An Inheritance-Based Technique for Building Simulation Proofs Incrementally[*]

Idit Keidar, Roger Khazan, Nancy Lynch,
MIT Lab for Computer Science
200 Technology Sq., Room 367
Cambridge, MA 02139, USA
{idish, roger, lynch}@theory.lcs.mit.edu
+1 617 253 6054

Alex Shvartsman
University of Connecticut
Comp. Science and Engineering Dept.
Storrs, CT 06269-3155, USA
aas@cse.uconn.edu
+1 860 486 2672

January 3, 2002

Abstract

This paper presents a formal technique for *incremental* construction of system specifications, algorithm descriptions, and *simulation proofs* showing that algorithms meet their specifications.

The technique for building specifications and algorithms incrementally allows a child specification or algorithm to inherit from its parent by two forms of incremental modification: (a) *signature extension*, where new actions are added to the parent, and (b) *specialization* (subtyping), where the child's behavior is a specialization (restriction) of the parent's behavior. The combination of signature extension and specialization provides a powerful and expressive incremental modification mechanism for introducing new types of behavior without overriding behavior of the parent; this mechanism corresponds to the subclassing for extension form of inheritance.

In the case when incremental modifications are applied to both a parent specification S and a parent algorithm A, the technique allows a simulation proof showing that the child algorithm A' implements the child specification S' to be constructed incrementally by extending a simulation proof that algorithm A implements specification S. The new proof involves reasoning about the modifications only, without repeating the reasoning done in the original simulation proof.

The paper presents the technique mathematically, in terms of automata. The technique has been used to model and verify a complex middleware system; the methodology and results of that experiment are summarized in this paper.

General term: Verification. **Categories and subject descriptors:** F.3.1 Specifying and Verifying and Reasoning about Programs; D.2.1 Requirements/Specifications: Methodologies - object-oriented. **Additional keywords and phrases:** inheritance by specialization and subclassing for extension, simulation proofs, refinements, incremental proof techniques, proof reuse.

[*]Pre-print of article in ACM Transactions on Software Engineering and Methodology (TOSEM) 11(1), pages 1–29, January 2002. Preliminary version appeared in [26]. This work was supported by Air Force Aerospace Research (OSR) contracts F49620-00-1-0097 and F49620-00-1-0327, NTT contract MIT9904-12, and NSF grants CCR-9909114 and EIA-9901592. The work of the fourth author was additionally supported by the NSF Grant CCR-9988304 and the NSF Career Award CCR-9984778.

1 Introduction

Formal modeling and validation of software systems is a major challenge, because of their size and complexity. Among the factors that could increase widespread usage of formal methods is improved cost-effectiveness and scalability (see [17, 19]). Current software engineering practice addresses difficulties of building complex systems by the use of incremental development techniques based on an object-oriented approach. We believe that successful efforts in system modeling and validation will also require incremental techniques, which will enable reuse of models and proofs.

In this paper we provide a framework for reuse of proofs analogous and complementary to the reuse provided by object-oriented software engineering methodologies. Specifically, we present a formal technique for incrementally constructing safety specifications (requirements), abstract algorithm descriptions, and *simulation proofs* that algorithms meet their specifications. Simulation proofs are one of the most important techniques for proving properties of complex systems; such proofs exhibit a *simulation relation* (also known as *abstraction* or *refinement*) between a formal description of a system (algorithm) and its specification [2, 33, 23, 31]. These two formal descriptions are stated using the same notation. The distinction between the "specification" and the "algorithm" for a system is based only on the intention of using the former as the high-level model and the latter as the low-level model of the system. In general, our technique applies to modeling and verification at any level of abstraction.

The formalism presented in this paper has evolved with our experience in the context of a large-scale modeling and validation project: we have successfully used this technique for modeling and validating a complex group communication system [25] that is implemented in C++, and that interacts with two other services developed by different teams. We have modeled both the specification and the algorithm of this system incrementally, at each step strengthening the model with additional constraints. Reuse of models and proofs was essential in order to make this task feasible. For example, it has allowed us to avoid repeating the five-page long correctness proof of the algorithm that provides the basic semantics when proving the correctness of the algorithm that extends the first algorithm with more sophisticated semantics. The correctness proof of the most sophisticated algorithm, by comparison, was only two and a half pages long. We describe our experience with that project as well as the methodology that evolved from it in Section 6.

Our approach to the reuse of specifications and algorithms through inheritance uses incremental modification to derive a new component (specification or algorithm), called *child*, from an existing component called *parent*. Specifically, we present two constructions for modifying existing components:

1. We allow the child to *specialize* the parent by reusing its state in a read-only fashion, by adding new state components (which are allowed to be modified), and by constraining the set of behaviors of the parent. This corresponds to the *subtyping* view of inheritance [7]. We will show that any observable behavior of the child is *subsumed* (see [1]) by the possible behaviors of the parent, making our specialization analogous to *substitution inheritance* [7]. In particular, the child can be used anywhere the parent can be used.

2. A child can also be derived from a parent by means of *signature extension*. In this case the state of the parent is unchanged, but the child may include new actions not found in the parent and new parameters to actions that exist at the parent. When such new actions and parameters are hidden, then any behavior of the child is exactly as some behavior of the parent.

The combination of signature extension and specialization provides a powerful mechanism for incrementally constructing specifications and algorithms; this combination corresponds to the *subclassing for extension* form of inheritance [7].

Consider the following example. The parent defines an unordered point-to-point messaging service with the send(msg) and recv(msg) interface. Specialization can be used to extend the parent to preserve fifo ordering, by restricting recv(msg) actions to deliver messages only according to the sending order. Subclassing for extension can be used to augment the messaging service with an acknowledgment mechanism: The parent's signature can be extended with ack(msg) actions, and then the parent can be specialized to handle acknowledgments by allowing an ack(msg) action to occur only after the corresponding message was received by the recipient. The specialization and subclassing for extension constructs can be applied at both the specification level and the algorithm level in a way that preserves the relationship between the specification and the algorithm.

The main technical challenge addressed in this paper is the provision of a formal framework for the reuse of simulation proofs, especially for the specialization construct. Consider the example in Figure 1: Let S be a specification, and A an abstract algorithm description. Assume that we have proven that A *implements* S using a simulation relation $\mathtt{R_p}$. Assume further that we specialize the specification S, yielding a new *child* specification S'. At the same time, we specialize the algorithm A to construct an algorithm A' which supports the additional semantics required by S'.

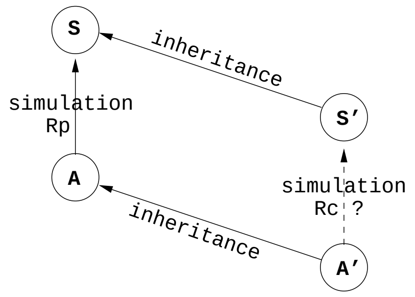

Figure 1: Algorithm $\mathtt{A}$ simulates specification $\mathtt{S}$ with $\mathtt{R_p}$. Can $\mathtt{R_p}$ be reused for building a simulation $\mathtt{R_c}$ from a child $\mathtt{A'}$ of $\mathtt{A}$ to a child $\mathtt{S'}$ of $\mathtt{S}$?

When proving that A' implements S', we would like to rely on the fact that we have already proven that A implements S, and to avoid the need to repeat the same reasoning. We would like to reason only about the new features introduced by S' and A'. The proof reuse theorem provides the means for incrementally building simulation proofs in this manner.

Simulation proofs lend themselves naturally to be supported by interactive theorem provers. Such proofs typically break down into many simple cases based on different actions. These can be checked by hand or with the help of interactive theorem provers. Our incremental simulation proofs break down in a similar fashion.

The formalism described in this paper is presented in the context of the I/O automaton model [31, 32], but it is more general than that particular model. We believe that the essence of our approach is applicable to any state-transition formal model, such as TLA [28], UNITY [34], and Process Algebra [22, 33]. I/O automata have been widely used to model complex systems and reason about them [18, 12, 8, 5, 21]. An important feature of the I/O automaton model is its strong support of composition, which allows an automaton representing a complex system to be constructed by composing automata representing individual system components. For example, Hickey et al. [21] used the compositional approach for modeling and verification of certain modules in Ensemble [16], a large-scale, modularly structured, group communication system. Composition and inheritance are two complementary means for modular system design.

Inheritance, as a means for modular system design, has been a subject of extensive research for decades. Many researchers employed formal methods to define various inheritance constructs and study their properties [1, 5, 10, 11, 20, 24, 29, 30, 35, 39, 37, 38, 4, 34, 15]. Our distinguishing contribution is a provision of a formal framework that allows simulation proofs to be constructed incrementally when inheritance is applied at two levels: specification and algorithm. Thus, we extend the applicability of inheritance from the realm of incremental system design to the realm of incremental system verification.

Roadmap

The rest of the paper is organized as follows: Section 2 reviews and exemplifies the I/O automaton model and the simulation proof technique; the examples of subsequent sections are built upon the ones presented in Section 2. In Section 3, we formally define the specialization construct and investigate its properties. Then, in Section 4, we present a general theorem that enables incremental verification of systems that are modeled and specified incrementally using the specialization construct. This theorem provides the foundation for incremental construction of simulation proofs, and is the key contribution of this paper. In Section 5, we extend the theory of incremental modeling and proof construction to the subclassing for extension form of inheritance: we give a formal definition of the signature extension construct and show how it can be used in conjunction with the specialization construct to achieve subclassing for extension; we then extend the proof-reuse theory presented in Section 4 to this situation.

The paper employs a simple running example to illustrate the use of the presented formalism. Section 6 illustrates the utility of this formalism by describing its use in a large-scale modeling and verification project. In Section 7 we discuss the modeling methodology that we have used with our formalism in the context of the same project.

Section 8 compares our results with related work. Section 9 concludes the paper.

2 Background: I/O Automata and Simulation Proofs

This section presents background on the I/O automaton model, based on [31], Ch. 8. In this model, a system component is described as a state-machine, called an *I/O automaton*. The transitions of the automaton are associated with named actions, classified as *input*, *output* and *internal*. Input and output actions model the component's interaction with other components, while internal actions are externally unobservable. Note that an action can be either an input or an output, but not both; a function call that returns a value can be modeled using two actions – an input and an output.

Formally, an I/O automaton `A` consists of: a signature `sig(A)`, consisting of input, output, and internal actions; a set of states `states(A)`; a set of start states `start(A)`; and a state-transition relation `trans(A)` — a subset of `states(A)` $\times$ `sig(A)` $\times$ `states(A)`. An action is *external* if it is not internal; the part of an automaton's signature consisting of its external actions is called the automaton's *external signature*. Complex automata can be constructed by *composing* smaller automata that interact via their input and output actions.

