A Proof of Burns N-Process Mutual Exclusion Algorithm using Abstraction

Henrik E. Jensen[1] and Nancy A. Lynch[2]

[1] Department of Computer Science, Institute for Electronic Systems, Aalborg University, DK-9220 Aalborg Ø, Denmark.
e-mail: ejersbo@cs.auc.dk

[2] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 USA.
e-mail: lynch@theory.lcs.mit.edu

Abstract. Within the Input/Output Automata framework, we state and prove a general abstraction theorem giving conditions for preservation of safety properties from one automaton to another. We use our abstraction theorem to verify that Burns distributed mutual exclusion algorithm parameterized in the number of processes n satisfies the mutual exclusion property. The concrete n-process algorithm is abstracted by a simple 2-process algorithm which is property preserving with respect to the mutual exclusion property. The condition for property preservation is proved to be satisfied by use of the LP theorem prover with a minimum of user assistance, and the 2-process abstraction is automatically verified using the SPIN model checker.

1 Introduction

The majority of existing formal verification methods can be characterized as being either *theorem proving methods* or *model checking methods*, each of these having their own well-known advantages and disadvantages. Theorem proving methods can be applied to arbitrary systems and provide good insight into the systems at hand, but the methods require intelligent user interaction and are therefore only computer–assisted in a limited way. Model checking methods on the other hand are fully automatic, but limited to systems with finite state models or restricted kinds of infinite state models.

To benefit from the advantages of both methodologies there has recently been an increasing interest into the development of verification frameworks integrating theorem proving and model checking approaches, the key idea in this integration being the use of *abstraction*.

Given a system model, too large to be verified automatically, abstraction techniques are used to reduce this concrete model to a small (finite-state) abstract model which is *property preserving*. Meaning, that if the abstract model enjoys a property that implies, by the abstraction relation, the concrete property of interest, then the concrete model enjoys the concrete property.

The abstract model provides insight, as it captures the essence of the behavior of the concrete model with respect to the property of interest, and as it is finite state it can be verified by model checking methods. Theorem proving methods are used to prove, that the abstract model is indeed property preserving, and as a result no restrictions need to be imposed on the kind of concrete system models to which abstraction is amenable.

We propose a method, in the line of above, in the framework of Lynch and Tuttle's *Input/Output Automata (IOA)* [1,2]. We are interested in verifying safety properties of IOA. Properties are expressed as sets of traces, and hence verifying that an IOA A satisfies a trace safety property P amounts to proving that the set of traces of A is included in the set of traces of P. Given a concrete IOA C together with a safety property P_C, we give a general abstraction theorem stating conditions for an abstract IOA A and an abstract property P_A to be property preserving in the sense of above.

The theorem allows for abstraction of concrete system models regardless of the reason for their large size, being e.g. unbounded data structures or an unbounded number of identical processes (parameterized systems). The theorem states as a condition for property preservation the existence of a *parameterized simulation relation* from the concrete IOA to the abstract one, which allows for the abstraction of just a subset of the concrete behaviors.

We illustrate the use of our theorem on the case study of Burns distributed mutual exclusion algorithm parameterized in the number n of processes. We provide a 2-process abstraction and prove using the Larch Proof Assistant [3] that this abstraction satisfies the conditions for preservation of the mutual exclusion property. We verify, using the SPIN [4] model checker, that the abstraction enjoys the abstract mutual exclusion property, and by our abstraction theorem, the n-process algorithm then enjoys the original property.

Related Work

Property preserving abstraction methods have been studied e.g. in [5–10]. These methods are, like ours, all based on proving the existence of some kind of 'mimicing' relation from concrete system models to abstract ones. Different kinds of relations such as *simulation relations*, *homomorphic functions* [10,8,6,7] and *Galois connections* [9,5] have been considered. Our notion of parameterized simulation relations is a generalization of standard simulation relations.

Fully algorithmic methods have been developed, that use automatic abstraction to construct finite state abstract models of restricted kinds of large concrete models s.t. properties are preserved in both directions between the concrete and abstract models. Almost all existing model checkers for dense reactive systems (real–time/hybrid) are based on automatically constructed strongly preserving abstractions [11–13]. The idea is to let abstract states be *equivalence classes* of concrete states with respect to either some behavioral equivalence on concrete states or with respect to an equivalence on concrete states induced by satisfaction of the same properties in some property language.

Structural induction techniques have, together with model checking techniques, been used to verify parameterized systems. By model checking it is verified that *one* process enjoys the property of interest, and assuming that the property holds for some number n of processes (induction hypothesis) one just needs to prove that the property holds for $n+1$ processes as well. Using a finite representation of n processes assumed to enjoy the considered property, now allows model checking to establish that this representation composed with just one more process satisfies the property, and by induction principle this concludes that the property holds for *any* number of composed processes. Works on such techniques have been reported on in [14, 15].

