A_\rho) \mid \alpha \models \rho(Q)\}$, which proves that L_ρ is induced by $\rho(Q)$.

■

B.2.2 Timed Systems

Proof of Proposition 4.17:

Let $L_{i,s}$, for each $1 \le i \le N$, be a sampling characterization of L_i such that $L_{i,s}$ is induced by Q_i. We have

$$\begin{aligned} L &\stackrel{1}{=} \{\Sigma \in t\text{-}exec^\infty(A) \mid \Sigma\lceil A_1 \in L_1, \ldots, \Sigma\lceil A_N \in L_N\} \\ &\stackrel{2}{=} \{\Sigma \in t\text{-}exec^\infty(A) \mid (\forall \alpha_1 \text{ samples } \Sigma\lceil A_1 : \alpha_1 \in L_{1,s}), \ldots, \\ &\qquad\qquad (\forall \alpha_N \text{ samples } \Sigma\lceil A_N : \alpha_1 \in L_{N,s})\} \\ &\stackrel{3}{=} \{\Sigma \in t\text{-}exec^\infty(A) \mid \forall \alpha \text{ samples } \Sigma : \alpha\lceil A_1 \in L_{1,s}, \ldots, \alpha\lceil A_N \in L_{N,s}\} \end{aligned}$$

where Step 1 follows from Definition 2.26, Step 2 follows from the definition of sampling characterizations, and Step 3 follows from Lemma 4.15 Part 3.

This proves (using Lemma 4.13 Part 2) that L is induced by $L_s = \{\alpha \in exec^\infty(A) \mid \alpha\lceil A_1 \in L_{1,s}, \ldots, \alpha\lceil A_N \in L_{N,s}\}$, and we have

$$\begin{aligned} L_s &\stackrel{1}{=} \{\alpha \in exec^\infty(A) \mid \alpha\lceil A_1 \models Q_1, \ldots, \alpha\lceil A_N \models Q_N\} \\ &\stackrel{2}{=} \{\alpha \in exec^\infty(A) \mid \alpha \models Q_1 \wedge \cdots \wedge Q_N\} \end{aligned}$$

where Step 1 follows from the definition of sampling characterization being induced by temporal formulas and Step 2 follows from Lemma 4.16.

This proves that L_s and, in turn, L are induced by $Q_1 \wedge \ldots \wedge Q_N$.

■

Proof of Proposition 4.18:

Let $(A_{\mathcal{A}}, L_{\mathcal{A}}) = (A, L) \setminus \mathcal{A}$. The proof is trivial since, by Definitions 2.19 and 2.27, $exec(A_{\mathcal{A}}) = exec(A)$, $t\text{-}exec(A_{\mathcal{A}}) = t\text{-}exec(A)$, and $L = L_{\mathcal{A}}$.

■

Proof of Proposition 4.19:

Let $(A_\rho, L_\rho) = \rho((A, L))$ and let L_s be a sampling characterization of L such that L_s is induced by Q. By Definition 2.28 we have $(A_\rho, L_\rho) = (\rho(A), \{\rho(\Sigma) \mid \Sigma \in L\})$.

First note that since Q is a temporal formula over A, Definition 2.20 implies that $\rho(Q)$ is a temporal formula over A_ρ.

Now, it is clear that $exec(A_\rho) = \{\rho\alpha \mid \alpha \in exec(A)\}$ and that $\alpha \models Q$ iff $\rho(\alpha) \models \rho(Q)$. Thus, $L_{\rho,s} = \{\rho(\alpha) \mid \alpha \in L_s\}$ is induced by $\rho(Q)$. Since also $t\text{-}exec(A_\rho) = \{\rho\Sigma \mid \Sigma \in t\text{-}exec(A)\}$ and α samples Σ iff $\rho(\alpha)$ samples $\rho(\Sigma)$, we immediately get that L_ρ is induced by $L_{\rho,s}$. That suffices.

■

B.2.3 Embedding

Proof of Lemma 4.21:

Since Q is a temporal formula over A, α is an execution over A_p, $variables(A) \subseteq variables(A_p)$, and $acts(A) \subseteq acts(A_p)$, Lemma 4.1 yields

$$(\alpha \upharpoonright (variables(A), acts(A))) \models Q \quad \text{iff} \quad \alpha \models Q \qquad (*)$$

Furthermore, by definition of $untime(\alpha)$ we have $untime(\alpha) \simeq (\alpha \upharpoonright (variables(A), acts(A)))$, and since Q is stuttering-insensitive we have

$$untime(\alpha) \models Q \qquad \text{iff} \qquad (\alpha \upharpoonright (variables(A), acts(A))) \models Q \tag{$**$}$$

Then $(*)$ and $(**)$ imply the result.

■

Proof of Proposition 4.22:

First note that since $variables(A) \subseteq variables(A_p)$ and $acts(A) \subseteq acts(A_p)$, Q is a temporal formula over A_p. We have

$$\begin{aligned} L_p &\stackrel{1}{=} \{\Sigma \in t\text{-}exec^\infty(A_p) \mid untime(\Sigma) \in L\} \\ &\stackrel{2}{=} \{\Sigma \in t\text{-}exec^\infty(A_p) \mid untime(\Sigma) \models Q\} \\ &\stackrel{3}{=} \{\Sigma \in t\text{-}exec^\infty(A_p) \mid \text{for all } \alpha, \text{ if } \alpha \text{ samples } \Sigma, \text{ then } untime(\alpha) \models Q\} \\ &\stackrel{4}{=} \{\Sigma \in t\text{-}exec^\infty(A_p) \mid \text{for all } \alpha, \text{ if } \alpha \text{ samples } \Sigma, \text{ then } \alpha \models Q\} \end{aligned}$$

where Step 1 follows from Definition 2.35, Step 2 follows from the fact that L is induced by Q (and $untime(\Sigma) \in exec(A)$ by definition of *untime*), Step 3 follows Lemma 4.20, and Step 4 follows from Lemma 4.21.

This proves, by Lemma 4.13 Part 2, that L_p is induced by Q.

We show that Q is minimal. Thus, for arbitrary admissible execution α of A_p with $\alpha \models Q$, we must show the existence of a timed execution $\Sigma \in L_p$ such that α samples Σ.

Let α be an arbitrary admissible execution α of A_p such that $\alpha \models Q$. Let Σ be a timed execution of A_p such that α samples Σ. By Lemmas 4.11 and 4.13 Σ exists and is admissible. By Lemma 4.20 $untime(\alpha) = untime(\Sigma)$ and Lemma 4.21 gives $untime(\alpha) \models Q$. Thus, $untime(\Sigma) \models Q$, which implies $untime(\Sigma) \in L$. Then, by definition of L_p (Definition 2.35), $\Sigma \in L_p$. That suffices.

■

B.3 Proofs in Chapter 5

B.3.1 Untimed Systems

Proof of Lemma 5.10:

Let m be an arbitrary index mapping from α to α' with respect to R.

$\Longrightarrow$: Assume $\alpha \models \Diamond\Box\neg\langle C\rangle$. Then, by Lemma 3.5 Part 3, there exists an index i such that ${}_i|\alpha \models \Box\neg\langle C\rangle$. Thus, no actions in C occur in $trace({}_i|\alpha)$. By Lemma 5.6 and the fact that C contains external actions only, no actions in C occur in the suffix ${}_{m(i)}|\alpha'$. Thus, ${}_{m(i)}|\alpha' \models \Box\neg\langle C\rangle$, which, by Lemma 3.5 Part 4, implies that $\alpha' \models \Diamond\Box\neg\langle C\rangle$. That suffices.

$\Longleftarrow$: Assume $\alpha' \models \Diamond\Box\neg\langle C\rangle$. Then, by Lemma 3.5 Part 3, there exists an index j such that ${}_j|\alpha' \models \Box\neg\langle C\rangle$. Now, by Condition 4 of Definition 5.4, there exists an $i \leq |\alpha|$ such that $m(i) \geq j$. Then ${}_{m(i)}|\alpha'$ is a suffix of ${}_j|\alpha'$, and consequently, by Lemma 3.5 Part 1, ${}_{m(i)}|\alpha' \models \Box\neg\langle C\rangle$.