An action π is said to be *enabled* in a state `s` if the automaton has a transition of the form (`s`, π, `s'`); input actions are enabled in every state. An *execution fragment* of an automaton `A` is an alternating sequence of states and actions such that every successive triple of this sequence is an allowable transition; an empty step (`s`, ϵ, `s`) is also an execution fragment. An *execution* is an execution fragment that begins with a start state. The *trace* of an execution α of `A`, denoted by `trace(`α`)`, is a subsequence of α consisting of all the external actions in α. We denote the set of executions of `A` by `execs(A)`, and the set of traces of `A` by `traces(A)`. When reasoning about an automaton, we are only interested in its externally-observable behavior as reflected in its traces.

I/O automata are conveniently presented using the *precondition-effect* style. In this style, typed state variables with initial values specify the set of states and the start states. Transitions are grouped by action name, and are specified using a `pre:` block with preconditions (guards) on the states in which the action is enabled and an `eff:` block which specifies how the pre-state is modified. The effect is executed *atomically* to yield the post-state.

Example 2.1 *Figure 2 presents an I/O automaton,* `UpSeq`*, that prints nondecreasing sequences of integers. The automaton is expressed in the precondition-effect notation. The signature of* `UpSeq` *consists of output actions of the type* `print(x)`*, where* `x` *is an integer. The state of* `UpSeq` *consists of a single integer variable,* `last`*, initialized to an arbitrary value. The transitions of* `UpSeq` *specify that action* `print(x)`*, with a given* `x`*, is enabled in every state in which* `x` $\geq$ `last`*, as enforced by the* `pre:` *statement; once* `print(x)` *occurs, the automaton moves into a state in which* `last` $=$ `x`*, as specified by the* `eff:` *statement.*

The following is a sample infinite execution of `UpSeq`*, where square brackets represent states of* `UpSeq`*, that is, values of* `last`*:*

$$[3], \texttt{print(5)}, [5], \texttt{print(11)}, [11], \texttt{print(11)}, [11], \texttt{print(14)}, [14], \ldots$$

The trace of this execution is "`print(5), print(11), print(11), print(14), ...`*." In general, the set of traces of* `UpSeq` *is the set of all possible sequences printing nondecreasing integers, both finite and infinite.*

```
AUTOMATON UpSeq

Signature:    Output print(x), x ∈ Integer

State:        last ∈ Integer, initially arbitrary

Transitions: OUTPUT print(x)
              pre: x ≥ last
              eff: last ← x
```

Figure 2: Automaton `UpSeq` printing a nondecreasing sequence of integers.

A common way to specify which traces a system is allowed to exhibit is to define an abstract, high-level I/O automaton that generates the allowed set of traces. If every trace of the automaton modeling the system is also a trace of the specification automaton, then the system always does what is allowed by its specification. In this case, we say that the system automaton *satisfies*, or *implements*, the specification automaton. By way of an example, regard automaton `UpSeq` of Example 2.1 to be a specification for the sequences of nondecreasing integers. In order for some automaton to satisfy this specification, any possible trace of this automaton has to be a trace of `UpSeq`. In the following example we present such an automaton.

Example 2.2 *Figure 3 contains an automaton,* `FibSeq`*, that prints, as its sole infinite trace, the suffix of the Fibonacci sequence that begins with "$1, 2, \dots$". The Fibonacci sequence is an infinite sequence that begins with* 0 *and* 1*, and in which every further element is equal to the sum of the two preceding elements.*

```
AUTOMATON FibSeq

Signature:    Output print(x), x ∈ Integer

State:        n ∈ Integer, initially 0
              m ∈ Integer, initially 1

Transitions: OUTPUT print(x)
              pre: x = n + m
              eff: n ← m
                   m ← x
```

Figure 3: Automaton `FibSeq` printing the Fibonacci sequence.

The signature of the `FibSeq` *automaton is the same as that of* `UpSeq`*. The state of* `FibSeq` *consists of two integer variables,* `n` *and* `m`*, initialized to* `0` *and* `1`*, respectively. The transitions of* `FibSeq` *specify that action* `print(x)` *is enabled in every state in which* `x` *is equal to the sum of* `n` *and* `m`*, and that, once* `print(x)` *occurs, the automaton moves into a state in which* `n` *has the value that* `m` *has in the pre-state, and* `m` *has the value of* `x`*. Thus,* `FibSeq` *uses* `n` *and* `m` *to store the last two elements printed and to compute from them the next element to be printed. The traces generated by* `FibSeq` *are the infinite sequence of Fibonacci numbers, "*`print(1), print(2), print(3), print(5), print(8), print(13), ...` *" and all of its prefixes.*

Every trace of `FibSeq` is clearly a trace of `UpSeq`. Therefore, automaton `FibSeq` satisfies, or implements, automaton `UpSeq`. But how can we prove this formally?

A common technique for establishing that the set of traces of one automaton is included in the set of traces of another is to exhibit a so-called *simulation* relation (also known as an *abstraction relation*) that relates the states of the two automata and to prove that this relation satisfies certain conditions [2, 33, 23, 31], as defined below:

Definition 2.1 *Let* `A` *and* `S` *be two automata with the same external signature. A relation* $\mathtt{R} \subseteq \mathtt{states(A)} \times \mathtt{states(S)}$ *is a* simulation from `A` to `S` *if it satisfies the following two conditions:*

1. *If* `t` *is any initial state of* `A`*, then there is an initial state* `s` *of* `S` *such that* $\mathtt{s} \in \mathtt{R(t)}$*, where we use notation* `R(t)` *as an abbreviation for* $\{\mathtt{s} : (\mathtt{t}, \mathtt{s}) \in \mathtt{R}\}$*.*

2. *If* `t` *and* $\mathtt{s} \in \mathtt{R(t)}$ *are reachable states of* `A` *and* `S` *respectively, and if* $(\mathtt{t}, \pi, \mathtt{t}')$ *is a step of* `A`*, then there exists an execution fragment of* `S` *from* `s` *to some* $\mathtt{s}' \in \mathtt{R}(\mathtt{t}')$*, having the same trace as step* $(\mathtt{t}, \pi, \mathtt{t}')$*. The latter condition means that the only external action in the execution fragment is* π*.*

The two conditions above guarantee that whatever steps `A` executes, there is always a way for `S` to produce the same trace. The following theorem (from [31], Ch. 8) expresses this property formally:

Theorem 2.1 *If* `A` *and* `S` *are two automata with the same external signature and if* `R` *is a simulation from* `A` *to* `S` *then* $\mathtt{traces(A)} \subseteq \mathtt{traces(S)}$*.*

Any finite trace inclusion can be shown by using simulation relations, possibly after adding a special kind of variables, called "prophecy variables" [2, 36].

In some cases, a simulation relation is actually a function from states of the implementation automaton to the states of the specification automaton. In this case it is called a *simulation mapping* (also known as *abstraction function* or *refinement*). If `R` is a simulation function and `t` is a state of the implementation automaton, we use `R(t)` to denote the corresponding state of the specification automaton.

Example 2.3 *We illustrate the simulation technique by presenting a simulation function* `R` *from* `FibSeq` *to* `UpSeq`*.* `R` *maps a state* `t` *of* `FibSeq` *to the state* `s` *of* `UpSeq` *with* `s.last` = `t.m`*, where* `s.last` *denotes an instance of variable* `last` *in state* `s`*, and* `t.m` *denotes* `m` *in state* `t`*. We now argue that* `R` *satisfies Definition 2.1:*

1. *In the initial state* $\mathtt{t}_0$ *of* `FibSeq`*,* $\mathtt{t}_0\mathtt{.m} = 1$*; therefore* $\mathtt{R}(\mathtt{t}_0)\mathtt{.last} = 1$*, which is a valid initial state of* `UpSeq`*.*

2. *Consider a step* $(\mathtt{t}, \mathtt{print(x)}, \mathtt{t}')$ *of* FibSeq. *We claim that* $(\mathtt{R(t)}, \mathtt{print(x)}, \mathtt{R(t')})$ *is a legal step of* UpSeq.

   (a) *We show that, in state* R(t) *of* UpSeq, print(x) *is enabled, that is, that its precondition,* $\mathtt{x} \geq \mathtt{R(t).last}$, *is satisfied. The fact that* $(\mathtt{t}, \mathtt{print(x)}, \mathtt{t}')$ *is a step of* FibSeq *implies that the precondition,* $\mathtt{x} = \mathtt{t.n} + \mathtt{t.m}$, *holds in state* t. *Since* R(t).last *is equal to* t.m *by definition of* R, $\mathtt{x} = \mathtt{t.n} + \mathtt{R(t).last}$. *Therefore,* $\mathtt{x} \geq \mathtt{R(t).last}$, *since* $\mathtt{t.n} \geq 0$, *as stated in the following invariant:*

   Invariant 2.1 *In every reachable state* t *of* FibSeq, $\mathtt{t.n} \geq 0$ *and* $\mathtt{t.m} \geq 0$.
   Proof: *The proposition is true in the initial state* $\mathtt{t}_0$, *since* $\mathtt{t}_0.\mathtt{n} = 0$ *and* $\mathtt{t}_0.\mathtt{m} = 1$. *The proposition is true in state* t′, *after a step* $(\mathtt{t}, \mathtt{print(x)}, \mathtt{t}')$ *of* FibSeq, *assuming it is true in state* t, *since* $\mathtt{t}'.\mathtt{n} = \mathtt{t.m} \geq 0$ *and* $\mathtt{t}'.\mathtt{m} = \mathtt{t.n} + \mathtt{t.m} \geq 0$. ■

   (b) *After* print(x) *occurs in state* R(t), *the value of* last *in the resulting post-state* s′ *is* x *(see Figure 2). In state* t′, *the value of* m *is also* x *(see Figure 3). Hence, by definition of* R, $\mathtt{s}' = \mathtt{R(t')}$.

Therefore, R *is a simulation mapping from* FibSeq *to* UpSeq, *and, as implied by Theorem 2.1,* FibSeq *satisfies* UpSeq.

3 Specialization

We now present the *specialization* construct for creating a child automaton by specializing the parent automaton. This construct captures the notion of subtyping [7]. In the next section, we present the main technical contribution of this paper: a theorem that allows one to construct a simulation proof from a specialization of an algorithm to a specialization of its specification by extending the original simulation proof from the algorithm to its specification.

The specialization construct defined below operates on a parent automaton, and accepts three additional parameters: a *state extension* – the new state components, an *initial state extension* – the initial values of the new state components, and a *transition restriction* specifying how the child specializes the parent's transitions.