Outline

This paper is organized as follows. In Section 2 we give some mathematical preliminaries used in the rest of the paper. In Section 3 we give the formal background of the IOA framework, and in Section 4 we present our abstraction theory. In Section 5 we present Burns n-process mutual exclusion algorithm which will serve as case-study for the use of our abstraction theorem. Section 6 describes the property preserving abstraction of Burns algorithm and Section 7 describes how the condition for preservation is proved and how LP is used in the proof. Section 8 describes the model checking of the abstract algorithm in the SPIN tool and Section 9 concludes.

2 Mathematical Preliminaries

Relations

A *relation* over sets X and Y is defined to be any subset of the cartesian product $X \times Y$. If R is a relation over X and Y, then we define the *domain* of R to be $dom(R) = \{x \in X \mid (x, y) \in R \text{ for some } y \in Y\}$, and the range of R to be $ran(R) = \{y \in Y \mid (x, y) \in R \text{ for some } x \in X\}$. If $dom(R) = X$ we say that R is *total* (on X). For $x \in X$, we define $R[x] = \{y \in Y \mid (x, y) \in R\}$.

Sequences

Let S be any set. The set of finite and infinite sequences of elements of S is denoted $seq(S)$. The symbol λ denotes the empty sequence and the sequence containing one element $s \in S$ is denoted by s. Concatenation of a finite sequence with a finite or infinite sequence is denoted by juxtaposition. A sequence σ is a *prefix* of a sequence ρ, denoted by $\sigma \leq \rho$, if either $\sigma = \rho$, or σ is finite and $\rho = \sigma\sigma'$ for some sequence σ'. A set Σ of sequences is *prefix closed* if, whenever some sequence is in Σ, all its prefixes are as well. A set Σ of sequences is *limit closed* if, an infinite sequence is in Σ whenever all its finite prefixes are.

If σ is a nonempty sequence then $first(\sigma)$ denotes the first element of σ, and $tail(\sigma)$ denotes the sequence obtained from σ by removing $first(\sigma)$. Also, if σ is finite, $last(\sigma)$ denotes the last element of σ.

If $\sigma \in seq(S)$, and $S' \subseteq S$, then $\sigma\lceil S'$ denotes the restriction of σ to elements in S', i.e. the subsequence of σ consisting of the elements of S'. If $\Sigma \subseteq seq(S)$, then $\Sigma\lceil S'$ is the set $\{\sigma\lceil S' \mid \sigma \in \Sigma\}$.

Assume $R \subseteq S \times S'$ is a total relation between sets S and S'. If $\sigma = s_0 s_1 s_2 \ldots$ is a nonempty sequence in $seq(S)$ then $R(\sigma)$ is the set of sequences $s'_0 s'_1 s'_2 \ldots$ over $ran(R)$ such that for all i, $s'_i \in R[s_i]$. If $\sigma = \lambda$ then $R(\sigma) = \{\lambda\}$. If $\Sigma \subseteq seq(S)$, then $R(\Sigma) = \bigcup_{\sigma \in \Sigma} R(\sigma)$

Lemma 1. *Let S and S' be sets and let $R \subseteq S \times S'$ be some total relation. For Σ and Σ' non-empty subsets of $seq(S)$, if $\Sigma \subseteq \Sigma'$ then $R(\Sigma) \subseteq R(\Sigma')$.*

Proof. Follows from the fact that the set $R(\sigma)$ is unique for any $\sigma \in seq(S)$. □

3 I/O Automata

As we will only be considering safety issues, we will use simplified versions of standard I/O automata that do not incorporate notions of fairness.

Definition 1. *An I/O automaton A is a tuple $(sig(A), states(A), start(A), trans(A))$ where,*

- *$sig(A)$ is a tuple $(in(A), out(A), int(A))$, consisting of disjoint sets of input, output and internal actions, respectively. The set $ext(A)$ of external actions of A is $in(A) \cup out(A)$, and the set $acts(A)$ of actions of A is $ext(A) \cup int(A)$.*
- *$states(A)$ is a set of states.*
- *$start(A) \subseteq states(A)$ is a nonempty set of start states.*
- *$trans(A) \subseteq states(A) \times acts(A) \times states(A)$ is a state transition relation.*

We let $s, s', u, u', \ldots$ range over states, and $\pi, \pi', \ldots$ over actions. We write $s \xrightarrow{\pi}_A s'$, or just $s \xrightarrow{\pi} s'$ if A is clear from the context, as a shorthand for $(s, \pi, s') \in trans(A)$.

An *execution fragment* $s_0 \pi_1 s_1 \pi_2 s_2 \ldots$ of an I/O automaton A is a finite or infinite sequence of alternating states and actions beginning with a state, and if it is finite also ending with a state, s.t. for all i, $s_i \xrightarrow{\pi_{i+1}} s_{i+1}$. An *execution* of A is an execution fragment α where $first(\alpha) \in start(A)$. A state s of A is *reachable* if $s = last(\alpha)$ for some finite execution α of A. The *trace* of an execution α, written $trace(\alpha)$, is the subsequence consisting of all the external actions occurring in α. We say that β is a trace of A if there is an execution α of A with $\beta = trace(\alpha)$. We denote the set of traces of A by $traces(A)$.