Thus, no actions in C occur in $trace({}_{m(i)}|\alpha')$. By Lemma 5.6 and the fact that C contains external actions only, no actions in C occur in the suffix ${}_i|\alpha$. Thus, ${}_i|\alpha \models \Box\neg\langle C\rangle$, which, by Lemma 3.5 Part 4, implies that $\alpha \models \Diamond\Box\neg\langle C\rangle$. That suffices.

■

Proof of Lemma 5.11:

Let m be an arbitrary index mapping from α to α' with respect to R.

Assume $\alpha' \models \Diamond\Box Q$. Then, by Lemma 3.5 Part 3, there exists an index j such that ${}_j|\alpha' \models \Box Q$. Thus, for each state u in ${}_j|\alpha'$, we have $u \models Q$. Now, by Condition 4 of Definition 5.4 and the fact that m is nondecreasing, we get the existence of an index i such that for all $i \leq k \leq |\alpha|$, $m(k) \geq j$. Then, for each state s of α with index k ($i \leq k \leq |\alpha|$) we have $s \models P$ since (by Condition 2 of Definition 5.4) there exists some u in ${}_j|\alpha'$ such that $(s, u) \in R$.

This gives us, for all $k > 0$, $({}_i|\alpha, k) \models P$. (Even if ${}_i|\alpha$ is finite this is true since P holds in the stuttering step that stutters the last state since it holds in the last state.). Thus, ${}_i|\alpha \models \Box P$, which finally, by Lemma 3.5 Part 4, $\alpha \models \Diamond\Box P$.

■

Proof of Lemma 5.13:

1. Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$. Let $s_{h0} \in \mathit{start}(A_h)$ be such that $s_{h0} \upharpoonright \mathit{variables}(A) = s_0$. Define $\alpha_{h0} = s_{h0}$. Then $\alpha_{h0} \upharpoonright (\mathit{variables}(A), \mathit{acts}(A)) = s_0$.

   Define α_{hn} inductively as follows. Assume $\alpha_{h(n-1)} = s_{h0} a_1 s_{h1} a_2 s_{h2} \ldots a_{n-1} s_{h(n-1)}$ is an execution of A_h such that $\alpha_{h(n-1)} \upharpoonright (\mathit{variables}(A), \mathit{acts}(A)) = \alpha|_{n-1}$. Then, by Lemma 5.12 Part 1, there exists a step $(s_{h(n-1)}, a_n, s_{hn}) \in \mathit{steps}(A_h)$.
   Define $\alpha_{hn} = s_{h0} a_1 s_{h1} a_2 s_{h2} \ldots a_{n-1} s_{h(n-1)} a_n s_{hn}$. Then $\alpha_{hn} \upharpoonright (\mathit{variables}(A), \mathit{acts}(A)) = \alpha|_n$.

   Then, $\alpha_h = \lim_{n \to |\alpha|} \alpha_{hn}$ has the required property.

2. Directly from Lemma 5.12 Part 2.

■

Proof of Lemma 5.14:

$A \sqsubseteq_{\mathrm{S}} A_h$: Let $\beta \in \mathit{traces}(A)$ and let $\alpha \in \mathit{exec}(A)$ be such that $\mathit{trace}(\alpha) = \beta$. By Lemma 5.13 Part 1 there exists an execution $\alpha_h \in \mathit{exec}(A_h)$ such that $\alpha_h \upharpoonright (\mathit{variables}(A), \mathit{acts}(A)) = \alpha$. Then, since $\mathit{ext}(A) = \mathit{ext}(A_h)$, we have $\mathit{trace}(\alpha_h) = \mathit{trace}(\alpha) = \beta$. Thus, $\beta \in \mathit{traces}(A_h)$. That suffices.

$A_h \sqsubseteq_{\mathrm{S}} A$: Let $\beta \in \mathit{traces}(A_h)$ and let $\alpha_h \in \mathit{exec}(A_h)$ be such that $\mathit{trace}(\alpha_h) = \beta$. By Lemma 5.13 Part 2, $\alpha_h \upharpoonright A \in \mathit{exec}(A)$. Then, since $\mathit{ext}(A) = \mathit{ext}(A_h)$, we have $\mathit{trace}(\alpha_h) = \mathit{trace}(\alpha_h \upharpoonright A) = \beta$. Thus, $\beta \in \mathit{traces}(A)$. That suffices.

■

Proof of Lemma 5.15:

$(A, L) \sqsubseteq_{\mathrm{L}} (A_h, L_h)$: Let $\beta \in \mathit{traces}(L)$ and let $\alpha \in L$ be such that $\mathit{trace}(\alpha) = \beta$. By Lemma 5.13 Part 1 there exists an execution $\alpha_h \in \mathit{exec}(A_h)$ such that $\alpha_h \upharpoonright A = \alpha$. Thus, by definition of L_h we have $\alpha_h \in L_h$, and since $\mathit{ext}(A) = \mathit{ext}(A_h)$ we finally get $\mathit{trace}(\alpha_h) = \mathit{trace}(\alpha) = \beta$, and thus, $\beta \in \mathit{traces}(L_h)$. That suffices.

$(A_h, L_h) \sqsubseteq_{\mathrm{L}} (A, L)$: Let $\beta \in \mathit{traces}(L_h)$ and let $\alpha_h \in L_h$ be such that $\mathit{trace}(\alpha_h) = \beta$. By definition of L_h, $\alpha_h \upharpoonright A \in L$. Then, since $\mathit{ext}(A) = \mathit{ext}(A_h)$, we have $\mathit{trace}(\alpha_h) = \mathit{trace}(\alpha_h \upharpoonright A) = \beta$. Thus, $\beta \in L$. That suffices.

■

Proof of Lemma 5.16:

We have

$$\begin{aligned} L_h &= \{\alpha_h \in exec(A_h) \mid \alpha_h \upharpoonright A \models Q\} \\ &= \{\alpha_h \in exec(A_h) \mid \alpha_h \models Q\} \end{aligned}$$

where the first equality follows from the definition of L_h and Lemma 5.13 Part 2, and the last equality follows from Lemma 4.1. This shows that L_h is induced by Q.

■

B.3.2 Timed Systems

Proof of Lemma 5.28:

1. Let $\Sigma = \omega_0 a_1 \omega_1 a_2 \omega_2 \cdots$. Define h_0 to be a value of h such that $(\mathit{fstate}(\omega) \cup [h \mapsto h_0]) \in \mathit{start}(A_h)$. Define, for all $t \in \mathit{dom}(\omega_0)$, $\omega_{h0}(t) \upharpoonright \mathit{variables}(A) = \omega_0(t)$ and $\omega_{h0}(t).h = h_0$. Then ω_{h0} is a trajectory of A_h.

   Now we define ω_{hn} inductively. By the properties of timed executions, $(\omega_{n-1}, a_n, \omega_n) \in \mathit{steps}(A)$. Then by Lemma 5.27 Part 1 where exists a value h_n such that $(\omega_{n-1} \cup [h \mapsto h_{n-1}], a_n, \omega_n \cup [h \mapsto h_n]) \in \mathit{steps}(A_h)$. Then, for all $t \in \mathit{dom}(\omega_n)$, define $\omega_{hn}(t) \upharpoonright \mathit{variables}(A) = \omega_0(t)$ and $\omega_{hn}(t).h = h_n$.

   Then, $\Sigma_h = \omega_{h0} a_1 \omega_{h1} a_2 \omega_{h2} \cdots$ is a timed execution of A_p and $\Sigma_h \upharpoonright \mathit{variables}(A) = \Sigma$.

2. Directly from Lemma 5.27 Part 2.

■

Proof of Lemma 5.32:

Let L_s be a sampling characterization of L such that L_s is induced by Q and define

$$L_{h,s} \triangleq \{\alpha_h \in exec^{\infty}(A_h) \mid \alpha_h \upharpoonright A \in L_s\}$$

Similar to the proof of Lemma 5.16 it is easy to see that $L_{h,s}$ is induced by Q. It now suffices to show that L_h is induced by $L_{h,s}$. We must check two conditions.