Definition 3.1 (Specialization) *Let* A *be an automaton;* N *be any set of states, called a* state extension*;* $\mathtt{N}_0$ *be a non-empty subset of* N, *called an* initial state extension*; and* $\mathtt{TR} \subseteq (\mathtt{states(A)} \times \mathtt{N}) \times \mathtt{sig(A)} \times \mathtt{N}$ *be a relation, called a* transition restriction.
Then $\mathtt{specialize(A)(N, N_0, TR)}$ *defines the following automaton* A′*:*

- $\mathtt{sig(A')} = \mathtt{sig(A)}$;
- $\mathtt{states(A')} = \mathtt{states(A)} \times \mathtt{N}$;
- $\mathtt{start(A')} = \mathtt{start(A)} \times \mathtt{N_0}$;
- $\mathtt{trans(A')} = \{ (\langle \mathtt{t_p}, \mathtt{t_n} \rangle, \pi, \langle \mathtt{t'_p}, \mathtt{t'_n} \rangle) : (\mathtt{t_p}, \pi, \mathtt{t'_p}) \in \mathtt{trans(A)} \wedge (\langle \mathtt{t_p}, \mathtt{t_n} \rangle, \pi, \mathtt{t'_n}) \in \mathtt{TR} \}$, *where* $\langle \mathtt{t_p}, \mathtt{t_n} \rangle$ *denotes a state in* states(A′).

Notation 3.2 *If* $\mathtt{A}' = \mathtt{specialize(A)(N,N_0,TR)}$ *we use the following notation: Given* $\mathtt{t} \in \mathtt{states(A')}$*, we write* $\mathtt{t}|_{\mathtt{p}}$ *to denote its parent component and* $\mathtt{t}|_{\mathtt{n}}$ *to denote its new component. If* α *is an execution fragment of* $\mathtt{A}'$*, then* $\alpha|_{\mathtt{p}}$ *and* $\alpha|_{\mathtt{n}}$ *denote sequences obtained by replacing each state* $\mathtt{t}$ *in* α *with* $\mathtt{t}|_{\mathtt{p}}$ *and* $\mathtt{t}|_{\mathtt{n}}$*, respectively.*

In the precondition-effect notation, a transition restriction (TR) can be specified for each action π by (a) additional preconditions that a child places on π, and (b) additional effects that specify how the *new* state components are modified as a result of a child taking a step involving π. Note that these additional effects can rely on but cannot modify the parent's state components. The additional preconditions work in conjunction with the preconditions placed on π by the parent automaton, and the additional effects are executed before the parent's effects; thus, when the additional effects read parent state components, they observe their pre-state values. The transition restriction expressed in this style is the union of the following two sets:

- All triples of the form $(\mathtt{t}, \pi, \mathtt{t}|_{\mathtt{n}})$ for which π is not mentioned in the code for $\mathtt{A}'$, that is, for which $\mathtt{A}'$ does not restrict transitions involving π. Note that the post-state $\mathtt{t}|_{\mathtt{n}}$ is the same as the new state component of the pre-state `t`.
- All triples $(\mathtt{t}, \pi, \mathtt{t}'_{\mathtt{n}})$ for which state `t` satisfies the new preconditions on π placed by $\mathtt{A}'$, and state $\mathtt{t}'_{\mathtt{n}}$ is the result of applying π's new effects to `t`.

Example 3.1 *Figure 4 below illustrates the use of the specialization construct. It presents precondition-effect code for automaton* `AccSeq`*, which specializes automaton* `UpSeq` *of Figure 2 on page 6 to print only* accelerating *sequences, that is, sequences in which the differences between consecutive elements are nondecreasing (in addition to the sequence itself being nondecreasing).*

```
AUTOMATON AccSeq  SPECIALIZES UpSeq

State Extension:    diff ∈ Integer, initially arbitrary

Transitions Restriction:
                OUTPUT print(x)
                new pre: x - last ≥ diff
                new eff: diff ← x - last
```

Figure 4: Automaton `AccSeq` printing accelerating sequences of integers.

`AccSeq` *extends the state of* `UpSeq` *with a new integer variable* `diff` *having an arbitrary initial value. This variable is used for storing the difference between the last pair of elements printed. The new precondition placed on* `print(x)` *states that* $\mathtt{x} - \mathtt{last}$ *has to be greater than or equal to* `diff`*; it works in conjunction with the precondition,* $\mathtt{x} \geq \mathtt{last}$*, of* `print(x)` *in* `UpSeq`*. The new effect updates* `diff` *to be the current difference,* $\mathtt{x} - \mathtt{last}$*; it occurs before the effect that updates* `last` *in* `UpSeq`*.*

As a result of the new precondition and effect, transitions of $\mathtt{UpSeq}$ *are restricted to only those in which* $\mathtt{diff}$ *is non-decreasing. Thus, the sample trace of* $\mathtt{UpSeq}$ *given in Example 2.1 is* not *a trace of* $\mathtt{AccSeq}$ *because* $(\mathtt{11} - \mathtt{11}) \not\geq (\mathtt{11} - \mathtt{5})$, *while that in Example 2.2 is.*

Our specialization construct is defined so that any behavior of a child is allowed by its parent. Theorem 3.1 below states this property formally: it says that (1) every execution α of a specialization $\mathtt{A}'$ of an automaton $\mathtt{A}$ is also an execution of $\mathtt{A}$ when the state extension of $\mathtt{A}'$ is projected out from α; and (2) every trace of $\mathtt{A}'$ is a trace of $\mathtt{A}$.

Theorem 3.1 *If* $\mathtt{A}'$ *is a specialization of automaton* $\mathtt{A}$, *then:*

1. $\alpha \in \mathtt{execs}(\mathtt{A}') \Rightarrow \alpha|_{\mathtt{p}} \in \mathtt{execs}(\mathtt{A})$.
2. $\beta \in \mathtt{traces}(\mathtt{A}') \Rightarrow \beta \in \mathtt{traces}(\mathtt{A})$.

Proof 3.1:

1. Let α be an execution of $\mathtt{A}'$, which, by definition of execution, means that α begins in some initial state $\mathtt{t}_0$ and that every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α is a transition of $\mathtt{A}'$. By Definition 3.1, $\mathtt{t}_0|_{\mathtt{p}}$ is an initial state of $\mathtt{A}$ and, for every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α, the triple $(\mathtt{t_i}|_{\mathtt{p}}, \pi, \mathtt{t_{i+1}}|_{\mathtt{p}})$ is a transition of $\mathtt{A}$. From this it follows that the sequence obtained by replacing each state $\mathtt{t}$ in α with $\mathtt{t}|_{\mathtt{p}}$ is an execution of $\mathtt{A}$. Since this sequence is $\alpha|_{\mathtt{p}}$, we conclude that $\alpha|_{\mathtt{p}}$ is an execution of $\mathtt{A}$.
2. Follows from Part 1 and the fact that $\mathtt{sig}(\mathtt{A}') = \mathtt{sig}(\mathtt{A})$. ■

As a consequence of part 2 of Theorem 3.1, we have the following corollary:

Corollary 3.2 *If automaton* $\mathtt{A}$ *satisfies automaton* $\mathtt{S}$ *in terms of trace inclusion, then a specialization* $\mathtt{A}'$ *of automaton* $\mathtt{A}$ *also satisfies* $\mathtt{S}$ *in terms of trace inclusion.*

Moreover, given a simulation relation $\mathtt{R_p}$ from $\mathtt{A}$ to $\mathtt{S}$, the same relation is a simulation from $\mathtt{A}'$ to $\mathtt{S}$, except for the obvious projection of the states of $\mathtt{A}'$ onto the states of $\mathtt{A}$.

Corollary 3.3 *If relation* $\mathtt{R_p}$ *is a simulation from* $\mathtt{A}$ *to* $\mathtt{S}$, *and* $\mathtt{A}'$ *is a specialization of* $\mathtt{A}$, *then relation* $\mathtt{R}'_{\mathtt{p}} = \{(\mathtt{t}, \mathtt{s}) \;:\; \mathtt{t} \in \mathtt{states}(\mathtt{A}') \;\wedge\; (\mathtt{t}|_{\mathtt{p}}, \mathtt{s}) \in \mathtt{R_p}\}$ *is a simulation from* $\mathtt{A}'$ *to* $\mathtt{S}$.

Many similar inheritance constructs, such as, for example, [29, 30, 11, 4] and superposition of [34], were defined and proven to satisfy properties similar to those of Theorem 3.1 and Corollary 3.2. However, these properties are *not enough* to address the situation illustrated in Figure 1, where we are interested in reusing and extending a proof that automaton $\mathtt{A}$ satisfies automaton $\mathtt{S}$ in order to prove that a specialization $\mathtt{A}'$ of $\mathtt{A}$ satisfies a specialization $\mathtt{S}'$ of $\mathtt{S}$. Indeed, from Theorem 3.1 and Corollary 3.2, we know only that $\mathtt{traces}(\mathtt{S}') \subseteq \mathtt{traces}(\mathtt{S})$ and that $\mathtt{traces}(\mathtt{A}') \subseteq \mathtt{traces}(\mathtt{A}) \subseteq \mathtt{traces}(\mathtt{S})$; the solid arrows in Figure 1 correspond to these trace inclusions. But, we do not know whether $\mathtt{traces}(\mathtt{A}') \subseteq \mathtt{traces}(\mathtt{S}')$; this is what we would like to be able to show without having to repeat the reasoning used in showing that $\mathtt{traces}(\mathtt{A}) \subseteq \mathtt{traces}(\mathtt{S})$. In the next section, we address this question by developing a general theorem that facilitates reuse of simulation proofs at the parent level for the construction of simulation proofs at the child level. The theorem pinpoints exactly which parts of the child-level proof follow from the parent-level proof (these are the parts reused), and which do not, and therefore still need to be done in order to complete the proof.

4 Incremental Proofs

We now present the main technical contribution of this paper — a general theorem that lays the foundation for incremental proof construction. Consider the situation illustrated in Figure 1, where $\mathtt{A}'$ and $\mathtt{S}'$ are specializations of automata $\mathtt{A}$ and $\mathtt{S}$ respectively. Given a simulation relation $\mathtt{R_p}$ from $\mathtt{A}$ to $\mathtt{S}$, Theorem 4.1 below states conditions for reusing and extending $\mathtt{R_p}$ to a simulation relation $\mathtt{R_c}$ from $\mathtt{A}'$ to $\mathtt{S}'$. Relation $\mathtt{R_c}$ has to relate every initial state of $\mathtt{A}'$ to some initial state extension of $\mathtt{S}'$, and it has to satisfy a step condition similar to the one in Definition 2.1, but only involving the transition restriction relation of $\mathtt{S}'$.