Composition

We can compose individual automata to represent complex systems of interacting components. We impose certain restrictions on the automata that may be composed.

Formally, we define a countable collection $\{A_i\}_{i\in I}$ of automata to be *compatible* if for all $i, j \in I$, $i \neq j$, all of the following hold: $int(A_i) \cap acts(A_j) = \emptyset$, $out(A_i) \cap out(A_j) = \emptyset$, and no action is contained in infinitely many sets $acts(A_i)$.

Definition 2. *The composition $A = \prod_{i\in I} A_i$ of a countable, compatible collection of I/O automata $\{A_i\}_{i\in I}$ is the automaton with:*

- $in(A) = \cup_{i\in I} in(A_i) - \cup_{i\in I} out(A_i)$
- $out(A) = \cup_{i\in I} out(A_i)$
- $int(A) = \cup_{i\in I} int(A_i))$
- $states(A) = \prod_{i\in I} states(A_i)$
- $start(A) = \prod_{i\in I} start(A_i)$
- *$trans(A)$ is the set of triples (s, π, s') such that, for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise $s_i = s'_i$*

The $\prod$ in the definition of $states(A)$ and $start(A)$ refers to ordinary Cartesian product. Also, s_i in the definition of $trans(A)$ denotes the ith component of state vector s.

Trace Properties

We will be considering properties to be proved about an I/O automaton A, as properties about the ordering, in traces of A, of some external actions from a subset of $ext(A)$.

A *trace property* P is a tuple $(sig(P), traces(P))$ where, $sig(P)$ is a pair $(in(P), out(P))$, consisting of disjoint sets of input and output actions, respectively. We let $acts(P)$ denote the set $in(P) \cup out(P)$. $traces(P)$ is a set of (finite or infinite) sequences of actions in $acts(P)$. We will be considering only *safety* properties, so we assume $traces(P)$ is nonempty, prefix-closed, and limit-closed.

An I/O automaton A and a trace property P are said to be *compatible* if, $in(P) \subseteq in(A)$ and $out(P) \subseteq out(A)$.

Definition 3. *Let A be an I/O automaton and P a trace property such that A and P are compatible. Then A satisfies P if, $traces(A) \lceil acts(P) \subseteq traces(P)$.*

4 Abstraction Theory

Suppose A is an I/O automaton and P is a trace property such that A and P are compatible. We will denote the pair (A, P) a *verification problem.* If (A, P) and (A', P') are two verification problems, we say that (A', P') is *safe* for (A, P) provided that A' satisfies P' implies that A satisfies P. In this section we give a general abstraction theorem, stating when one verification problem is safe for another.

If A and A' are two I/O automata and R is some relation from $ext(A')$ to $ext(A)$, we write, $s \overset{\beta}{\Longrightarrow}_{A'} s'$, when A' has a finite execution fragment α with $first(\alpha) = s$, $last(\alpha) = s'$ and $trace(\alpha) \lceil dom(R) = \beta$.

We now define the notion of a *parameterized simulation relation* between two automata A and A', and we give a soundness result needed for the abstraction theorem.

Definition 4. *Let A and A' be two I/O automata and let R be a relation from $ext(A')$ to $ext(A)$. A relation $f_R \subseteq states(A) \times states(A')$ is a simulation relation from A to A' parameterized by R provided,*

1. *If $s \in start(A)$ then $f_R[s] \cap start(A') \neq \emptyset$.*
2. *If $s \xrightarrow{\pi}_A s'$, $u \in f_R[s]$, and s and u are reachable states of A and A' respectively, then*
   - *(a) If $\pi \in ran(R)$, then $\exists \pi', u'$ such that $u \overset{\pi'}{\Longrightarrow}_{A'} u'$, $(\pi', \pi) \in R$ and $(s', u') \in f_R$.*
   - *(b) If $\pi \notin ran(R)$, then $\exists u'$ such that $u \overset{\lambda}{\Longrightarrow}_{A'} u'$ and $(s', u') \in f_R$.*

We write $A \leq_R A'$ if there is a simulation from A to A' parameterized by R.

Lemma 2. $A \leq_R A' \Rightarrow traces(A)\lceil ran(R) \subseteq R(traces(A')\lceil dom(R))$