1. Assume $\Sigma_h \in L_h$. We must show that for all α_h that samples Σ_h, $\alpha_h \in L_{h,s}$. So, assume α_h samples Σ_h. Since Σ_h is admissible, also α_h is admissible by Lemma 4.13. Thus, by definition of $L_{h,s}$ it suffices to show that $\alpha_h \upharpoonright A \in L_s$.

   Since $\Sigma_h \in L_h$, we have $\Sigma_h \upharpoonright A \in L$. Lemma 5.31 Part 1 gives $\alpha_h \upharpoonright A$ samples $\Sigma_h \upharpoonright A$. Then $\alpha_h \upharpoonright A \in L_s$ since L_s is a sampling characterization of L. That suffices.

2. Assume $\Sigma_h \in t\text{-}exec^{\infty}(A_h)$ and for all α_h samples Σ_h, $\alpha_h \in L_{h,s}$. We must show that $\Sigma_h \in L_h$. By definition of L_h it suffices to show that $\Sigma_h \upharpoonright A \in L$.

   Let α be an arbitrary execution of A such that α samples $\Sigma_h \upharpoonright A$. Then Lemma 5.31 Part 2 gives the existence of an execution α_h of A_h such that $\alpha = \alpha_h \upharpoonright A$ and α_h samples Σ_h. Thus, the assumption for this case implies $\alpha_h \in L_{h,s}$. By Lemma 4.13 α_h is admissible. Then the definition of $L_{h,s}$ implies that $\alpha \in L_s$. Since α was arbitrary, the definition of sampling characterization implies that $\Sigma_h \upharpoonright A \in L$. That suffices.

That concludes the proof.

■

Appendix C

Invariance Proofs

In this chapter we prove the invariants stated in the G and C specifications. We use the normal proof technique:

- Show that the invariant is satisfied in every initial state.
- Assume the invariant and all previously proved invariants hold in a state s, and for all steps (s, a, s') show that the invariant holds in s'.

Many of the invariants consist of several parts. We prove that the conjunction of these parts is an invariant. It follows that each conjunct (part) is then itself an invariant. All the parts of the invariants are of the form

If C then P

where, in some cases, $C = true$. For the sake of brevity we consider only, in the second part of the proof technique above, the steps that can change C from *false* to *true* or make P *false* while C is *true* since these are the only steps that might invalidate the invariant. We refer to such steps as the *critical* steps for the invariant (part).

C.1 Proof of Invariants at the G Level

Proof of Invariant 8.1

- Since $mode_s =$ `idle` in the initial states of G, it follows that both parts of the invariant are satisfied in the initial states.
- We now consider the two parts separately
  1. We consider the critical steps. (Note that none of the steps in G can remove elements from $used_s$)

     $\underline{a = choose\_id(id, m)}$

     This step changes $mode_s$ to `send` but at the same time the new value of $last_s$ is appended to the end of $used_s$, so Part 1 holds after the step.

     $\underline{a \in \{receive\_pkt_{rs}(id, b), recover_s\}}$

     Both of these steps can change $last_s$ but at the same time $mode_s$ is changed to non-`send`, so Part 1 holds after the steps.

2. The proof of this part follows directly from the proof of Part 1 and the fact the $used_s$ is a queue of IDs. (Remember that nil $\notin ID$).

■

Proof of Invariant 8.2

- Since $mode_s$ = idle and $used_s = \varepsilon$ in initial states of G, both parts of the invariant hold in the initial states.
- We assume that both parts hold in state s. For each part we consider the critical steps of the form (s, a, s').

1. $\underline{a = prepare}$

   This step changes $mode_s$ to needid but at the same time $good_s$ is changed to $\emptyset$, so Part 1 holds in s'.

   $\underline{a = choose\_id(id, m)}$

   This step adds an id to $used_s$ but at the same time $mode_s$ is changed to send, so Part 1 holds in s'.

   $\underline{a = grow\_good_s(ids)}$

   We consider this case when $s.mode_s$ = needid. The step adds identifiers to $used_s$ but since $s.mode_s$ = needid the step can only add ids that do not intersect with $s.used_s$. Thus, since Part 1 is assumed to hold in s, it also holds in s'.

2. $\underline{a = choose\_id(id, m)}$

   This step adds the element id from $s.good_s$ to $used_s$ but since $s.mode_s$ = needid, the assumption that Part 1 holds in s gives us that $id \notin s.used_s$. Thus Part 2 holds in s'.

■

Proof of Invariant 8.3

- Initially $mode_r$ = idle so the invariant holds.
- Assume that the invariant holds in s. We now consider all the critical steps of the form (s, a, s').

1. $\underline{a = receive\_pkt_{sr}(m, id)}$

   If this step changes $mode_r$ to rcvd, it also adds an element to buf_r, so Part 1 holds in s'.

   $\underline{a = receive\_msg(m)}$

   This step can make buf_r empty, but in this case, $mode_r$ is changed to ack, so Part 1 holds in s'.

■

Proof of Invariant 8.4

- Part 1 holds initially because $mode_s$ = idle. $issued_r$ is initially a superset of $good_r$ thus satisfying Part 2. For Parts 3, 4, 5, and 6 the sets that are supposed to be subsets are initially empty, so the result follows. Since $last_r$ is initially nil, Parts 7 and 8 are also satisfied.

- For each part of the invariant we consider the critical steps (s, a, s'), where we assume that all parts of this invariant hold in s, and that previously proved invariants hold in both s and s'. (For Parts 1, 2, and 3, note that $issued_r$ can never shrink, and for Parts 4, 5, 6, and 8, note that $used_s$ can never shrink.)

1. $\underline{a = prepare}$

   This step changes $mode_s$ to `needid`, but at the same time $good_s$ is made empty, so Part 1 holds in s'.

   $\underline{a = recover_r}$

   This step changes $mode_r$ from `rec` to non`rec` (`idle`) but at the same time $issued_r$ is changed to some superset of $good_s$, so Part 1 holds in s'.

   $\underline{a = grow\_good_s(ids)}$

   We consider the case where $s.mode_s =$ `needid` and $s.mode_r \neq$ `rec`. The step adds some elements to $good_s$, but in the case we consider, the elements that are added are all in $s.issued_r$. So, since we assume Part 1 holds in s, it also holds in s'.

2. $\underline{a = grow\_good_r(ids)}$

   This step adds elements to $good_r$ but at the same time the same elements are added to $issued_r$. So, since we assume that Part 2 holds in s, it also holds in s'.

3. $\underline{a = recover_r}$

   This step changes $mode_r$ from `rec` to non-`rec`, but at the same time $issued_r$ is changed to some superset of $used_s$, so Part 3 holds in s'.

   $\underline{a = prepare}$

   Consider this step when $s.mode_r \neq$ `rec`. We add an element id from $s.good_s$ to $used_s$. From Part 1 we get that id belongs to $s.issued_r$ so adding id to $used_s$ does not violate Part 3.

4. In the proof, we let $id\text{-}set$ denote the set $ids(sr) \cup (\text{if } mode_s =$ `send` then $\{last_s\})$ in any state of G.

   $\underline{a = choose\_id(id, m)}$

   This step changes $mode_s$ to `send` so $s'.last_s$ gets added to $id\text{-}set$, but from Invariant 8.1 Part 1 we get that $s'.last_s \in s'.used_s$, so Part 4 is not violated.

   $\underline{a = send\_pkt_{sr}(m, id)}$

   This step might add a packet to the channel (sr), but since a precondition for the step is $s.mode_s =$ `send`, the id on the packet is already in $id\text{-}set$, thus this step does not change $id\text{-}set$. So, since Part 4 holds in s, it also holds in s'.

5. $\underline{a = receive\_pkt_{sr}(m, id)}$

   This is the only step that may add an identifier to $nack\text{-}buf_s$. The identifier id added is in $ids(s.sr)$, so since we assume that Part 4 holds in state s we get that $id \in s.used_s$, so Part 5 is not violated.

6. $\underline{a = \mathit{send\_pkt}_{rs}(id, \mathit{true})}$

   This step can add a packet with identifier $s.\mathit{last}$ to the return channel rs. The action is only possible if $s.\mathit{last}_r \in \mathit{ID}$, i.e., if $s.\mathit{last}_r \neq \texttt{nil}$. But then Part 8 gives us that $s.\mathit{last}_r \in s.\mathit{used}_s$, thus this step cannot violate Part 6.