Theorem 4.1 *Let automaton $\mathtt{A}'$ be a specialization of automaton $\mathtt{A}$. Let automaton $\mathtt{S}'$ be a specialization of automaton $\mathtt{S}$, such that $\mathtt{S}' = \mathtt{specialize(S)(N, N_0, TR)}$. Assume that $\mathtt{A}$ and $\mathtt{S}$ have the same external signatures and that $\mathtt{A}$ implements $\mathtt{S}$ via a simulation relation $\mathtt{R_p}$.*

A relation $\mathtt{R_c} \subseteq \mathtt{states(A')} \times \mathtt{states(S')}$, defined in terms of relation $\mathtt{R_p}$ and a new relation $\mathtt{R_n} \subseteq \mathtt{states(A')} \times \mathtt{N}$ as $\{(\mathtt{t}, \mathtt{s}) : (\mathtt{t}|_\mathtt{p}, \mathtt{s}|_\mathtt{p}) \in \mathtt{R_p} \wedge (\mathtt{t}, \mathtt{s}|_\mathtt{n}) \in \mathtt{R_n}\}$, is a simulation from $\mathtt{A}'$ to $\mathtt{S}'$ if $\mathtt{R_c}$ satisfies the following two conditions:

1. *For every $\mathtt{t} \in \mathtt{start(A')}$, there exists a state $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$ such that $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$.*
2. *If $\mathtt{t}$ is a reachable state of $\mathtt{A}'$, $\mathtt{s}$ is a reachable state of $\mathtt{S}'$ such that $\mathtt{s}|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t}|_\mathtt{p})$ and $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$, and $(\mathtt{t}, \pi, \mathtt{t}')$ is a step of $\mathtt{A}'$, then there exists a finite sequence α of alternating states and actions of $\mathtt{S}'$, beginning from $\mathtt{s}$ and ending at some state $\mathtt{s}'$, and satisfying the following conditions:*[1]

   (a) $\alpha|_\mathtt{p}$ is an execution fragment of $\mathtt{S}$.
   (b) For every step $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}})$ in α, $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}|_\mathtt{n}) \in \mathtt{TR}$.
   (c) $\mathtt{s}'|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t}'|_\mathtt{p})$.
   (d) $\mathtt{s}'|_\mathtt{n} \in \mathtt{R_n}(\mathtt{t}')$.
   (e) α has the same trace as $(\mathtt{t}, \pi, \mathtt{t}')$.

The theorem follows from Corollary 3.3 and Lemma 4.2 below. Recall that Corollary 3.3 defines a simulation relation $\mathtt{R_p'}$ from $\mathtt{A}'$ to $\mathtt{S}$ in terms of the simulation relation $\mathtt{R_p}$ from $\mathtt{A}$ to $\mathtt{S}$ (see Figure 5). The lemma considers how to construct a simulation relation $\mathtt{R_c}$ from $\mathtt{A}'$ to $\mathtt{S}'$ from the simulation relation $\mathtt{R_p'}$. This is a special case of Theorem 4.1, when $\mathtt{A}'$ is the same as $\mathtt{A}$. The statement of this lemma is almost identical to that of Theorem 4.1; the only difference is that, in Theorem 4.1, state $\mathtt{t}$ of $\mathtt{states(A')}$ is projected onto its parent's state in order to be used in the simulation relation $\mathtt{R_p}$. The lemma is stated in terms of $\mathtt{A}'$ and $\mathtt{R_p'}$ in order to match the notation in Theorem 4.1.

Lemma 4.2 *Let $\mathtt{S}$ and $\mathtt{A}'$ be automata with the same external signatures, and let relation $\mathtt{R_p'}$ be a simulation from $\mathtt{A}'$ to $\mathtt{S}$. Let $\mathtt{S}' = \mathtt{specialize(S)(N, N_0, TR)}$. A relation $\mathtt{R_c} \subseteq \mathtt{states(A')} \times \mathtt{states(S')}$, defined in terms of relation $\mathtt{R_p'}$ and a new relation $\mathtt{R_n} \subseteq \mathtt{states(A')} \times \mathtt{N}$ as $\{(\mathtt{t}, \mathtt{s}) : (\mathtt{t}, \mathtt{s}|_\mathtt{p}) \in \mathtt{R_p'} \wedge (\mathtt{t}, \mathtt{s}|_\mathtt{n}) \in \mathtt{R_n}\}$, is a simulation from $\mathtt{A}'$ to $\mathtt{S}'$ if $\mathtt{R_c}$ satisfies the following two conditions:*

[1] *Note that we do not require α to be an execution fragment of $\mathtt{S}'$.*

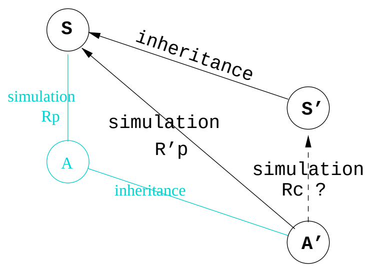

Figure 5: Intermediate step: Reusing $\mathtt{R}'_\mathtt{p}$ for building $\mathtt{R_c}$.

1. *For every* $\mathtt{t} \in \mathtt{start}(\mathtt{A}')$*, there exists a state* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n}(\mathtt{t})$ *such that* $\mathtt{s}|_\mathtt{n} \in \mathtt{N}_0$.
2. *If* $\mathtt{t}$ *is a reachable state of* $\mathtt{A}'$*,* $\mathtt{s}$ *is a reachable state of* $\mathtt{S}'$ *such that* $\mathtt{s}|_\mathtt{p} \in \mathtt{R}'_\mathtt{p}(\mathtt{t})$ *and* $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n}(\mathtt{t})$*, and* $(\mathtt{t}, \pi, \mathtt{t}')$ *is a step of* $\mathtt{A}'$*, then there exists a finite sequence* α *of alternating states and actions of* $\mathtt{S}'$*, beginning from* $\mathtt{s}$ *and ending at some state* $\mathtt{s}'$*, and satisfying the following conditions:*

   (a) $\alpha|_\mathtt{p}$ *is an execution fragment of* $\mathtt{S}$.

   (b) For every step $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}})$ *in* α*,* $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}|_\mathtt{n}) \in \mathtt{TR}$.

   (c) $\mathtt{s}'|_\mathtt{p} \in \mathtt{R}'_\mathtt{p}(\mathtt{t}')$.

   (d) $\mathtt{s}'|_\mathtt{n} \in \mathtt{R_n}(\mathtt{t}')$.

   (e) α *has the same trace as* $(\mathtt{t}, \pi, \mathtt{t}')$.

Proof 4.2: We show that $\mathtt{R_c}$ satisfies the two conditions of Definition 2.1:

1. Consider an initial state $\mathtt{t}$ of $\mathtt{A}'$. By the fact that $\mathtt{R}'_\mathtt{p}$ is a simulation, there must exist a state $\mathtt{s}|_\mathtt{p} \in \mathtt{R}'_\mathtt{p}(\mathtt{t})$ such that $\mathtt{s}|_\mathtt{p} \in \mathtt{start}(\mathtt{S})$. By condition 1 of the lemma, there must exist a state $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n}(\mathtt{t})$ such that $\mathtt{s}|_\mathtt{n} \in \mathtt{N}_0$. Consider state $\mathtt{s} = \langle \mathtt{s}|_\mathtt{p}, \mathtt{s}|_\mathtt{n} \rangle$. State $\mathtt{s}$ is in $\mathtt{R_c}(\mathtt{t})$ by definition. Also, by Definition 3.1, $\mathtt{start}(\mathtt{S}') = \mathtt{start}(\mathtt{S}) \times \mathtt{N}_0$; therefore, $\mathtt{s} = \langle \mathtt{s}|_\mathtt{p}, \mathtt{s}|_\mathtt{n} \rangle \in \mathtt{start}(\mathtt{S}) \times \mathtt{N}_0 = \mathtt{start}(\mathtt{S}')$.
2. First, notice that the definitions of state $\mathtt{s}$ and relation $\mathtt{R_c}$ imply that $\mathtt{s} \in \mathtt{R_c}(\mathtt{t})$; also, notice that conditions 2c and 2d imply that $\mathtt{s}' \in \mathtt{R_c}(\mathtt{t}')$.

   Next, we show that α is an execution fragment of $\mathtt{S}'$ with the right trace. Indeed, every step of α is consistent with $\mathtt{trans}(\mathtt{S})$ (by 2a) and is consistent with $\mathtt{TR}$ (by 2b). Therefore, by definition of $\mathtt{trans}(\mathtt{S}')$ (Definition 3.1), every step of α is consistent with $\mathtt{trans}(\mathtt{S}')$. In other words, α is an execution fragment of $\mathtt{S}'$ that starts with state in $\mathtt{R_c}(\mathtt{t})$, ends with state in $\mathtt{R_c}(\mathtt{t}')$, and has the same trace as $(\mathtt{t}, \pi, \mathtt{t}')$ (by 2e).

■

We are now ready to prove Theorem 4.1:
Proof 4.1: Theorem 4.1 follows immediately from Lemma 4.2 applied to automata $\mathtt{A}'$, $\mathtt{S}$, and $\mathtt{S}'$, with a simulation relation $\mathtt{R}'_\mathtt{p}$ from $\mathtt{A}'$ to $\mathtt{S}$ being $\{(\mathtt{t}, \mathtt{s}) \ : \ \mathtt{t} \in \mathtt{states}(\mathtt{A}') \ \wedge \ (\mathtt{t}|_\mathtt{p}, \mathtt{s}) \in \mathtt{R_p}\}$, as proved in Corollary 3.3. Each of the conditions in this theorem implies the corresponding condition in the lemma. ■

In practice, Theorem 4.1 (or Lemma 4.2) would be exploited as follows: The simulation proof between the parent automata already provides a corresponding execution fragment of the parent specification for every step of the parent algorithm. It is typically the case that the same execution fragment, padded with new state variables, corresponds to the same step at the child algorithm. Thus, conditions 2a, 2c, and 2e of Lemma 4.2 hold for this fragment. The only conditions that have to be verified are 2b, and 2d, that is, that every step of this execution fragment is consistent with the transition restriction $\mathtt{TR}$ placed on $\mathtt{S}$ by $\mathtt{S}'$ and that the values of the new state variables of $\mathtt{S}'$ in the final state of this execution are related to the post-state of the child algorithm. The verification of these two conditions may depend on some of the invariant assertions that were uncovered during the parent proof.

To exemplify how Theorem 4.1 and Lemma 4.2 would be exploited in practice, we use Lemma 4.2 to prove that `FibSeq` satisfies `AccSeq`, a specialization of `UpSeq`. Automata `UpSeq`, `FibSeq`, and `AccSeq` are simple enough to keep the example tractable, but they are arguably too simple to demonstrate the full utility of incremental proof construction. In Section 6 we describe how this framework was exploited in the design of a complex group communication service.

Example 4.1 *Recall that in Example 2.3 we presented a simulation mapping* $\mathtt{R}$ *from the states of* `FibSeq` *to the states of* `UpSeq`. *To construct a simulation mapping* $\mathtt{R}'$ *from* `FibSeq` *to* `AccSeq`, *we extend* $\mathtt{R}$ *with the following mapping* $\mathtt{R_n}$ *that maps each state* $\mathtt{t}$ *of* `FibSeq` *to the state extension* $\mathtt{s}$ *of* `AccSeq` *such that*

$$\mathtt{s.diff} = \begin{cases} \mathtt{t.m} - \mathtt{t.n} & \textit{if}\ \mathtt{t.n} \neq 0 \\ 0 & \textit{otherwise} \end{cases}$$

In order to prove that $\mathtt{R}'$ *is a simulation mapping we have to prove that it satisfies each of the conditions of Lemma 4.2.*

Condition 1 is satisfied because, if $\mathtt{t}$ *is the initial state of* `FibSeq`, $\mathtt{R_n(t).diff} = 0$ *is a valid initial value for the state extension of* `AccSeq`.