Proof. Analogous to proof for standard forward simulation [16]. □

Theorem 1. *Let (A, P) and (A', P') be two verification problems. Also, let R be a relation from $ext(A')$ to $ext(A)$, with $dom(R) = acts(P')$ and $ran(R) = acts(P)$, such that $R(traces(P')) \subseteq traces(P)$. If,*

$$A \leq_R A' \quad \text{and} \quad A' \text{ satisfies } P'$$

then

$$A \text{ satisfies } P$$

Proof. Assume that $A \leq_R A'$ and that A' satisfies P'. From second assumption we have $traces(A')\lceil acts(P') \subseteq traces(P')$ and from Lemma 1 we get $(*)$ $R(traces(A')\lceil acts(P')) \subseteq R(traces(P'))$, as R is total on $acts(P')$. Also, from Lemma 2, and the fact that $dom(R) = acts(P')$ and $ran(R) = acts(P)$, we have that $traces(A)\lceil acts(P) \subseteq R(traces(A')\lceil acts(P'))$ and this together with $(*)$ now gives us that $traces(A)\lceil acts(P) \subseteq R(traces(P'))$ and finally as $R(traces(P')) \subseteq traces(P)$ we get the wanted result, namely $traces(A)\lceil acts(P) \subseteq traces(P)$ i.e. A satisfies P. □

5 Burns N–Process Mutual Exclusion Algorithm

In this section we present Burns n-process distributed mutual exclusion algorithm, which we will verify with respect to the mutual exclusion property using the abstraction approach from the previous section.

The algorithm runs on a shared memory model consisting of n processes $P_1, \ldots, P_n$ together with n shared variables $flag_1, \ldots, flag_n$, each $flag_i$ writable by process P_i and readable by all other processes. Each process P_i is acting on behalf of a user process U_i which can be thought of as some application program. The processes $P_1, \ldots, P_n$ competes for mutual exclusive access to a shared resource by reading and writing the shared variables in a way determined by the algorithm.

We model the algorithm formally as an I/O automaton *BurnsME*, which is the composition of a shared memory automaton M and a set of user automata $U_1, \ldots, U_n$. M models the n processes $P_1, \ldots, P_n$ together with the set of shared variables $\mathit{flag}_1, \ldots, \mathit{flag}_n$, and it is modelled as one big I/O automaton, where the process and variable structure is captured by means of some locality restrictions on transitions. Each state in M consists of a state for each process P_i, plus a value for each shared variable flag_i. A state variable v of process P_i in automaton M is denoted $M.v_i$. Similarly, $U.v_i$ denotes a state variable v of automaton U_i. We omit the preceding $U(M)$ and the subscripts i when these are clear from the context.

The inputs to M are (for all $1 \leq i \leq n$) actions try_i, which models a request by user U_i to process P_i for access to the shared resource, and actions exit_i, which models an announcement by user U_i to process P_i that it is done with the resource. The outputs of M are crit_i, which models the granting from process P_i of the resource to U_i, and rem_i, which models P_i telling U_i that it can continue with the rest of its work.

Each process P_i executes three loops. The first two loops involve checking the flags of all processes with smaller indices, i.e. all flag_j, $1 \leq j < i$. The first loop is actually not needed for the mutual exclusion condition, but is important to guarantee progress. The two loops are modelled in M by internal actions $\mathit{test\text{-}sml\text{-}fst}(j)_i$ and $\mathit{test\text{-}sml\text{-}snd}(j)_i$, where j is a parameter denoting the index of the flag to be read by process P_i. In between the first two loops process P_i sets its own flag_i to 1, modelled in M by internal action $\mathit{set\text{-}flg\text{-}1}_i$. If both loops are successfully passed, meaning all the considered flags have value 0, then P_i can proceed to the third loop, which involves checking the flags of all processes with larger indices, i.e. flag_j, $i < j \leq n$. This is modelled by internal action $\mathit{test\text{-}lrg}(j)_i$. If process P_i passes all three loops successfully, it proceeds to its critical region. Process P_i keeps the value of its flag_i to 1 from when it starts testing flags with larger indices and until it leaves its critical region.

The User Automata: Each automaton U_i has as single state variable a program counter pc initially having the value rem, indicating that U_i starts in its remainder region ready to make a request for access to the shared resource.

```
output: $try_i$
    Pre: $pc = rem$
    Eff: $pc := try$

input: $crit_i$
    Eff: $pc := crit$

output: $exit_i$
    Pre: $pc = crit$
    Eff: $pc = exit$

input: $rem_i$
    Eff: $pc := rem$
```