   $\underline{a = \mathit{send\_pkt}_{rs}(id, \mathit{false})}$

   This step can add a packet with an identifier from $s.\mathit{nack\text{-}buf}$ to rs. From Part 5 in state s we get that this identifier is in $s.\mathit{used}_s$, so the step cannot violate Part 6.

7. $\underline{a = \mathit{receive\_pkt}_{sr}(m, id)}$

   This step can change last_r to id which belongs to $s.\mathit{good}_r$. However, at the same time id is removed from good_r. It remains to be shown that $id \notin \overline{s'.\mathit{issued}_r}$. Since we assume that all parts of this invariant hold in s, Part 2 gives us that $id \in s.\mathit{issued}_r$ and since issued_r is not changed in the step, we get $id \in s'.\mathit{issued}_r$. The result follows directly.

   $\underline{a = \mathit{recover}_r}$

   This step changes last_r to $\texttt{nil}$. But since $\mathit{good\text{-}ids}$ is a set of elements from ID and $\texttt{nil} \notin \mathit{ID}$, Part 7 holds in state s'.

   $\underline{a = \mathit{grow\_good}_r(\mathit{ids})}$

   This step does not change $\mathit{good\text{-}ids}$, so Part 7 holds in state s'.

8. $\underline{a = \mathit{receive\_pkt}_{sr}(m, id)}$

   This is the only step that can change last_r to non-$\texttt{nil}$. last_r is changed to an identifier id in a packet in $s'.sr$. From Part 4 in state s we get that $id \in s.\mathit{used}_s$, so since used_s does not change in the step, Part 8 holds in state s'.

■

Proof of Invariant 8.5

- Initially $sr = \emptyset$ and $\mathit{mode}_s \neq \texttt{send}$, so the invariant holds.
- We consider the critical steps (s, a, s'), where we assume that this invariant hold in s, and that previously proved invariants hold in both s and s'. Note that no step can change $\mathit{current\text{-}msg}_s$ and end up in a state with $\mathit{mode}_s = \texttt{send}$. Also, no step, except $\mathit{choose\_id}(id, m)$ can change last_s and end up in a state with $\mathit{mode}_s = \texttt{send}$.

  1. $\underline{a = \mathit{choose\_id}(id, m)}$

     This step changes mode_s from $\texttt{needid}$ to $\texttt{send}$. From Invariant 8.4 Part 4 we get that $s.\mathit{used}_s \supseteq \mathit{ids}(s.sr)$. From Invariant 8.2 Part 1 and the definition of $\mathit{choose\_id}(id, m)$ we then get that $s'.\mathit{last}_s \notin \mathit{ids}(s'.sr)$, so this step does not invalidate the invariant.

■

Proof of Invariant 8.6

- Initially $\mathit{current\text{-}ok} = \mathit{false}$, so all parts of the invariant hold.

- For each part of the invariant we consider the critical steps (s, a, s'), where we assume that all parts of this invariant hold in s, and that previously proved invariants hold in both s and s'. Note, for the Parts 3, 4, 6, and 7, that no step, except $choose\_id(id, m)$, can change $last_r$ without also changing $mode_s$ to something other that `send`.

1. $\underline{a = prepare}$

   This changes *current-ok* to *true* if $s.mode_r \neq$ `rec`, but at the same time $mode_s$ is changed to `needid`, so Part 1 holds in state s'.

   $\underline{a = receive\_pkt_{rs}(id, b)}$

   In order for this step to change $mode_s$ to `idle`, we must have $s.mode_s =$ `send` and $(s.last_s, b) \in s.rs$. In that case the step can only violate Part 1 if $s.current\text{-}ok = true$, but this cannot be the case since we assume that Part 4 holds in state s. Thus, the step cannot violate Part 1.

   $\underline{a = crash_s}$

   This step can change $mode_s$ from `needid` or `send` to `rec`, but at the same time *current-ok* is set to *false*, so Part 4 holds in state s

2. $\underline{a = prepare}$

   This step changes *current-ok* to *true*, but only if $mode_r \neq$ `rec`, so Part 2 holds in s.

   $\underline{a = crash_r}$

   This is the only step that can change $mode_r$ from non-`rec` to `rec` but at the same time *current-ok* is made *false*, so Part 2 holds in s.

3. $\underline{a = choose\_id(id, m)}$

   This is the only step that can change the condition in Part 2 from *false* to *true*. This happens if $s.current\text{-}ok = true$. Since $s.mode_s =$ `needid`, Part 5 which we assume holds in s gives us that $s'.last_s \in s.good_r$ which again implies that $s'.last_s \in s'.good_r$. From Invariant 8.4 Part 7 we get that $s'.last_r \notin s'.good_r$. Thus $s'.last_s \neq s'.last_r$, so Part 3 holds in s'.

   $\underline{a = receive\_pkt_{sr}(m, id)}$

   This step can make $s'.last_s = s'.last_r$ but in this case *curremt-ok* is changed to *false*, so Part 3 holds in s'.

   $\underline{a = recover_r}$

   Consider this step when $mode_s =$ `send` and $current\text{-}ok = true$. The step changes $last_r$ to `nil` but from Invariant 8.1 Part 2 we have $s'.mode_s \neq$ `nil`, so Part 3 holds in s'.

4. $\underline{a = choose\_id(id, m)}$

   This is the only step that can change the condition in Part 2 from *false* to *true*. This happens if $s.current\text{-}ok = true$, so assume this. In state s we get from Invariant 8.4 Part 6 that all *ids* on $s.rs$ are in $s.used_s$. From Invariant 8.2 Part 1 we get that $s'.last_s \notin s.used_s$. Since *rs* is not changed in the step, we finally conclude that $(s'.last_s, b) \notin s'.rs$, so Part 4 holds in state s'.

$\underline{a = send\_pkt_{rs}(id, true)}$

Consider this action when $mode_s = \mathtt{send}$ and $current\text{-}ok = true$. $(s.last_r, true)$ might be added to rs, but from Part 3 we get that Part 4 is not violated.

$\underline{a = send\_pkt_{rs}(id, false)}$

Consider this action when $mode_s = \mathtt{send}$ and $current\text{-}ok = true$. A packet with an id from $s.nack\text{-}buf_r$ might be added to rs, but from Part 7 (which we assume holds in s) we get that Part 4 is not violated.

5. $\underline{a = prepare}$

   This step can make $current\text{-}ok = true$ and $mode_s = \mathtt{needid}$ but at the same time $good_s$ is made empty, so Part 5 holds in state s'.

   $\underline{a = grow\_good_s(ids)}$

   This step can only add elements from $good_r$ to $good_s$ when $current\text{-}ok = true$ and $mode_s = \mathtt{needid}$, so Part 5 holds in state s'.

   $\underline{a = shrink\_good_r(ids)}$

   This step can only remove elements not in $good_s$ from $good_r$ when $current\text{-}ok = true$ and $mode_s = \mathtt{needid}$, so Part 5 holds in state s'.

6. $\underline{a = choose\_id(id, m)}$

   Consider this step when $s.current\text{-}ok = true$. The step changes $mode_s$ to $\mathtt{send}$ and changes $last_s$ to a value from $s.good_s$. Since $s.mode_s = \mathtt{needid}$, Part 5 gives us that $s'.last_s \in s.good_r$, so since $good_r$ is not changed in the step, Part 6 holds in s'.

   $\underline{a = shrink\_good_r(ids)}$

   When $current\text{-}ok = true$ and $mode_s = \mathtt{send}$, this step cannot remove $s.last_s$ from $good_r$, so Part 6 holds in s'.

7. $\underline{a = choose\_id(id, m)}$

   Consider this step when $s.current\text{-}ok = true$. The step changes $mode_s$ to $\mathtt{send}$ and changes $last_s$ to a value from $s.good_s$. Since $s.mode_s = \mathtt{needid}$, Invariant 8.2 Part 1 gives us that $s'.last_s \notin s.used_s$. From Invariant 8.4 Part 5 we then get that $s'.last_s \notin s.nack\text{-}buf_r$ which again implies $s'.last_s \notin s'.nack\text{-}buf_r$ since $nack\text{-}buf_r$ is not changed in the step. So, Part 7 holds in state s'.