For Condition 2, the action correspondence is the same as in the simulation of `UpSeq` *by* `FibSeq`*: a step of* `AccSeq` *involving* `print(x)` *is simulated whenever* `FibSeq` *takes a step involving* `print(x)`. *Conditions 2a, 2c, and 2e are implied by the fact that* $\mathtt{R}$ *is a simulation relation from* `FibSeq` *to* `UpSeq`*; these were proven in Example 2.3. Thus, we only need to prove conditions 2b and 2d. Condition 2b requires the new precondition,* $\mathtt{x} - \mathtt{last} \geq \mathtt{diff}$, *to be satisfied in state* $\mathtt{R}'(\mathtt{t})$, *provided the parent's precondition,* $\mathtt{x} = \mathtt{n} + \mathtt{m}$, *holds in state* $\mathtt{t}$. *Condition 2d requires the* $\mathtt{R_n}$ *mapping to be preserved in the post-transition states of* `FibSeq` *and* `AccSeq`*; namely, the value of the new state variable* `diff` *in the post-transition state of* `AccSeq` *has to be the same as that of* $\mathtt{R_n(t').diff}$. *Proving that these two conditions are satisfied involves reasoning only about how* `AccSeq` *specializes* `UpSeq`.

We now prove that conditions 2b and 2d hold. Consider a step $(\mathtt{t}, \mathtt{print(x)}, \mathtt{t}')$ *of* `FibSeq`*; it implies that* $\mathtt{x} = \mathtt{t.n} + \mathtt{t.m}$, *and that* $\mathtt{t}'.\mathtt{n} = \mathtt{t.m}$ *and* $\mathtt{t}'.\mathtt{m} = \mathtt{t.n} + \mathtt{t.m}$.

- *Condition 2b: We have to show that the corresponding* `print(x)` *step of* `AccSeq` *is enabled in state* $\mathtt{R}'(\mathtt{t})$, *that is, that* $\mathtt{x} - \mathtt{R}'(\mathtt{t}).\mathtt{last} \geq \mathtt{R}'(\mathtt{t}).\mathtt{diff}$. *By using the simulation mapping, we derive:* $\mathtt{x} - \mathtt{R}'(\mathtt{t}).\mathtt{last} = \mathtt{x} - \mathtt{R(t).last} = \mathtt{x} - \mathtt{t.m} = \mathtt{t.n} + \mathtt{t.m} - \mathtt{t.m}$

$= \texttt{t.n}$. *If* $\texttt{t.n} = 0$ *(as in the initial state of* FibSeq*), then, by definition of* $\texttt{R}'$ *and* $\texttt{R}_\texttt{n}$, $\texttt{R}'(\texttt{t}).\texttt{diff} = \texttt{R}_\texttt{n}(\texttt{t}).\texttt{diff} = 0$*, and we are done. Otherwise, if* $\texttt{t.n} \neq 0$*, then* $\texttt{R}'(\texttt{t}).\texttt{diff} = \texttt{R}_\texttt{n}(\texttt{t}).\texttt{diff} = \texttt{t.m} - \texttt{t.n}$*, and it remains to show that* $\texttt{t.n} \geq \texttt{t.m} - \texttt{t.n}$*. Invariant 4.2 below establishes this fact by relying on the following auxiliary invariant:*

Invariant 4.1 *In every reachable state* t *of* FibSeq, $\texttt{t.m} \geq \texttt{t.n}$.
Proof: *The proposition is true in the initial state* $\texttt{t}_0$*, since* $\texttt{t}_0\texttt{.n} = 0$ *and* $\texttt{t}_0\texttt{.m} = 1$*. The proposition is true in state* $\texttt{t}'$*, after a step* $(\texttt{t}, \texttt{print}(\texttt{x}), \texttt{t}')$ *of* FibSeq*, since* $\texttt{t.n} \geq 0$ *(Invariant 2.1), and hence* $\texttt{t}'\texttt{.m} = \texttt{t.n} + \texttt{t.m} \geq \texttt{t.m} = \texttt{t}'\texttt{.n}$*.* ■

Invariant 4.2 *In every reachable state* t *of* FibSeq, $\texttt{t.n} \geq \texttt{t.m} - \texttt{t.n}$*, if* $\texttt{t.n} \neq 0$*.*
Proof: *The proposition is vacuously true in the initial state* $\texttt{t}_0$*, since* $\texttt{t}_0\texttt{.n} = 0$*. The proposition is true in state* $\texttt{t}'$*, after a step* $(\texttt{t}, \texttt{print}(\texttt{x}), \texttt{t}')$ *of* FibSeq*, since* $\texttt{t.m} \geq \texttt{t.n}$ *(Invariant 4.1), and therefore* $\texttt{t}'\texttt{.n} = \texttt{t.m} \geq \texttt{t.n} = \texttt{t}'\texttt{.m} - \texttt{t.m} = \texttt{t}'\texttt{.m} - \texttt{t}'\texttt{.n}$*.* ■

- *Condition 2d: According to the code, the post-transition value of* diff *is* $\texttt{x} - \texttt{R}'(\texttt{t}).\texttt{last} = \texttt{t.n} = \texttt{t}'\texttt{.m} - \texttt{t.m} = \texttt{t}'\texttt{.m} - \texttt{t}'\texttt{.n}$*. If* $\texttt{t}'\texttt{.n} \neq 0$*, then* $\texttt{t}'\texttt{.m} - \texttt{t}'\texttt{.n} = \texttt{R}_\texttt{n}(\texttt{t}').\texttt{diff}$*, and we are done. Otherwise, if* $\texttt{t}'\texttt{.n} = 0$, $\texttt{R}_\texttt{n}(\texttt{t}').\texttt{diff} = 0$ *by definition, and the post-transition value of* diff *is also* 0*, since* $0 = \texttt{t}'\texttt{.n} = \texttt{t.m} \geq \texttt{t.n} \geq 0$ *(Invariants 2.1 and 4.1).*

Notice that, in verifying conditions 2b and 2d in Example 4.1, we relied on Invariant 2.1, which was stated and proven during the simulation proof from FibSeq to UpSeq. In general, knowing the invariant assertions that have been uncovered during the parent's proof can be helpful in extending that proof to the children.

5 Subclassing for Extension

In this section, we extend the theory of incremental modeling and proof construction to a new modification construct, called *specialized extension*; the construct is formulated in Definition 5.2 and the extended proof-reuse theorem appears as Theorem 5.4. This construct corresponds to the *subclassing for extension* form of inheritance [7], which is similar to specialization in that a child cannot override its parent's behavior, but it is more powerful than specialization in that a child can introduce new types of behavior through new actions, nonexistent in the parent.

We define a specialized extension of an automaton by first extending the parent automaton with new actions using a new construct, called *signature extension*, and then applying specialization of Section 3. The new actions introduced by signature extension are enabled in every state and do not modify the state; the subsequent specialization operation gives meaning to these new actions by restricting transitions involving the new actions, and, possibly, those involving parent's actions as well. The resulting automaton can interact with its environment through both the parent's actions and the new ones. Because new actions (even after being specialized) do not affect the parent's state, any trace of the child is indistinguishable from a trace of the parent when new actions are projected out from the trace.[2]

The signature extension construct, formulated in Definition 5.1, creates a new automaton by adding new actions to an existing automaton. The new automaton has an extended signature, but the same states and start states as the original automaton; the new state-transition relation is the same as the one in the original automaton, except that it includes additional transitions that relate every state to itself via new actions (i.e., new actions are enabled in every state, but do not modify the state); such transitions are called "stuttering" steps by Lamport [28].

In addition, the signature extension construct allows the new automaton to rename some or all of the original automaton's actions. The renaming is specified by a *signature-mapping* function that maps actions in the new signature to their counterparts in the parent signature. The function is allowed to be many-to-one, which means that the same action of the parent may be renamed into several actions of the child; this is useful because it allows a child to add new parameters to its parent's actions, and because instances of the same parent's action can be specialized differently under different names. The signature-mapping is onto, that is, every parent action has at least one corresponding action at the child. The function is defined only for actions inherited from the parent (renamed or not); it is undefined for new actions introduced by the signature extension. If π is such a new action and $\mathtt{f}$ is a signature-mapping, we write $\mathtt{f}(\pi) = \perp$ to denote the fact that π is not in the domain of definition of $\mathtt{f}$; $\perp$ is a assumed to be different from any action name.

Definition 5.1 (Signature Extension) *Let* $\mathtt{A}$ *be an automaton, and* $\mathtt{X}$ *be some signature.*

Let $\mathtt{f}$ *be a partial function, called a* signature-mapping, *from* $\mathtt{X}$ *to* $\mathtt{sig(A)}$ *such that* $\mathtt{f}$ *is onto and preserves the classification of actions as "input", "output", and "internal"; the latter means that, if* $\mathtt{f}(\pi)$ *is defined, it is of the same classification as* π.[3]

Then, $\mathtt{extend(A)(X, f)}$ *is defined to be the following automaton* $\mathtt{A'}$*:*

- $\mathtt{sig(A')} = \mathtt{X}$,
- $\mathtt{states(A')} = \mathtt{states(A)}$,
- $\mathtt{start(A')} = \mathtt{start(A)}$, *and*
- $\mathtt{trans(A')} = \{(\mathtt{t}, \pi, \mathtt{t'}) \in \mathtt{states(A')} \times \mathtt{sig(A')} \times \mathtt{states(A')}$:
  $((\mathtt{f}(\pi) = \perp) \wedge (\mathtt{t} = \mathtt{t'})) \vee ((\mathtt{f}(\pi) \in \mathtt{sig(A)}) \wedge ((\mathtt{t}, \mathtt{f}(\pi), \mathtt{t'}) \in \mathtt{trans(A)}))\}$.

We say that $\mathtt{A'}$ is the *signature extension* of $\mathtt{A}$ with signature-mapping $\mathtt{f}$ if $\mathtt{A'}$ is such that $\mathtt{A'} = \mathtt{extend(A)(sig(A'), f)}$ for some signature-mapping $\mathtt{f}$ from $\mathtt{sig(A')}$ to $\mathtt{sig(A)}$.

Having defined the signature extension construct, we now combine it with specialization to yield specialized extensions of automata.

[2]Notice that this is stronger than behavioral subtyping of Liskov and Wing [30, 29], in which a trace of a child is required to be indistinguishable from a trace of its parent only when the trace does not contain actions introduced by the child (see Section 8).

[3]Signature-mapping is similar to *strong correspondence* of [41].

Definition 5.2 (Specialized Extension) *Automaton* `A'` *is called a* specialized extension *of an automaton* `A` *if* `A'` *is a specialization of a signature extension of* `A`*.*

In precondition-effect notation, we express a specialized extension `A'` of an automaton `A` by writing "`A'` MODIFIES `A`" and then specifying the signature extension and the specialization parts of `A'`. The signature extension part contains the new actions labeled with a keyword new, and the renamed actions labeled with their original names in `sig(A)`, according to the signature-mapping; for example, if the signature mapping maps π of `A'` to σ of `A`, we write "π modifies σ". We omit specifying the actions of `sig(A')` that are inherited from `A` without renaming. The specialization part contains the state extension and the transition restriction specifications, as described in Section 3 on page 9.