The Shared Memory Automaton: The state of each process P_i in M is modelled by two state variables: a program counter pc initially having the value *rem*, and a set S of process id's initially empty, used to keep track of the indices of all shared flags that have successfully been checked in one of the three loops.

```
input: $try_i$
    Eff: $pc := set\text{-}flg\text{-}0$

internal: $set\text{-}flg\text{-}0_i$
    Pre: $pc = set\text{-}flg\text{-}0$
    Eff: $flag_i := 0$
         if $i = 1$ then
           $pc := set\text{-}flg\text{-}1$
         else
           $pc := test\text{-}sml\text{-}fst$

internal: $test\text{-}sml\text{-}fst(j)_i$
    Pre: $pc = test\text{-}sml\text{-}fst$
         $j \notin S$
         $1 \le j \le i - 1$
    Eff: if $flag_j = 1$ then
           $S := \emptyset$
           $pc := set\text{-}flg\text{-}0$
         else
           $S := S \cup \{j\}$
           if $|S| = i - 1$ then
             $S := \emptyset$
             $pc := set\text{-}flg\text{-}1$

internal: $set\text{-}flg\text{-}1_i$
    Pre: $pc = set\text{-}flg\text{-}1$
    Eff: $flag_i := 1$
         if $i = 1$ then
           $pc := test\text{-}lrg$
         else
           $pc := test\text{-}sml\text{-}snd$

internal: $test\text{-}sml\text{-}snd(j)_i$
    Pre: $pc = test\text{-}sml\text{-}snd$
         $j \notin S$
         $1 \le j \le i - 1$
    Eff: if $flag_j = 1$ then
           $S := \emptyset$
           $pc := set\text{-}flg\text{-}0$
         else
           $S := S \cup \{j\}$
           if $|S| = i - 1$ then
             $S := \emptyset$
             if $i = n$ then
               $pc := leave\text{-}try$
             else
               $pc := test\text{-}lrg$

internal: $test\text{-}lrg(j)_i$
    Pre: $pc = test\text{-}lrg$
         $j \notin S$
         $i + 1 \le j \le n$
    Eff: if $flag_j = 1$ then
           $S := \emptyset$
         else
           $S := S \cup \{j\}$
           if $|S| = n - i$ then
             $pc := leave\text{-}try$

output: $crit_i$
    Pre: $pc = leave\text{-}try$
    Eff: $pc := crit$

input: $exit_i$
    Eff: $pc := reset$

internal: $reset_i$
    Pre: $pc = reset$
    Eff: $flag_i := 0$
         $S := \emptyset$
         $pc := leave\text{-}exit$

output: $rem_i$
    Pre: $pc = leave\text{-}exit$
    Eff: $pc := rem$
```

The mutual exclusion property for *BurnsME* is a set of trace properties $P_{\{i,j\}}$, one for each subset $\{i,j\}_{i \neq j}$ in the set of process indices $\{1,\ldots,n\}$, such that $sig(P_{\{i,j\}})$ has as its only actions the set of output actions from *BurnsME* with indices i and j, and $traces(P_{\{i,j\}})$ is the set of sequences such that no two $crit_i$, $crit_j$ events occur (in that order) without an intervening $exit_i$ event, and similarly for i and j switched.

6 Abstracting *BurnsME*

To construct a property-preserving abstraction of *BurnsME* we examine the mutual exclusion property as stated in the previous section. The property is the conjunction of properties $P_{\{i,j\}}$, one for each subset $\{i,j\}$ of indices in $\{1,\ldots,n\}$, with each $P_{\{i,j\}}$ saying that processes P_i and P_j can not both be in their critical section at the same time.

The abstraction idea is now as follows. We will construct a single finite-state abstraction which preserves the external behavior of *any* two concrete processes P_i and P_j running in the environment of *all* other processes and users. This abstraction will then preserve the mutual exclusion property between any pair of concrete processes and hence the complete property.

Formally, we construct an abstract automaton *ABurnsME*, which is the composition of a shared memory automaton AM, with two user automata AU_0 and AU_1. AM models two abstract processes AP_0 and AP_1 together with two shared variables $flag_0$ and $flag_1$. AP_0 and AP_1 are abstract representations of any pair of concrete processes P_i and P_j within the environment of all other concrete processes, such that AP_0 represents *the smaller process* P_i and AP_1 represents *the larger process* P_j for $i < j$.

A state of AM consists of a state for each of the abstract processes AP_0 and AP_1 together with values for each of the shared variables $flag_0$ and $flag_1$. A state variable v of process AP_i in automaton AM is denoted $AM.v_i$. Similarly, $AU.v_i$ denotes a variable v of automaton AU_i. We omit the preceding $AM(AU)$ and the subscripts i when these are clear from the context.

The interface between AU_0 and AP_0 (AU_1 and AP_1) is identical to the interface between any concrete user automaton U_i and corresponding concrete process P_i, except for a change of indices. Process AP_0 has as actions abstracted versions of all actions actions in any smaller process P_i, and AP_1 has abstracted versions of all actions in any larger process P_j.

The automata AU_0 and AU_1 of *ABurnsME*, are identical to each other and to any concrete user automaton U_i except for a change of indices.