   $\underline{a = receive\_pkt_{sr}(m, id)}$

   This step can add an identifier to $nack\text{-}buf_r$. Assume $s.current\text{-}ok = true$ and $s.mode_s = \mathtt{send}$. We must show that $s.last_s$ $(= s'.last_s)$ cannot be added to $nack\text{-}buf_r$ under these assumptions. From Part 6 we have that that $s.last_s \in s.good_r$, so from the definition of $receive\_pkt_{sr}(m, id)$ we get that $nack\text{-}buf_r$ is not changed. Thus, Part 7 holds in state s'.

■

Proof of Invariant 8.7

Parts 1 and 2 are reformulations of Invariant 8.6 Parts 3 resp. 4.

■

Proof of Invariant 8.8

- Since initially $mode_s = \texttt{idle}$ and $current\text{-}ack_s = false$, all parts hold.

- For each part of the invariant we consider the critical steps (s, a, s'), where we assume that all parts of this invariant hold in s, and that previously proved invariants hold in both s and s'. Note, for the Parts 1, 2, and 3 that no step, except $choose\_id(id, m)$, can change $last_r$ without also changing $mode_s$ to something other that $\texttt{send}$. Note also that no steps can make $good\text{-}ids$ grow. $good\text{-}ids$ can only shrink.

1. $\underline{a = choose\_id(id, m)}$

   This step changes $mode_s$ to $\texttt{send}$. In state s we get from Invariant 8.4 Part 4 that $s.used_s \supseteq ids(sr)$. From the definition of $choose\_id(id, m)$ we see that $s'.last_s$ is placed at the end of $used_s$, thus by the definition of the partial order of identifiers we see that Part 1 holds in s'.

   $\underline{a = send\_pkt_{sr}(m, id)}$

   This step might add $(m, s.last_s)$ to sr while $mode_s = \texttt{send}$. But since Part 1 is assumed to hold in s, it is obvious that it also holds in s'.

2. $\underline{a = choose\_id(id, m)}$

   Although this step changes $mode_s$ from $\texttt{needid}$ to $\texttt{send}$, it does not make $last_s = last_r$. To see why this is so, note that either $s'.last_r = \texttt{nil}$ in which case the result follows directly (since $s'.last_s \neq \texttt{nil}$ by Invariant 8.1 Part 1) or $s'.last_r = s.last_r \neq \texttt{nil}$ in which case Invariant 8.4 Part 8 implies that $s'.last_r \in s.used_s$ and Invariant 8.2 Part 1 implies that $s'.last_s \notin s.used_s$, so again the result follows. Thus, Part 2 holds in s'.

   $\underline{a = receive\_pkt_{sr}(m, id)}$

   Consider the case where $s.mode_s = s'.mode_s = \texttt{send}$, $id = s.last_s = s'.last_s \in s.good_r$, and $s.mode_r = s'.mode_r \neq \texttt{rec}$. In this case we get $s'.last_s = s'.last_r$. We must show that $(\{s'.last_s\} \cup ids(s'.sr)) \cap s'.good\text{-}ids = \emptyset$. From Invariant 8.4 Parts 3 and 4 we get that $s'.issued_r \supseteq \{s'.last_s\} \cup ids(s'.sr)$. So what remains to be shown is that $(\{s'.last_s\} \cup ids(s'.sr)) \cap s.good_r = \emptyset$. From Part 1 we get that $id \geq (\{s.last_s\} \cup ids(s.sr))$. Since we remove all identifiers less than or equal to id from $good_s$ in this step, and since Invariant 8.4 Part 4 ensures that all packets in sr have identifiers that are related to id, the result follows. Thus, Part 2 holds in s'.

   $\underline{a = send\_pkt_{sr}(m, id)}$

   This step can change sr, but only with a packet with the identifier $s.last_s$. Since we assume that this Part 2 holds in s, it follows that it also holds in s'.

3. $\underline{a = choose\_id(id, m)}$

   Although this step changes $mode_s$ from $\texttt{needid}$ to $\texttt{send}$, it does not make the packet $(s'.last_s, true)$ belong to $s'.rs$. We show why this is so. Since rs is not changed in the step, we get from Invariant 8.4 Part 6 that $s.used_s \supseteq ids(s'.rs)$. Invariant 8.2 Part 1 together with the definition of $choose\_id(id, m)$ gives us $s'.last_s \notin s.used_s$. Thus we get $s'.last_s \notin ids(s'.rs)$ which gives the result. So, Part 3 holds in s'.

$a = send\_pkt_{rs}(id, true)$

Consider this step while $s.mode_s = s'.mode_s = \texttt{send}$ and $id = s.last_r = s.last_s = s'.last_s$. The step might succeed in putting the packet $(s'.last_s, true)$ into the channel. We show that $(\{s'.last_s\} \cup ids(s'.sr)) \cap s'.good\text{-}ids = \emptyset$. From Part 2 we get that $(\{s.last_s\} \cup ids(s.sr)) \cap s.good\text{-}ids = \emptyset$. Since neither $last_s$, sr, nor $good\text{-}ids$ are changed in the step, the result follows directly. So, Part 3 holds in s'.

$a = send\_pkt_{sr}(m, id)$

This step can change sr, but only with a packet with the identifier $s.last_s$. Since we assume that this Part 2 holds in s, it follows that it also holds in s'.

4. $a = receive\_pkt_{rs}(id, b)$

This step can change $mode_s$ to `idle` and $current\text{-}ack_s$ to $true$ if $b = true$ and $id = s.last_s$, thus, $(s.last_s, true)$ must be on $s.rs$. Then Part 3 implies that $ids(s.sr) \cap good\text{-}ids = \emptyset$. It now directly follows that Part 4 holds in state s'.

■

Proof of Invariant 8.9

- Since initially $buf_r = \varepsilon$, all parts of the invariant hold.
- For each part of the invariant we consider the critical steps (s, a, s'), where we assume that all parts of this invariant hold in s, and that previously proved invariants hold in both s and s'.

1. $a = recover_r$

This step changes $mode_r$ to `idle` but at the same time buf_r is made empty, so Part 1 holds in s'.

$a = send\_pkt_{rs}(id, true)$

This step can change $mode_r$ to `idle`, but from Part 2 in state s we get $buf_r = \varepsilon$, so Part 1 holds in s'.

$a = cleanup_r$

This step changes $mode_r$ to `idle` but since $s.mode_r \in \{\texttt{idle}, \texttt{ack}\}$ from the precondition, this part and Part 2 imply that buf_r was already empty. Thus, Part 1 holds in s'.

2. $a = receive\_pkt_{sr}(m, id)$

We consider this step in two different situations

- The step can make buf_r nonempty but at the same time $mode_r$ is changed to `rcvd`.
- The step can change $mode_r$ from `idle` to `ack`, but then Part 1 implies that buf_r was already false.

So, Part 2 holds in state s'.

$\underline{a = receive\_msg(m)}$

This step can change $mode_r$ to $\texttt{ack}$ but this only happens if $s'.buf_r = \varepsilon$, so Part 2 holds in state s'.

3. $\underline{a = choose\_id(id, m)}$

   Although this step makes $mode_s = \texttt{send}$, it does not make the packet $(s'.last_s, true)$ belong to $s'.rs$. The argument is the same as for the corresponding case in the proof of Invariant 8.8 Part 3. So, Part 3 holds in state s'.

   $\underline{a = send\_pkt_{rs}(id, true)}$

   This step can put $(s'.last_s, true)$ into rs but since $s.mode_r = \texttt{ack}$, Part 2 gives us that $s.buf_r (= s'.buf_r) = \varepsilon$. So, Part 3 holds in state s'.

   $\underline{a = receive\_pkt_{sr}(m, id)}$

   This step might add an element to buf_r. We show that this cannot happen while $mode_s = \texttt{send}$ and $(last_s, true) \in rs$. If an element is added to buf_r in the step, then $id \in s.good_r$, i.e., $ids(s.sr) \cup s.good\text{-}ids \neq \emptyset$ but this contradicts Invariant 8.8 Part 3. So, Part 3 holds in state s'.