We now exemplify how the signature extension construct can be used in conjunction with the specialization construct to create specialized extensions.

Example 5.1 *Figure 6 presents automaton* `FibSeq+` *that modifies automaton* `FibSeq` *to print each element of the Fibonacci sequence together with its sequence number. The signature-mapping specified by the Signature Extension clause maps actions* `print(i, x)` *where* `i` $\in$ `Integer` *to actions* `print(x)` *of* `FibSeq`*. Thus, for example, actions* `print(8, 43)` *and* `print(23, 43)` *of* `FibSeq+` *are among those actions mapped to the* `print(43)` *action of* `FibSeq`*. Then, the specialization construct adds a new state variable,* `last_i`*, that keeps track of the sequence number of the last Fibonacci element printed; it also adds a new precondition and a new effect to the* `print(i, x)` *action to maintain* `i` *and* `last_i` *properly.*

```
AUTOMATON FibSeq+  MODIFIES FibSeq

Signature Extension: Output print(i, x), i ∈  Integer modifies FibSeq.print(x)

New State:   last_i ∈ Integer, initially 0

Transition Restriction:
               OUTPUT print(i, x)
               new pre: i = last_i + 1
               new eff: last_i ← i
```

Figure 6: Automaton `FibSeq+` specifying enumerated Fibonacci sequences.

Notice that any execution α of `FibSeq+` is an execution of `FibSeq` when the newly added state variable, `last_i`, is projected out from every state in α and when every action in α is renamed according to the specified signature-mapping. Theorem 5.2 below formalizes this property in general. It follows from Theorem 3.1, which is a similar execution-inclusion property of specialization. This is because, modulo the signature-mapping, a signature extension of an automaton and the automaton itself have exactly the same executions and traces; we prove this result in Lemma 5.1 below.

Notation 5.3 *Let* `A'` *be a signature extension of* `A` *with a signature-mapping* `f`*.*

If α is a sequence of alternating states and actions of $\mathtt{A'}$, then $\mathtt{f}(\alpha)$ denotes the sequence obtained by replacing each action π in α with $\mathtt{f}(\pi)$, and then collapsing every triple of the form $(\mathtt{t}, \perp, \mathtt{t})$ to $\mathtt{t}$. Triples of the form $(\mathtt{t}, \perp, \mathtt{t'})$ where $\mathtt{t'} \neq \mathtt{t}$ are not collapsed; such triples are possible because α is not necessarily an execution sequence of $\mathtt{A'}$.

Likewise, if β is a sequence of external actions of $\mathtt{A'}$, then $\mathtt{f}(\beta)$ denotes a sequence obtained by replacing each action π in β with $\mathtt{f}(\pi)$, and then removing all the occurrences of $\perp$.

Lemma 5.1 *Let automaton $\mathtt{A'}$ be a signature extension of $\mathtt{A}$ with a signature-mapping $\mathtt{f}$.*

Let α be a sequence of alternating states and actions of $\mathtt{A'}$ and let β be a sequence of external actions of $\mathtt{A'}$. Then:

1. $\alpha \in \mathtt{execs(A')} \Leftrightarrow \mathtt{f}(\alpha) \in \mathtt{execs(A)}$.
2. $\beta \in \mathtt{traces(A')} \Leftrightarrow \mathtt{f}(\beta) \in \mathtt{traces(A)}$.

Proof 5.1: The proof follows from Definition 5.1 and Notation 5.3.

1. $\Rightarrow$: Let α be an execution of $\mathtt{A'}$. By definition of execution, α begins in some initial state $\mathtt{t_0}$, and every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α is a transition of $\mathtt{A'}$. From this and Definition 5.1, $\mathtt{t_0}$ is an initial state of $\mathtt{A}$, and, for every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α, either $(\mathtt{t_i}, \mathtt{f}(\pi), \mathtt{t_{i+1}})$ is a step of $\mathtt{A}$ when $\mathtt{f}(\pi) \in \mathtt{sig(A)}$, or $\mathtt{t_i} = \mathtt{t_{i+1}}$ when $\mathtt{f}(\pi) = \perp$.

   Therefore, by definition of execution, the sequence obtained by replacing every step $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ in α with either $(\mathtt{t_i}, \mathtt{f}(\pi), \mathtt{t_{i+1}})$ when $\mathtt{f}(\pi) \in \mathtt{sig(A)}$, or $\mathtt{t_i}$ when $\mathtt{f}(\pi) = \perp$ is an execution of $\mathtt{A}$. Since this sequence is $\mathtt{f}(\alpha)$, we conclude that $\mathtt{f}(\alpha) \in \mathtt{execs(A)}$.

   $\Leftarrow$: Let α be a sequence of alternating states and actions of $\mathtt{A'}$ such that $\mathtt{f}(\alpha) \in \mathtt{execs(A)}$. This means that α begins with some initial state $\mathtt{t_0}$ of $\mathtt{A}$, and that, for every triple $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ of elements in α, either $(\mathtt{t_i}, \mathtt{f}(\pi), \mathtt{t_{i+1}})$ is a step of $\mathtt{A}$ when $\mathtt{f}(\pi) \in \mathtt{sig(A)}$, or $\mathtt{t_i} = \mathtt{t_{i+1}}$ when $\mathtt{f}(\pi) = \perp$. From this assumption and Definition 5.1, it follows that $\mathtt{t_0}$ is an initial state of $\mathtt{A'}$ and that every triple $(\mathtt{t_i}, \pi, \mathtt{t_{i+1}})$ of elements in α is a transition of $\mathtt{A'}$. Thus, $\alpha \in \mathtt{execs(A')}$.

2. Follows from part 1 and the fact that `f` preserves the classification of actions as "input", "output", and "internal". ∎

Theorem 5.2 *If $\mathtt{A'}$ is a specialized extension of $\mathtt{A}$ with a signature-mapping $\mathtt{f}$, then*

1. $\alpha \in \mathtt{execs(A')} \Rightarrow \mathtt{f}(\alpha|_{\mathtt{p}}) \in \mathtt{execs(A)}$.
2. $\beta \in \mathtt{traces(A')} \Rightarrow \mathtt{f}(\beta) \in \mathtt{traces(A)}$.

Proof 5.2: Follows immediately from Theorem 3.1 and Lemma 5.1 ∎

Since signature extension does not modify the original automata beyond simple renaming of actions, we would expect it to have minimal effect on the proof-reuse theorems (Theorem 4.1 and Lemma 4.2) of Section 4 when those theorems are used in verifying specialized extensions of automata. We prove this intuition correct in Theorem 5.4 below; this theorem is an adaptation of Theorem 4.1 for the case when child automata are specialized extensions of their parents. The theorem follows from Theorem 4.1 and the following lemma, which establishes that a simulation relation between two automata is preserved when these automata are signature-extended:

Lemma 5.3 *Let $\mathtt{A}'$ be the signature extension of $\mathtt{A}$ with a signature-mapping $\mathtt{f}$. Let $\mathtt{S}'$ be the signature extension of $\mathtt{S}$ with a signature-mapping $\mathtt{g}$. Assume that $\mathtt{A}$ has the same external signature as $\mathtt{S}$ and that there is a simulation relation $\mathtt{R}$ from $\mathtt{A}$ to $\mathtt{S}$. Assume further that $\mathtt{A}'$ has the same external signature as $\mathtt{S}'$, and that, for every external action $\pi \in \mathtt{sig(A')}$, $\mathtt{g}(\pi) = \mathtt{f}(\pi)$. Then, $\mathtt{R}$ is a simulation relation from $\mathtt{A}'$ to $\mathtt{S}'$.*

Proof 5.3: Follows straightforwardly from Definitions 2.1 and 5.1. ■

The only difference between the statements of Theorem 5.4 below and Theorem 4.1 is that here, whenever child's actions are used in the context of the parent automaton (as in Condition 2a), they are translated via the signature-mapping to the corresponding actions of the parent.

Theorem 5.4 *Let automaton $\mathtt{A}'$ be a specialized extension of $\mathtt{A}$ with a signature-mapping $\mathtt{f}$. Let automaton $\mathtt{S}'$ be a specialized extension of $\mathtt{S}$ with a signature-mapping $\mathtt{g}$, such that $\mathtt{S}' = \mathtt{specialize(extend(S)(G,g))(N,N_0,TR)}$. Assume that $\mathtt{A}$ and $\mathtt{S}$ have the same external signatures and that $\mathtt{A}$ implements $\mathtt{S}$ via a simulation relation $\mathtt{R_p}$. Assume further that $\mathtt{A}'$ and $\mathtt{S}'$ have the same external signatures, and that, for every external action $\pi \in \mathtt{A}'$, $\mathtt{g}(\pi) = \mathtt{f}(\pi)$.*

A relation $\mathtt{R_c} \subseteq \mathtt{states(A')} \times \mathtt{states(S')}$, defined in terms of relation $\mathtt{R_p}$ and a new relation $\mathtt{R_n} \subseteq \mathtt{states(A')} \times \mathtt{N}$ as $\{(\mathtt{t},\mathtt{s}) \ : \ (\mathtt{t}|_\mathtt{p}, \mathtt{s}|_\mathtt{p}) \in \mathtt{R_p} \wedge (\mathtt{t}, \mathtt{s}|_\mathtt{n}) \in \mathtt{R_n}\}$, is a simulation from $\mathtt{A}'$ to $\mathtt{S}'$ if $\mathtt{R_c}$ satisfies the following two conditions:

1. *For every $\mathtt{t} \in \mathtt{start(A')}$, there exists a state $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$ such that $\mathtt{s}|_\mathtt{n} \in \mathtt{N_0}$.*
2. *If $\mathtt{t}$ is a reachable state of $\mathtt{A}'$, $\mathtt{s}$ is a reachable state of $\mathtt{S}'$ such that $\mathtt{s}|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t}|_\mathtt{p})$ and $\mathtt{s}|_\mathtt{n} \in \mathtt{R_n(t)}$, and $(\mathtt{t}, \pi, \mathtt{t}')$ is a step of $\mathtt{A}'$, then there exists a finite sequence α of alternating states and actions of $\mathtt{S}'$, beginning from $\mathtt{s}$ and ending at some state $\mathtt{s}'$, and satisfying the following conditions:*

   (a) $\mathtt{g}(\alpha|_\mathtt{p})$ *is an execution fragment of* $\mathtt{S}$.

   (b) For every step $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}})$ *in* α, $(\mathtt{s_i}, \sigma, \mathtt{s_{i+1}}|_\mathtt{n}) \in \mathtt{TR}$.

   (c) $\mathtt{s}'|_\mathtt{p} \in \mathtt{R_p}(\mathtt{t}'|_\mathtt{p})$.