The Abstract Shared Memory Automaton: The state of each of the abstract processes AP_0 and AP_1 is modelled, analogous to the state of concrete processes, by two state variables: a program counter pc, initially rem and a set S of indices, initially empty. The transitions for AP_0 are as follows.

```
input: $try_0$
    Eff: $pc := set\text{-}flg\text{-}0$

internal: $set\text{-}flg\text{-}0_0$
    Pre: $pc = set\text{-}flg\text{-}0$
    Eff: $flag_0 := 0$
         $pc := test\text{-}sml\text{-}fst$

internal: $set\text{-}flg\text{-}0\text{-}sml_0$
    Pre: $pc = set\text{-}flg\text{-}0$
    Eff: $flag_0 := 0$
         $pc := set\text{-}flg\text{-}1$

internal: $test\text{-}sml\text{-}fail_0$
    Pre: $pc \in \{test\text{-}sml\text{-}fst, test\text{-}sml\text{-}snd\}$
    Eff: $pc := set\text{-}flg\text{-}0$

internal: $test\text{-}sml\text{-}fst\text{-}succ_0$
    Pre: $pc = test\text{-}sml\text{-}fst$
    Eff: $pc := set\text{-}flg\text{-}1$

internal: $set\text{-}flg\text{-}1_0$
    Pre: $pc = set\text{-}flg\text{-}1$
    Eff: $flag_0 := 1$
         $pc := test\text{-}sml\text{-}snd$

internal: $set\text{-}flg\text{-}1\text{-}sml_0$
    Pre: $pc = set\text{-}flg\text{-}1$
    Eff: $flag_0 := 1$
         $pc := test\text{-}lrg$

internal: $test\text{-}sml\text{-}snd\text{-}succ_0$
    Pre: $pc = test\text{-}sml\text{-}snd$
    Eff: $pc := test\text{-}lrg$
```

```
internal: $test\text{-}other\text{-}flg_0$
    Pre: $pc = test\text{-}lrg$
         $S = \emptyset$
    Eff: if $flag_1 = 0$ then
             $S := S \cup \{1\}$

internal: $test\text{-}lrg\text{-}fail_0$
    Pre: $pc = test\text{-}lrg$
    Eff: $S := \emptyset$

internal: $test\text{-}lrg\text{-}succ_0$
    Pre: $pc = test\text{-}lrg$
         $S = \{1\}$
    Eff: $pc := leave\text{-}try$

output: $crit_0$
    Pre: $pc = leave\text{-}try$
    Eff: $pc := crit$

input: $exit_0$
    Eff: $pc := reset$

internal: $reset_0$
    Pre: $pc = reset$
    Eff: $flag_0 := 0$
         $S := \emptyset$
         $pc := leave\text{-}exit$

output: $rem_0$
    Pre: $pc = leave\text{-}exit$
    Eff: $pc := rem$
```

One of the consequences of AP_0 representing the behavior of *any* smaller process is that AP_0 has two actions for setting its own flag to 0 (1): $set\text{-}flg\text{-}0\text{-}sml_0$ ($set\text{-}flg\text{-}1\text{-}sml_0$) and $set\text{-}flg\text{-}0_0$ ($set\text{-}flg\text{-}1_0$). The first representing that the concrete process P_1 (the one with smallest index) sets its flag to 0 (1), where after it skips the test of flags with smaller indices, as there are none, and sets it program counter to *set-flg-1* (*test-lrg*). The second representing that any other smaller process sets it flag to 0 (1) and thereafter tests flags with smaller indices, which do exist in this case. AP_0 represents that a smaller process fails or succeeds a test of smaller flags by allowing abstract fail or succeed actions whenever its program counter is *test-sml-fst* or *test-sml-snd*. No further preconditions apply to these actions as all information about the actual values of smaller flags have been abstracted away.

In order for AP_0 to succeed its test of flags with larger indices, it must test the flag of abstract process AP_1 as AP_1 represent some larger process. This test is modelled by the action *test-other-flg*$_0$. Having read this flag successfully (i.e. as 0) AP_0 can now enter its critical region. Also, as long as AP_0 has program counter *test-lrg* it can at any time perform an abstract action *test-lrg-fail*.

Abstract process AP_1 is modelled analogously to AP_0, and its transitions are as follows.

input: *try*$_1$
  Eff: $pc :=$ *set-flg-0*

internal: *set-flg-0*$_1$
  Pre: $pc =$ *set-flg-0*
  Eff: $flag_1 := 0$
    $pc =$ *test-sml-fst*

internal: *test-other-flg*$_1$
  Pre: $pc \in \{$*test-sml-fst*, *test-sml-snd*$\}$
    $S = \emptyset$
  Eff: if $flag_0 = 0$ then
      $S := S \cup \{0\}$

internal: *test-sml-fail*$_1$
  Pre: $pc \in \{$*test-sml-fst*, *test-sml-snd*$\}$
  Eff: $S := \emptyset$
    $pc :=$ *set-flg-0*

internal: *test-sml-fst-succ*$_1$
  Pre: $pc =$ *test-sml-fst*
    $S = \{0\}$
  Eff: $S := \emptyset$
    $pc :=$ *set-flg-1*

internal: *set-flg-1*$_1$
  Pre: $pc =$ *set-flg-1*
  Eff: $flag_1 := 1$
    $pc :=$ *test-sml-snd*

internal: *test-sml-snd-succ*$_1$
  Pre: $pc =$ *test-sml-snd*
    $S = \{0\}$
  Eff: $pc :=$ *test-lrg*

internal: *test-sml-snd-succ-lrg*$_1$
  Pre: $pc =$ *test-sml-snd*
    $S = \{0\}$
  Eff: $pc :=$ *leave-try*