4. $\underline{a = receive\_pkt_{rs}(id, true)}$

   Consider this step when $id = s.last_s$. Then $(s.last_s, true) \in s.rs$. Since $s.mode_s = \texttt{send}$, Part 3 implies that $s.buf_r = \varepsilon$ which in turn implies that $s'.buf_r = \varepsilon$. So, Part 4 holds in state s'.

   $\underline{a = receive\_pkt_{sr}(m, id)}$

   This step might add an element to buf_r. The argument that this cannot happen while $mode_s = \texttt{idle}$ and $current\text{-}ack_s = true$ is similar to the argument in the corresponding case in the proof of Part 3, only in this case we get a contradiction with Invariant 8.8 Part 4. So, Part 4 holds in state s'.

■

Proof of Invariant 8.10

- Initially $nack\text{-}buf_r = \varepsilon$ and $rs = \emptyset$, so the parts hold.
- For each part of the invariant we consider the critical steps (s, a, s'), where we assume that all parts of this invariant hold in s, and that previously proved invariants hold in both s and s'. Note, that no steps can make *good-ids* grow.

1. $\underline{a = receive\_pkt_{sr}(m, id)}$

   Consider this step when $s.mode_r \neq \texttt{rec}$ and $id \notin s.good_r$. Then id might be added to $nack\text{-}buf_r$. Since $id \notin s.good_r$ and $good_r$ is unchanged in the step we get $s'.nack\text{-}buf_r \cap s'.good_r = \emptyset$ (since we assume that this Part 1 holds in s). From Invariant 8.4 Parts 3 and 5 it follows that $s'.nack\text{-}buf_r \cap \overline{s'.issued_r} = \emptyset$. So, Part 1 holds in state s'.

2. $\underline{a = send\_pkt_{rs}(id, true)}$

   This step might add $(last_r, true)$ to rs but from Invariant 8.4 Part 7 we get that $last_r \notin good\text{-}ids$, so this step cannot violate Part 2.

$\underline{a = \mathit{send\_pkt}_{rs}(id, \mathit{false})}$

Then $id \in s.\mathit{nack\text{-}buf}_r$, so Part 1 directly gives us that this step cannot violate Part 2.

■

Proof of Invariant 8.11

- Initially $mode_s =$ `idle` so both parts hold.
- For each part of the invariant we consider the critical steps (s, a, s'), where we assume that both parts of this invariant hold in s, and that previously proved invariants hold in both s and s'. Note, no action, except $\mathit{choose\_id}(id, m)$, can change $last_s$ without also changing $mode_s$ to non-`send`. Also, from Invariant 8.1 Part 2 we get that all steps that change $last_r$ to `nil` are not critical.

1. $\underline{a = \mathit{choose\_id}(id, m)}$

   Although this step changes $mode_s$ to `send`, it does not make the packet $s'.last_s$ belong to $s'.\mathit{nack\text{-}buf}_r$. We show why this is so. Invariant 8.2 Part 1 implies that $s'.last_s \notin s.used_s$. From Invariant 8.4 Part 5 and the fact that $\mathit{nack\text{-}buf}_r$ is not changed in the step, we get that $s.used_s \supseteq s'.\mathit{nack\text{-}buf}_r$, which gives the result. So, Part 1 holds in state s'.

   $\underline{a = \mathit{receive\_pkt}(m, id)}$

   We consider two cases.

   - Consider the step when $id = last_s$. Then $last_s$ can be added to $\mathit{nack\text{-}buf}_r$ but this can only happen if $last_s \neq last_r$, so Part 1 is not violated.
   - Consider the step when $s.mode_r \neq$ `rec`, $id = last_s$, and $last_s \in s.good_r$. Then $s'.last_s = s'.last_r$. We show that then $s.last_s \notin s.\mathit{nack\text{-}buf}_r$ (which is the same as showing $s'.last_s \notin s'.\mathit{nack\text{-}buf}_r$). First assume $s.last_s \in s.\mathit{nack\text{-}buf}_r$. Then Invariant 8.10 Part 1 implies that $s.last_s \notin s.good_r$, but this contradicts the assumption that $last_s \in s.good_r$. Thus, Part 1 holds in state s'.

2. $\underline{a = \mathit{choose\_id}(id, m)}$

   Although this step changes $mode_s$ to `send`, it does not make the packet $(s'.last_s, \mathit{false})$ belong to $s'.rs$. The argument that this is so is similar to the argument in the corresponding case in the proof of Invariant 8.8 Part 3. So, Part 2 holds in state s'.

   $\underline{a = \mathit{send\_pkt}_{rs}(id, \mathit{false})}$

   Consider this step when $id = last_s$, i.e., $last_s$ is first on $s.\mathit{nack\text{-}buf}_r$. Then Part 1 implies that $s.last_s \neq s.last_r$, so, since neither $last_s$ nor $last_r$ change in the step, Part 2 holds in state s'.

   $\underline{a = \mathit{receive\_pkt}_{sr}(m, id)}$

   Assume $s.mode_r \neq$ `rec` and $last_s = id \in s.good_r$. Then $s'.last_r = s'.last_s$. We show that then $(last_s, \mathit{false}) \notin rs$. First assume $(last_s, \mathit{false}) \in rs$. Then Invariant 8.10 Part 2 implies that $last_s \notin s.\mathit{good\text{-}ids}$, but this contradicts the assumption that $last_s \in s.good_r$. Thus, Part 2 holds in state s'.

■

Proof of Invariant 8.12

- The invariant is explicitly required to hold in all start states.
- We consider the critical steps (s, a, s), where we assume that the invariant holds in s', and that previously proved invariants hold in both s' and s.

1. $\underline{a = recover_r \text{ or } a = shrink\_good_r(ids)}$

   These steps explicitly require the invariant to hold in s.

■

C.2 Proof of Invariants at the C Level

In this section we prove the invariants of ${A^h_{\mathrm{C}}}'$ presented in Section 10.5.2. As above we prove the invariants by induction, proving that they hold in the (unique) start state and that all steps preserve the invariants. As above, in the inductive step of the inductive arguments we only consider "critical steps" that might invalidate the invariant.

In the proofs the steps have the form (s, a, s').

Proof of Invariant 10.1

- Initially all the involved variables are 0, so all parts hold.
- 1. $\underline{a = tick_s(t)}$

     This step changes both $ctime_s$ and $time_s$ to t.

  2. $\underline{a = tick_r(t)}$

     This step changes both $ctime_r$ and $time_r$ to t.

  3. $\underline{a = \nu}$

     The precondition on the time-passing steps of the clock subsystem (and thus on all of C) ensures that $|s'.ctime_s - s'.now| \leq \epsilon$. Part 1 then gives the result.

  4. $\underline{a = \nu}$

     The precondition on the time-passing steps of the clock subsystem (and thus on all of C) ensures that $|s'.ctime_r - s'.now| \leq \epsilon$. Part 2 then gives the result.

  5. Parts 3 and 4 directly implies the result.

■

Proof of Invariant 10.2

- Initially $upper_r = \beta \geq 2\epsilon + l'_r \geq 2\epsilon$. Since initially $now = time_s = time_r = 0$, all the invariants hold.
- 1. $\underline{a = recover_r}$

     This makes $s'.mode_r \neq \texttt{rec}$ but at the same time $s'.upper_r = s'.time_r + \beta \geq s'.time_r + 2\epsilon + l'_r \geq s'.now + \epsilon + l'_r$, where the last inequality follows from Invariant 10.1 Part 4.

     $\underline{a = increase\text{-}upper_r(t)}$

     As for the previous case, $s'.upper_r \geq s'.now + \epsilon + l'_r$.

$a = \nu$

Assume $s.mode_r \neq \texttt{rec}$. From the upper time bound on the class $C^t_{\mathrm{C},r2}$ consisting of all $increase\text{-}upper_r(t)$ actions we have $s'.now \leq s.last(C^t_{\mathrm{C},r2})$. The variable $last(C^t_{\mathrm{C},r2})$ is set to $now + l'_r$ whenever a $recover_r$ step occurs (since then $C^t_{\mathrm{C},r2}$ becomes enabled) or a $increase\text{-}upper_r(t)$ step occurs (since then $increase\text{-}upper_r(t)$ becomes reenabled). Now, since we assume $s.mode_r \neq \texttt{rec}$, let now_0 and $upper_{r,0}$ denote real time and $upper_r$ right after the last $recover_r$ or $increase\text{-}upper_r(t)$ step. Then $s'.now \leq s.last(C^t_{\mathrm{C},r2}) = now_0 + l'_r$, so, $now_0 \geq s'.now - l'_r$. Now, from the $recover_r$ and $increase\text{-}upper_r(t)$ cases above we finally get $s'.upper_r = upper_{r,0} \geq now_0 + \epsilon + l'_r \geq (s'.now - l'_r) + \epsilon + l'_r = s'.now + \epsilon$.