   (d) $\mathtt{s}'|_\mathtt{n} \in \mathtt{R_n}(\mathtt{t}')$.

   (e) α *has the same trace as* $(\mathtt{t}, \pi, \mathtt{t}')$.

Proof 5.4: Follows straightforwardly as a corollary from Theorem 4.1 and Lemma 5.3. ■

Theorem 5.4 can be used in practice in the same way as Theorem 4.1 and Lemma 4.2 (see the discussion after the proof of Theorem 4.1 on page 12): Transitions involving new actions introduced by signature extension are defined entirely by the specialization code and, therefore, involve reasoning about this code alone. Transitions involving parent's actions, which are possibly renamed by the child, depend on the code of both the parent and the child. Even when actions are renamed, the task of proving that the simulation relation holds for such transitions typically allows one to rely on the simulation proof of the parent automata to deduce conditions 2a, 2c, and 2e, and requires verification of conditions 1, 2b, and 2d only.

6 Practical Experience With Incremental Modeling

The technique presented in this paper has evolved as part of our experience designing and modeling a complex group communication service [25]. In this section we describe our experience in that project, and how the framework presented in this paper was exploited. We use the example to illustrate circumstances under which the inheritance-based technique of this paper can be useful. In the next section we describe an interesting modeling methodology that has evolved with our experience in that project.

6.1 Group communication: background

Group communication services [6, 9] are powerful middleware systems that facilitate the development of fault-tolerant distributed applications. These services provide a notion of *group abstraction*, which allows application processes to easily organize themselves into multicast groups. Application processes can communicate with the members of a group by addressing messages to the group.

Group communication systems typically provide *reliable multicast* and *group membership* services. The task of the membership service is to maintain a listing of the currently active and connected processes and to deliver this information to the application whenever it changes. The output of the membership service is called a *view*. The reliable multicast services deliver messages to the current view members.

Group communication systems are complex software systems, and their behavior descriptions are correspondingly intricate. Such intricate behavior is often described as a collection of properties that the service guarantees (for a survey of such properties, see [9]).

6.2 Incremental modeling of group communication

In [25] we presented a formal design for a novel group communication service targeted for wide-area networks. The project included a specification of the service semantics, a model of the implementation, and an assertional correctness proof showing that the model satisfies the specification. The implementation used two auxiliary services: group membership and reliable multicast. We gave high-level abstract models of the behavior of these two services as I/O automata. We gave a low-level I/O automaton modeling the algorithm executed by the end-points of the service in different locations. The model of the implementation was then a *composition* of a collection of end-point automata (one for each end-point running the service) with the two high-level auxiliary service automata. The proof exhibited a simulation relation from the implementation model to the specification, which was also given as an I/O automaton.

The new algorithm run by the end-points of the service has been implemented in C++, using roughly 9,000 lines of code [40], including code for thread and socket maintenance, auxiliary classes for data structures maintenance, header files, in-program documentation, etc. The auxiliary membership service [27] was developed by another development team using roughly 20,000 lines of C++ code, and the reliable multicast service was implemented by a third team [3] using roughly 4,000 lines of C++ code. The I/O automaton model of the end-point algorithm required a total of approximately 120 lines of I/O automaton code, modeling fifteen different actions and using approximately ten data structures[4].

Modeling and validating a system of this scale and intricacy was a major challenge. Although formal approaches were previously used to specify group communication systems and to verify their applications, (see [8, 12, 21]), algorithms implementing the actual systems were not previously formally modeled or assertionally verified.

In order to manage the complexity of the project, we found a need to employ an object-oriented approach that would allow for reuse of models and proofs. Therefore, in [25], we used the I/O automaton formalism enriched with the inheritance-based incremental modification constructs presented in this paper to specify the safety properties of our service and to model the algorithm. We then exploited the proof reuse theorem when verifying the algorithm.

We specified, modeled, and verified our service in four steps; each step dealt with a certain group communication property. These four properties are typically defined using four separate logic formulas, for example, in [9]. Therefore, by specifying the properties incrementally, we have made it easier to relate our abstract specification automaton to existing group communication specifications. It was also important to model the algorithms implementing each of these properties one step at a time to reduce the complexity of the design and verification and to make it clear which algorithm implements which property.

We started with a simple service, FIFO, that provides reliable FIFO multicast within group membership views. The specification of the FIFO service took about 15 lines of I/O automaton code and consisted of three parameterized actions and three state variables, some of which were two-dimensional arrays. We modeled the end-point algorithm using roughly 50 lines of I/O automaton code; the code included eleven parameterized actions and seven state variables, some of which were arrays. The verification part presented a simulation proof showing that the composition of all the FIFO end-point automata and the high-level automata specifying the auxiliary services implements the FIFO specification. The proof took about five pages, included seven major invariants, and used the technique of *history variables* [2].

As a second step, we used specialized extension to modify the FIFO specification and algorithm to include an additional property, called VS. This property synchronizes view delivery and message delivery events in an execution. It requires that end-points that move together from one view to another (i.e., remain connected) deliver the same set of messages in the former view.

The extension of the FIFO specification introduced a new internal action and a new array variable to specify the appropriate synchronizations of view delivery and message delivery events. The extension then constrained the view delivery actions to occur exactly at the times when the specified synchronizations held; the constraint was expressed in terms of both parent and new variables. The extension of the specification took about ten lines of code; it relied on and reused the parent specification of how messages, views, and common data structures are handled.

[4]I/O automaton code is rather compact and therefore an I/O automaton model of a system is generally much shorter than the actual C++ code of the system. This is due to the fact that I/O code is at a higher level of abstraction, it does not include code for scheduling of actions, maintenance of threads, sockets, or data structures, garbage collection, header files, etc.

The extension of the FIFO end-point algorithm introduced a distributed synchronization protocol enforcing the VS property. The protocol involved four new actions and four new variables, some of which were arrays. It relied on the parent algorithm handling common events and data structures, such as message buffers and indices. In addition, the extension of the end-point algorithm modified four of the parent's actions. In particular, it constrained the view delivery and message delivery actions to respect the computed synchronizations. The constraints were expressed in terms of both parent and new variables. The verification exploited Theorem 5.4. The simulation proof focused solely on the VS property and took about two pages; no arguments from the five page parent-level proof needed to be repeated.

As a third step, we enriched our service with an additional property, called TS, which augments each view delivery with special information called a *transitional set* [9]. We specified this property using a stand-alone automaton, (i.e., without using inheritance), using about fifteen lines of code, two parameterized actions, and two array variables. The VS end-point algorithm was already computing the transitional set information as a by-product of implementing the VS property. We used the signature extension construct to modify the signature of the view delivery action to include the transitional set as an additional parameter. We then exhibited a simulation proof, showing that the modified algorithm satisfies the TS specification. This proof focused solely on the TS property, it took two and a half pages, included three major invariants, and used the technique of *prophecy variables* [2].

Finally, we used specialized extension to modify the VS specification and algorithm to include the fourth property, called SELF. This property requires that application processes receive their own messages before moving to the next view. SELF is another example of a synchronization property, which restricts possible action interleaving. The extension of the specification added a single constraint to the parent's view delivery action; the constraint was in terms of the parent's variables. The extension of the end-point algorithm was about fifteen lines of code and involved a synchronization with the end-point's client. Again, we exploited Theorem 5.4 in verifying that the final algorithm satisfies the final specification. The final step of the simulation proof focused solely on the SELF property; it took two and a half pages and included three major invariants.

Group communication systems are particularly amenable to incremental modeling and verification using our formalism because such systems involve a number of separate properties, each constraining or synchronizing the deliveries of messages and views. Given an algorithm (a specification) for one such property, an algorithm (a specification) that adds a second property enforces additional synchronization constraints. The child can reuse the handling of common events and data structures by the parent and introduce only the machinery required to provide the new property; the machinery can rely on both new and parent data structures. In order to provide the property, the child can establish the required synchronization, for example, using new actions, and then enforce it by adding new preconditions to the common actions. For example in [25], the algorithm that implemented the VS property established the synchronization by introducing a new protocol and then enforced the synchronization by requiring the protocol to complete before the view and message delivery events could execute. In summary, we believe that our inheritance-based formalism is particularly useful for modeling and verifying systems whose specifications consist of a number of different properties that constrain the same actions.

6.3 The benefits of incremental modeling and verification

Using the inheritance-based technique, we were able to present a complex algorithm step by step. This way, it was easy to see which part of the algorithm corresponds to which property of the specification. Correspondingly, the proof was broken up into pieces of manageable size. Moreover, each piece of the proof was focused on proving a specific property. This piece only needed to consider the part of the code that implements that property; discussing other pieces of the code would have distracted attention from what is being proven.

Previous projects that modeled large-scale complex systems using I/O automata relied on *composition*; complex algorithms were expressed by multiple manageable parts that jointly compose the algorithm (see, for example, [21]). In the context of our project, however, composition could not have been used in lieu of inheritance, for two reasons: Firstly, composition does not allow different components to share the same data structures. In contrast, all the parts of our algorithm share common data structures such as message buffers. Using composition, we would have had to duplicate these data structures as well as the book-keeping logic associated with them. This would make the algorithm models more cumbersome.

The second and more important reason is that composition does not allow for proof reuse, since it does not guarantee that one component does not violate the guarantees of the other. Consider our project, for example. Had we composed a FIFO multicast service that meets the FIFO specification with a VS service that synchronizes messages with views, we would have had no guarantee that the combined service preserves the FIFO order. In order to prove that the composition indeed satisfies the FIFO specification, we would have to prove (1) that the FIFO service orders messages in this order; and (2) that the VS service does not change message order in a way that would violate the property. When introducing a third component, (for example, SELF), we would, once again, have to prove that the new component does not violate the FIFO order. This repetition of reasoning is precisely what our inheritance-based technique allows us to avoid.

7 Discussion of Modeling Methodology

Our notion of inheritance allows a child to see the parent's internal variables, but not to write to them. In this respect, the parent's internal variables and actions can be seen as *protected* variables, but with additional restrictions. Specifically, specialization does not allow children to change state variables of their parents.

In some situations, however, one may see a need for a child to modify a parent's variable. We have encountered such situations when we modeled the algorithms in [25], as described in the previous section. We dealt with this case by introducing a certain level of non-determinism at the parent, thereby allowing the child to resolve (specialize) this nondeterminism later.

For example, the algorithm that implemented the second specification described above sometimes needed to forward messages to other processes, although such forwarding was not needed at the parent. The forwarded messages would have to be stored at the same buffers as other messages. However, these message buffers were variables of the parent, so the child was not allowed to modify them. We solved this difficulty by changing the parent automaton to have a forwarding action which forwards arbitrary messages to other processes. The parent stores incoming forwarded messages in the appropriate message buffers, in a manner that preserves the coherence of its data structures. The child then sets the policy for restricting the arbitrary message forwarding according to its algorithm. Using this methodology allowed us to benefit from proof reuse, without complicating the proofs.