internal: *test-lrg-fail*$_1$
  Pre: $pc =$ *test-lrg*
  Eff: $pc :=$ *test-lrg*

internal: *test-lrg-succ*$_1$
  Pre: $pc =$ *test-lrg*
  Eff: $pc :=$ *leave-try*

output: *crit*$_1$
  Pre: $pc =$ *leave-try*
  Eff: $pc :=$ *crit*

input: *exit*$_1$
  Eff: $pc :=$ *reset*

internal: *reset*$_1$
  Pre: $pc =$ *reset*
  Eff: $flag_1 := 0$
    $S := \emptyset$
    $pc :=$ *leave-exit*

output: *rem*$_1$
  Pre: $pc =$ *leave-exit*
  Eff: $pc :=$ *rem*

The abstract mutual exclusion property for *ABurnsME* is the one trace property $P_{(0,1)}$ with $sig(P_{(0,1)})$ having as its only actions the output actions of *ABurnsME* and $traces(P_{(0,1)})$ being the set of sequences such that no two $crit_0$ and $crit_1$ events occur (in that order) without an intervening $exit_0$ event, and similarly for 0 and 1 switched.

Now, for any $\{i,j\}$ we define a relation $R_{\{i,j\}}$ from $acts(P_{(0,1)})$ to $acts(P_{\{i,j\}})$. We assume $i < j$.

$$\begin{aligned} R_{\{i,j\}} = \{&(try_0, try_i), (try_1, try_j), (crit_0, crit_i), (crit_1, crit_j),\\ &(exit_0, exit_i), (exit_1, exit_j), (rem_0, rem_i), (rem_1, rem_j)\} \end{aligned}$$

By definition, $R_{\{i,j\}}(P_{(0,1)}) \subseteq P_{\{i,j\}}$. We use $R_{\{i,j\}}$ as parameter to a state relation $f_{R_{\{i,j\}}}$ defined as follows.

Definition 5. *$f_{R_{\{i,j\}}}$ is a relation from states(BurnsME) to states(ABurnsME) such that $f_{R_{\{i,j\}}}(s,u)$ iff :*

- *$u.AU.pc_0 = s.U.pc_i$ and $u.AU.pc_1 = s.U.pc_j$*
- *$u.AM.pc_0 = s.M.pc_i$ and $u.AM.pc_1 = s.M.pc_j$*
- *$u.flag_0 = s.flag_i$ and $u.flag_1 = s.flag_j$*
- *$u.AM.S_0 = \{1\}$ if $j \in s.M.S_i$ and $u.AM.S_1 = \{0\}$ if $i \in s.M.S_j$*

Note, that we use dot notation to denote the value of a given variable in a state.

Theorem 2. *For all $\{i,j\}$ subsets of $\{1,\ldots,n\}$, $f_{R_{\{i,j\}}}$ is a simulation relation from BurnsME to ABurnsME parameterized by $R_{\{i,j\}}$.*

7 The Simulation Proof

To prove Theorem 2 for *all* $\{i,j\}$ we prove it for *any* $\{i,j\}$ with i and j treated as Skolem constants. The proof follows the line of a standard forward simulation proof [2]. To see that $f_{R_{\{i,j\}}}$ is in fact a parameterized simulation relation we check the two conditions in Definition 4. The start condition is trivial, because the initial states of *BurnsME* and *ABurnsME* have the value of *pc* set to *rem* for all processes and users, and they have all flags set to 0 and all sets of indices empty.

Now, for the step condition suppose that $s \in states(BurnsME)$ and $u \in states(ABurnsME)$ s.t. $f_{R_{\{i,j\}}}(s,u)$. We then consider cases based on the type of action π_x performed by s on a transition $s \xrightarrow{\pi_x} s'$. For each action π_x we consider $x = i$, $x = j$ and $x \notin \{i,j\}$. The proof is relatively simple, as the execution fragment corresponding to a certain concrete action π_x for the most cases can be picked to be the abstract version of the concrete action. So the proof is a rather straightforward matching up of concrete actions with their abstract counterparts.

In [17] a framework is introduced for specifying and reasoning about IOA using the Larch tools. The notion of IOA is formalized in the Larch Shared Language (LSL) [18] which is supported by a tool that produces input for LP. LP is a theorem prover for first-order logic designed to assist users who employ standard proof techniques such as proofs by cases, induction, and contradiction.

In [17] LP is used to construct standard simulation proofs, and we use the framework introduced here to (re)do the proof of Theorem 2. Using LP for the simulation proof allows us to disregard many of the routine steps which are needed in the hand proof, as LP carries these out automatically. The main user assistance that LP needs for the proof is the input of the corresponding abstract execution fragment for each concrete action. The rest of the user guidance consists of directing LP to break some proof parts into cases, and directing LP to use whatever information it has already got to try and do some rewriting to complete proof subgoals.

Having proved Theorem 2 now allows us to apply Theorem 1 and conclude that if *ABurnsME* satisfies $P_{(0,1)}$ then *BurnsME* satisfies $P_{\{i,j\}}$ for all $\{i,j\}$ subsets of $\{1,\ldots,n\}$. That *ABurnsME* satisfies $P_{(0,1)}$ is model checked using the SPIN model checker.