Note: We are here actually departing from our normal way of proving invariants since we use more information, like now_0, than is available in s. What we could have done was to introduce a history variables now_0 that is set to now in $recover_r$ and $increase\text{-}upper_r(t)$ steps. We could then easily have proved the invariants

If $mode_r \neq \texttt{rec}$ then $last(C^t_{\mathrm{C},r2}) = now_0 + l'_r$ and $now \leq now_0 + l'_r$

$s'.upper_r \geq now_0 + \epsilon + l'_r$

from which the result would follow.

We go through the same arguments but have chosen, for brevity, to avoid explicitly introducing the extra history variable.

2. This part follows directly from Part 1 and Invariant 10.1 Part 3.
3. This part follows directly from Part 1 and Invariant 10.1 Part 4.

■

Proof of Invariant 10.3

- Initially $last_s = time_s = 0$ and $mode_s = \texttt{idle}$, so both parts hold.
- 1. $a \in \{choose\_id(t), recover_s, tick_s(t)\}$

  All such steps clearly preserve this part.

  2. $a = choose\_id(t)$

  Changes $mode_s$ to $\texttt{send}$ but also explicitly sets $s'.last_s = t > s.last_s \geq 0$.

■

Proof of Invariant 10.4

Straightforward.

■

Proof of Invariant 10.5

Straightforward.

■

Proof of Invariant 10.6

Straightforward.

■

Proof of Invariant 10.7

- Initially $lower_r = time_s = last_s = 0$, so both parts of the invariant hold.
- 1. No steps can make $time_s$ smaller, so we need only check the steps that make $lower_r$ bigger.

  $\underline{a = recover_r}$

  Then $s'.lower_r = s.upper_r$ and $s.upper_r + 2\epsilon < s.time_r$. Therefore, $s'.lower_r < s.time_r - 2\epsilon \leq s.time_s = s'.time_s$, where we have used Invariant 10.1 Part 5.

  $\underline{a = increase\text{-}lower_r(t)}$

  Then $s'.lower_r < s.time_r - \rho \leq s.time_r - (kl_s + d + 2\epsilon) \leq s.time_r - 2\epsilon \leq s.time_s = s'.time_s$, where we again have used Invariant 10.1 Part 5.

  $\underline{a = receive\_pkt_{sr}(m,t)}$

  The only way for $lower_r$ to increase is for $s'.lower_r = t$ but then, since $((m,t),\_) \in s.sr$, Invariants 10.6 Part 1 and 10.3 Part 1 imply that $s'.lower_r \leq s.last_s \leq s.time_s = s'.time_s$.

  2. $\underline{a \in \{recover_r, increase\text{-}lower_r(t)\}}$

  Same argument as for the previous part.

  $\underline{a = receive\_pkt_{sr}(m,t)}$

  Assume $s'.last_s < s'.time_s$. Since $s.last_s = s'.last_s$ and $s.time_s = s'.time_s$, we also have $s.last_s < s.time_s$. The only way for $lower_r$ to increase is for $s'.lower_r = t$ but then, since $((m,t),\_) \in s.sr$, Invariants 10.6 Part 1 implies that $s'.lower_r \leq s.last_s < s.time_s = s'.time_s$.

  $\underline{a = tick_s(t)}$

  Assume $s'.last_s < s'.time_s$. From Invariant 10.3 Part 1 we have $s.last_s \leq s.time_s$. We consider cases:

  - $s.last_s < s.time_s$
    Then $s.lower_r < s.time_s$ by the inductive hypothesis, so we have $s'.lower_r = s.lower_r < s.time_s \leq s'.time_s$, as needed, where the last inequality follows from the definition of $tick_s(t)$.
  - $s.last_s = s.time_s$
    Then since $s.last_s = s'.last < s'.time_s$ we have $s.time < s'.time$. Since $s'.lower_r = s.lower_r$, and $s.lower_r \leq s.time_s$ by Part 1, we have $s'.lower < s'.time_s$, as needed.

■

Proof of Invariant 10.8

Straightforward.

■

Proof of Invariant 10.9

Straightforward.

■

Proof of Invariant 10.10

- Initially $deadline = \infty$ and $now = 0$, and since $mode_s = \texttt{idle}$ we have $bound = \infty$, so all parts hold.

- 1. $\underline{a = choose\_id(t)}$

  Then $s'.last_s = t$. Let $m = s'.current\text{-}msg_s$.

  If $s.mode_r = s'.mode_r = \texttt{rec}$ then $s'.deadline = s.deadline$ and the induction hypothesis Part 7 implies that $s.deadline = \infty$, so we are done.

  So, assume $s.mode_r \neq \texttt{rec}$. From the precondition of $choose\_id(t)$ we have $t > s.last_s$. Now Invariants 10.5 Part 1 and 10.6 Part 1 imply, since $s'.count_{sr}(m,t) = s.count_{sr}(m,t)$ and $s'.rs = s.rs$, that $s'.count_{sr}(m,t) = 0$ and $(m,t) \notin packets(s'.sr)$. Now, since $C^t_{\mathrm{C},s}$ becomes reenabled in s' we have $s'.last(C^t_{\mathrm{C},s} = s'.now + l_s$. Thus,

$$\begin{aligned} s'.bound &= s'.last(C^t_{\mathrm{C},s}) + (k - 1 - s'.count_{sr}(m,t))l_s + d \\ &= s'.now + l_s + (k-1)l_s + d \\ &= s'.deadline \end{aligned}$$

  That suffices.

  $\underline{a = send\_pkt_{sr}(m,t)}$

  We consider cases

  - $(m,t) \in packets(s.sr)$
    Then $s'.bound = s.bound$ since the $mintime$ of the (p,t) packets does not change. Since also $s'.deadline = s.deadline$, the result follows.
  - $(m,t) \notin packets(s.sr)$
    * $(m,t) \notin packets(s'.sr)$
      Then $s'.count_{sr}(m,t) = s.count_{sr}(m,t) + 1$. We now have

$$\begin{aligned} s'.bound &= s'.last(C^t_{\mathrm{C},s}) + (k - 1 - s'.count_{sr}(m,t))l_s + d \\ &= s'.now + l_s + (k - 1 - s'.count_{sr}(m,t))l_s + d \\ &= s'.now + (k - 1 - s.count_{sr}(m,t))l_s + d \\ &\leq s.last(C^t_{\mathrm{C},s}) + (k - 1 - s.count_{sr}(m,t))l_s + d \\ &= s.bound \end{aligned}$$

      The induction hypothesis Part 1 now implies

$$s'.deadline = s.deadline \geq s.bound \geq s'.bound$$

    * $(m,t) \in packets(s'.sr)$
      Then $s'.bound = d + s'.now$ and

$$\begin{aligned} s.bound &= s.last(C^t_{\mathrm{C},s}) + (k - 1 - s.count_{sr}(m,t))l_s + d \\ &\geq s.last(C^t_{\mathrm{C},s}) + d \\ &\geq s'.now + d \\ &= s'.bound \end{aligned}$$

      where the first inequality follows from Invariant 10.5 Part 2 and the second inequality follows from facts that time cannot pass beyond any $last(C)$ variable and $s'.now = s.now$.
      The induction hypothesis Part 1 now implies

$$s'.deadline = s.deadline \geq s.bound \geq s'.bound$$

$a = receive\_pkt_{sr}(m, t)$

For such a step to change either *bound* or *deadline*, i.e., for such a step to be able to invalidate the invariant part under consideration, we must have $s.mode_s = \texttt{send}(= s'.mode_s)$ and $t = s.last_s\,(= s'.last_s)$. Invariant 10.6 Part 2 then implies that $m = s.current\text{-}msg_s\,(= s'.current\text{-}msg_s)$.