We liken this methodology to the use of *abstract methods* or *pure virtual methods* in object-oriented methodology, since the non-determinism is left at the parent as a "hook" for prospective children to specify any forwarding policy they might need. Thus, the parent specifies the policy and the integrity constraints for modifying its variables. The actual mechanism is implemented by the child while abiding to the parent's constraints; the child of course can also refine the policy and the integrity constraints.

8 Related work

The works that most closely relate to ours are those of Soundarajan and Fridella [37, 38] and Stata and Guttag [39]. Unlike our formalisms, both of these works are restricted to the context of sequential programming and do not encompass reactive components.

Like us, Soundarajan and Fridella [37, 38] have recognized that incremental reasoning is important in exploiting the full potential of inheritance. They present a specification notation and a verification procedure geared towards such incremental reasoning. However, they consider a more general type of inheritance — one that allows a child to override behavior of the parent. As a result, the proof-reuse result they obtain is much weaker and less structured than ours. In particular, reasoning reuse applies only when the simulation function (abstraction function, in their case) between child automata is identical to that between parent automata, and only to those actions that are inherited from the parent without any modification. In contrast, our framework applies to all types of actions, including those which are modified by the child.

Stata and Guttag [39] have also recognized the need for proof-reuse in a manner similar to that suggested in this paper. They suggest a framework for defining programming guidelines and supplement this framework with informal rules that may be used to facilitate reasoning about correctness of a subclass given the correctness of the superclass is known. However, they only addressed informal reasoning and did not provide the mathematical foundation for formal proofs.

Numerous other research projects, for example [1, 5, 10, 11, 20, 24, 29, 30, 35, 4, 34, 15], have dealt formally with inheritance and its semantics. In particular, many projects, such as [29, 30, 11, 4, 34], focus on defining inheritance constructs in ways that either automatically imply or simplify the task of proving that a child behaves indistinguishably from its parent, in other words, that the child satisfies its parent's specification. However, no other work that we are aware of allows for reuse of a parent-level simulation proof when showing that a child satisfies (simulates) *its own specification.*

To be fair, we note that it is not immediately obvious how to adapt our incremental verification ideas to the notions of inheritance in object-oriented programming languages. The denotational semantics of inheritance in these languages is more complex than what we consider in this paper; for example, it includes recursion. However, we also feel that the essence of our approach is general enough to be applicable to other state-transition formal models, such as TLA [28], UNITY [34], and Process Algebra [22, 33].

We note also that, while our definition of subclassing for extension is similar to behavioral subtyping of Liskov and Wing [29, 30], it is not identical: Behavioral subtyping requires only that a child behave indistinguishably from its parent when the child is used in the context of the parent, that is, when the execution of the child contains only the parent's actions, and none of the actions introduced by the child. Subclassing for extension enforces a stronger property: any trace (execution) of the child, even one that has actions introduced by the child, is indistinguishable from a parent's trace when all such new actions (and new state variables) are projected out.

9 Conclusions

In this paper, we have presented an inheritance-based formalism for modeling and verifying systems incrementally.

The formalism defines two inheritance constructs that can be used to model a modified version of an abstract model of a system by specifying how the modification is different from the original. Using these constructs, one can model (specify) a complex system incrementally, by starting from a basic model (specification) and then, at each step, adding support for new properties of the system.

For simplicity, the paper has described the formalism in terms of two levels of abstraction: "specification" and "algorithm"; but in general, the formalism is complementary to the technique of using successive refinement. It can be applied for modeling systems at any relevant level of abstraction, from the lowest level corresponding to software code, to the highest one corresponding to the most abstract system specification.

A distinguishing feature of our formalism is its support for incremental verification, which compliments incremental modeling. The formalism provides fundamental theorems (4.1 and 5.4) that state formally how a simulation proof of one abstract model of a system satisfying another can be reused and extended to a simulation proof for the modified versions of these models. This allows one, not only to model and specify a complex system incrementally, but also to verify incrementally that the model satisfies its specification.

The formalism, and in particular its incremental verification component, was motivated by and refined during a project designing and modeling a complex middleware system [25]. The ability to model and verify the system incrementally was critical in making the project tractable and in making it clear which part of the algorithm implemented which property. As Section 6.3 explains, standard compositional techniques would have not been sufficient.

As explained in Section 6 on page 21, the inheritance-based formalism was particularly useful in the context of that project because the modeled middleware system involved a number of separate properties, each constraining or synchronizing common events. We believe that the formalism would be useful for modeling and verifying other systems that include different properties constraining the same actions.

The formalism described in this paper has been presented using the I/O automaton model — the same model that we used to model the complex middleware system. The I/O automaton model has been used extensively for modeling and reasoning about complex distributed systems and has been developed into a programming and modeling language, called *IOA* [13, 14]. As one of our future projects, we plan to facilitate the incorporation of our inheritance-based approach into the IOA tool-set, thereby enriching its modeling and reasoning facilities.

The I/O automaton model has been a convenient model in which to express our formalism. The essence of the approach, however, is general enough to be applicable to other state-transition formal models, such as TLA [28], UNITY [34], and Process Algebra [22, 33], or, in other words, to any formal model that supports simulation proofs. One interesting direction for future research is to enrich the standard formal modeling languages with a version of our formalism.

The formalism presented in this paper allows modeling of systems using two standard and important types of inheritance: specialization and subclassing for extension. In our future work, we are planning to expand the formalism, including its incremental verification aspect, to support other types of inheritance.

Of particular importance is a construct that would allow modifications that override a system's behavior. In general, one would expect little, if any, proof reuse possible for such a construct, since modifications done to a system may invalidate whatever reasoning has been done about it. Nevertheless, useful approaches to circumventing this impasse could rely on limiting the types of the modifications allowed by the construct and on requiring the modifications to preserve certain invariants.

The formalism presented here is an important step toward scalable and cost-effective formal methods and toward practical software design methodologies that, in addition to facilitating reuse of code, also facilitate reuse of reasoning. In general, any extensions to the formalism that we make in the future will be motivated and guided by our work on designing and modeling complex distributed systems. This approach will ensure that, like the formalism presented in this paper, these extensions will have important, practical implications.

Acknowledgments

We thank Paul Attie, Alan Fekete, Steve Garland, Victor Luchangco, Jens Palsberg, John Penix, and the anonymous referees for their helpful comments and suggestions.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[3] T. Anker, G. Chockler, I. Shnaiderman, and D. Dolev. The design of Xpand: A group communication system for wide area networks. Technical Report 2000-31, Institute of Computer Science, Hebrew University, Jerusalem, Israel, July 2000.

[4] R.-J. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. *Formal Aspects of Computing*, 12:18–40, 2000.

[5] M. Bickford and J. Hickey. An object-oriented approach to verifying group communication systems. http://www.cs.cornell.edu/jyh/papers/cav99_ooioa/, 1998.

[6] K. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.

[7] T. Budd. *An Introduction to Object-Oriented Programming, 2nd Edition.* Addison Wesley Longman, 1996.

[8] G. V. Chockler. An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions. Master's thesis, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1997.

[9] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Comput. Surv.*, 2001. To appear. Tentative publication date: December 2001. Previous version: MIT Technical Report MIT-LCS-TR-790, September 1999.

[10] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994. Preliminary version in Proceedings of OOPSLA'89, ACM SIGPLAN 4th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, pages 433–443, New Orleans, Louisiana, October 1989.

[11] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[12] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, August 1997. Full version to appear in ACM Transactions on Computer Systems (TOCS).

[13] S. J. Garland and N. A. Lynch. Using I/O automata for developing distributed systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, USA, 2000.

[14] S. J. Garland, N. A. Lynch, and M. Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems.* MIT Lab for Computer Science, Cambridge, MA, December 1997. http://sds.lcs.mit.edu/~garland/ioaLanguage.html.

[15] D. Harel and O. Kupferman. On the behavioral inheritance of state-based objects. In *Technology of Object-Oriented Languages (TOOLS)*, 2000.

[16] M. Hayden and R. van Renesse. Optimizing layered communication protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, November 1996.

[17] M. P. Heimdahl and C. L. Heitmeyer. Formal methods for developing high assurance computer systems: Working group report. In *Second IEEE Workshop on Industrial-Strength Formal Techniques (WIFT'98)*, Oct. 1998.

[18] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Real Time Systems Symposium (RTSS)*, Dec. 1994. Full version: Naval Research Laboratory NRL Memorandum Report 7619 and MIT Laboratory for Computer Science Technical Memorandum MIT/LCS/TM-511.

[19] C. L. Heitmeyer. On the need for 'practical' formal methods. In *Formal Techniques in Real-Time and Real-Time Fault-Tolerant Systems, Proc., 5th Intern. Symposium (FTRTFT'98)*, pages 18–26, Sept. 1998. LICS 1486 (invited paper).

[20] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Proceedings of Theoretical Aspects of Computer Software*, pages 548–568. Springer-Verlag (*LNCS* 526), 1991.

[21] J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for ensemble layers. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer-Verlag, Mar. 1999.

[22] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[23] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, Mar. 1994.

[24] S. Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Fifteenth Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[25] I. Keidar and R. Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20th International Conference on Distributed Computing Systems (ICDCS)*, pages 344–355, April 2000. Full version: MIT Lab. for Computer Science Tech. Report MIT-LCS-TR-794.

[26] I. Keidar, R. Khazan, N. Lynch, and A. Shvartsman. An inheritance-based technique for building simulation proofs incrementally. In *22nd International Conference on Software Engineering (ICSE)*, pages 478–487, June 2000.

[27] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. In *20th International Conference on Distributed Computing Systems (ICDCS)*, pages 356–365, April 2000. Full version: MIT Technical Memorandum MIT-LCS-TM-593a, June 1999, revised September 2000.

[28] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[29] B. Liskov and J. M. Wing. A New Definition of the Subtype Relation. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 118–141, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[30] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. and Sys.*, 16(1):1811–1841, Nov. 1994.

[31] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[32] N. A. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[33] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1995.

[34] J. Misra and K. Chandy. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[35] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[36] A. P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Information Processing Letters*, 39(1):45–49, July 1991.

[37] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 206–215. IEEE Computer Society Press, 1998.

[38] N. Soundarajan and S. Fridella. Inheriting and modifying behavior. In R. Ege, M. Singh, and B. Meyer, editors, *Technology of Object-Oriented Languages (TOOLS23)*, pages 148–162. IEEE Computer Society Press, 1998.

[39] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of OOPSLA '95 Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 30 of *ACM SIGPLAN Notices*, pages 200–214. ACM, Oct. 1995.

[40] I. Tarashchanskiy. Virtual Synchrony Semantics: Client-Server Implementation. Master's thesis, MIT Department of Electrical Engineering and Computer Science, August 2000. Master of Engineering.

[41] D. Yates, N. Lynch, V. Luchangco, and M. Seltzer. I/O automaton model of operating system primitives. Technical report, Harvard University and Massachusetts Institute of Technology, May 1999. Bachelor's thesis.