If $s.deadline = \infty$, then also $s'.deadline = \infty$ and the result follows.

So, assume $s.deadline \neq \infty$. The induction hypothesis Part 7 then implies $s.mode_r \neq \texttt{rec}$.

We now show that $s.lower_r < t \leq s.upper_r$.
The lower bound follows from the induction hypothesis Part 6 and the fact that $t = s.last_s$.
For the upper bound we have from Invariants 10.2 Part 2 and 10.3 Part 1 that $s.upper_r \geq s.time_s \geq s.last_s = t$.
Then from the definition of $receive\_pkt_{sr}(m, t)$ we see that $s'.deadline = \infty$, and the result follows.

$a = receive\_pkt_{rs}(t, b)$

For such a step to be able to invalidate the invariant part under consideration, we must have $s.mode_s = \texttt{send}$ and $s.last_s = t$.
Then Invariant 10.6 Part 6 implies that $s.last_s = t \leq s.lower_r$, but then the induction hypothesis Part 6 implies that $s'.deadline = s.deadline = \infty$. That suffices.

2. $a = choose\_id(t)$

   Then Invariant 10.5 Part 1 and the definitions of *bound* and $last(C^t_{\mathrm{C},s})$ imply that

   $$s'.bound = s'.now + l_s + (k-1)l_s + d \geq s'.now$$

   $a = send\_pkt_{sr}(m, t)$

   We consider cases

   - $(m, t) \in packets(s'.sr)$
     * $(m, t) \in packets(s.sr)$ Then $s'.bound = s.bound$ (uses the fact that Invariant 10.9 Part 1 implies that $mintime((m, t), s'.sr) = mintime((m, t), s.sr))$, so the result follows from the induction hypothesis.
     * $(m, t) \notin packets(s.sr)$ Then $s'.bound = s'.now + d \geq s'.now$.
   - $(m, t) \notin packets(s'.sr)$ Then $s'.last(C^t_{\mathrm{C},s}) = s'.now + l_s$, so Invariant 10.5 Part 2 implies
     $$s'.bound = s'.now + l_s + (k - 1 - s'.count_{sr}(m, t))l_s + d \geq s'.now$$

   $receive\_pkt_{sr}(m, t)$

   For such a step to change *bound* we must have $s.mode_s = \texttt{send}$, $s.last_s = t$, and $s.current\text{-}msg_s = m$. In all other cases the induction hypothesis immediately gives the result.

   The step removes $((m, t), t')$, for some t', from sr. If $t' \neq mintime((m, t), s.sr)$ then $s'.bound = s.bound$, and again the induction hypothesis gives the result. So, assume $t' = mintime((m, t), s.sr)$.

   We consider cases

- $(m,t) \in packets(s'.sr$
  Then $mintime((m,t), s'.sr) \geq mintime((m,t), s.sr)$ which implies that $s'.bound \geq s.bound$ and the result follows.
- $(m,t) \notin packets(s'.sr$
  Then, since $s'.last(C^t_{C,s}) \geq s'.now$ we have (with a little help from Invariant 10.5 Part 2)
  $$s'.bound = s'.last(C^t_{C,s}) + (k - 1 - s'.count_{sr}(m,t))l_s + d \geq s'.now$$

<u>$a = \nu$</u>

If $s.mode_s = s'.mode_s \neq$ `send`, then $s'.bound = \infty$ and the result follows. So, assume $s.mode_s = s'.mode_s =$ `send`

Let $m = s.current\text{-}msg_s = s'.currnet\text{-}msg_s$ and $t = s.last_s = s'.last_s$. We consider cases

- $(m,t) \in packets(s.sr)$
  Then $((m,t), mintime((m,t), s.sr)) \in s.sr$ and from the precondition of the time passing steps of the channel sr we have $s'.now \leq mintime((m,t), s.sr)$. Thus, since $s'.sr = s.sr$,
  $$s'.now \leq mintime((m,t), s.sr) \leq mintime((m,t), s'.sr) + d = s'.bound$$
- $(m,t) \notin packets(s.sr)$
  Then, since $s'.last(C^t_{C,s}) \geq s'.now$ we have (with a little help from Invariant 10.5 Part 2)
  $$s'.bound = s'.last(C^t_{C,s}) + (k - 1 - s'.count_{sr}(m,t))l_s + d \geq s'.now$$

3. This part follows directly from Parts 1 and 2.
4. <u>$a = choose\_id(t)$</u>

   If $s.mode_r =$ `rec` then $s'.deadline = s.deadline = \infty$, by the induction hypothesis Part 7, so the result follows.

   So, assume $s.mode_r \leq$ `rec`. Then $s'.deadline = s'.now + kl_s + d$ and $s'.last_s = s'.time_s$. Invariant 10.1 Part 3 then implies that $s'.deadline \leq s'.last_s + \epsilon + kl_s + d$.

   <u>$a = recover_s$</u>

   Then the induction hypothesis Part 7 implies that $s.deadline = \infty$, and since we have $s'.deadline = s.deadline$, the result follows.
5. This part follows directly from Parts 3 and 4.
6. <u>$a \in \{recover_s, recover_r\}$</u>

   Then by the induction hypothesis Part 7 we have $s'.deadline = s.deadline = \infty$. That suffices.

   <u>$a = choose\_id(t)$</u>

   Then $s'.last_s = s'.time_s = s.time_s > s.last_s$, by definition of *choose_id*. By Invariant 10.7 Part 2, $s.lower_r < s.time_s$. But since $s'.lower_r = s.lower_r$ and $s.time_s = s'.last_s$, we have $s'.lower_r < s'.last_s$, as needed.

$a = \textit{increase-lower}_r(t)$

We only need to check such steps when $s'.\textit{deadline} = s.\textit{deadline} \neq \infty$.

By definition of $\textit{increase-lower}_r(t)$, we have $s'.\textit{lower}_r < s'.\textit{time}_r - \rho \leq s'.\textit{time}_r - (kl_s + d + 2\epsilon)$. It suffices to show that this is less than or equal to $s'.\textit{last}_s$. Since $s'.\textit{deadline} \neq \infty$, Part 5 implies that $s'.\textit{now} \leq s'.\textit{last}_s + \epsilon + kl_s + d$. By Invariant 10.1 Part 4, we know that $s'.\textit{time}_r \leq s'.\textit{now} + \epsilon$. Therefore, $s'.\textit{time}_r \leq s'.\textit{last}_s + kl_s + d + 2\epsilon$. This suffices.

$a = \textit{receive\_pkt}_{sr}(m, t)$

This increases $\textit{lower}_r$ if $s.\textit{mode}_r \neq \texttt{rec}$ and $s.\textit{lower}_r < t \leq s.\textit{upper}_r$.

If $s.\textit{deadline} = \infty$ then also $s'.\textit{deadline} = \infty$ and the result follows.

So assume $s.\textit{deadline} \neq \infty$. Then induction hypothesis Part 7 implies that $s.\textit{mode}_s = \texttt{send}$. Now, if $t = s.\textit{last}_s$ then $s'.\textit{deadline} = \infty$ and the result follows. If $t \neq s.\textit{last}_s$, then Invariant 10.6 Part 1 implies that $t < s.\textit{last}_s$. Then, since $s'.\textit{lower}_r = t$ and $s'.\textit{last}_s = s.\textit{last}_s$, we get $s'.\textit{lower}_r < s'.\textit{last}_s$, as needed.

7. Straightforward except for the case $a = \textit{receive\_pkt}_{rs}(t, b)$.

$a = \textit{receive\_pkt}_{rs}(t, b)$

This may invalidate the invariant by changing $\textit{mode}_s$ to `idle` if we have $t = s.\textit{last}_s$ and $s.\textit{mode}_s = \texttt{send}$.

Invariant 10.6 Part 6 implies that $s.\textit{last}_s \leq s.\textit{lower}_r$. From the induction hypothesis Part 6 we then get $s.\textit{deadline} = \infty$, and since $s'.\textit{deadline} = s.\textit{deadline}$ the result follows.

■

Proof of Invariant 10.12

Straightforward.

■

Proof of Invariant 10.13

Straightforward.

■