8 Model Checking *ABurnsME*

The SPIN verification tool relies on a simple yet powerful modelling language based on processes communicating either on asynchronous channels or via shared variables. As property language SPIN uses Linear Time Temporal Logic (LTL).

We translate the IOA description of *ABurnsME* into a SPIN model and we translate the property $P_{(0,1)}$ into an LTL formula suitable for SPIN. Automaton *ABurnsME* is translated into a SPIN model with two processes implementing the behavior of the composition of AU_0 with AP_0, and AU_1 with AP_1, respectively. Each process has variables representing the program counters and internal index sets of the corresponding IOA. The SPIN processes each execute a loop checking preconditions and performing effects of representations of the actions of their corresponding IOA. For each action, checking preconditions and performing effects is done atomically, i.e. non-interleaved with any other actions, hence preserving the exact IOA semantics.

The property $P_{(0,1)}$ is translated into an LTL property of the SPIN model. From *ABurnsME* it is easy to see, that the property $P_{(0,1)}$ can be stated (equivalently) as a property of states rather than actions. Recall, that $P_{(0,1)}$ is the set of sequences of external actions such that no two $crit_0$ and $crit_1$ events occur (in that order) without an intervening $exit_0$ event, and similarly for 0 and 1 switched. But, if an action $crit_i$, $i \in \{0,1\}$, is performed then $AM.pc_i$ gets the value *crit* and it can not change until an $exit_i$ action is performed. Consequently, the property $P_{(0,1)}$ can equivalently be stated as an invariant saying that for any state u it is the case that $u.AM.pc_0$ and $u.AM.pc_1$ can not both have the value *crit*. This property is exactly in the form of an LTL property and can be stated in the property language of SPIN without translation.

Using SPIN to analyse the abstracted algorithm with respect to its corresponding abstract property stated in LTL, immediately lead to a successful verification result.

9 Conclusion

In this paper we have presented a general abstraction theorem within the Input/Output Automata framework, which gives conditions for preservation of safety properties from one (abstract) automaton to another (concrete) automaton. The preservation condition is expressed by the requirement of a parameterized simulation relation from the concrete to the abstract automaton.

We have used our abstraction theorem to verify that Burns n-process mutual exclusion algorithm enjoys the mutual exclusion property, by constructing and proving a 2-process property preserving abstraction of the concrete algorithm. We have used the Larch Proof Assistant, LP to prove the conditions for property preservation, and by using the SPIN model checker we have successfully verified the abstraction.

Using our abstraction approach to prove Burns algorithm led to a proof style having the advantages of providing both essential insight into the algorithm and some automatic verification. The insight gained in the case of Burns algorithm is that its essential behavior with respect to the mutual exclusion property can be abstracted to the behavior of just two processes.

In general, our abstraction approach does of course not guarantee the existence of finite state abstractions for any concrete system neither does it provide a method for finding such abstractions. Further case studies needs to be considered to identify classes of systems to which certain specific abstraction techniques/patterns can be applied. The specific approach applied to the Burns algorithm has also been succesfully applied to the Bakery mutual exclusion algorithm, and it seems to be useful in general to many parameterized systems where the property of interest can be stated as a conjunction of equivalent properties over a finite subset of components.

Tool support is essential to assist in the process of finding common abstraction patterns for classes of systems, and we are investigating approaches to further integrate model checking facilities with the Larch tools.

References

1. Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, 2(3)219–246, 1989.
2. Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.
3. S.J. Garland and J.V. Guttag. A Guide to LP, the Larch Prover. Technical Report, Research Report 82, Digital Systems Research Center, 1991.
4. Gerard Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.
5. D. Dams. Abstract Interpretation and Partition Refinement for Model Checking. PhD thesis, Eindhoven University of Technology, 1996.
6. Jürgen Dingel and Thomas Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc. of CAV'95*, Lecture Notes in Computer Science, volume 939, pages 54–69, 1995.
7. E.M. Clarke, O. Grumberg and D.E. Long. Model Checking and Abstraction. In *Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1992.
8. R.P. Kurshan. Analysis of Discrete Event Coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, Lecture Notes in Computer Science, volume 430, pages 414–454. Springer Verlag, 1989.

9. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, pages 6:11–44, 1995.
10. Olaf Müller and Tobias Nipkow. Combining Model Checking and Deduction for I/O-Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, volume 1019, pages 1–16. Springer Verlag, 1995.
11. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of the 4*th *DIMACS Workshop on Verification and Control of Hybrid Systems*, Lecture Notes in Computer Science, October 1995.
12. Thomas A. Henzinger, Pei–Hsin Ho, and Howard Wong–Toi. A Users Guide to HyTech. Technical Report, Department of Computer Science, Cornell University, 1995.
13. Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *36th Annual Symposium on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.
14. Pierre Wolper and Vincianne Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. *International Workshop on Automatic Verification Methods for Finite State Machines*, Lecture Notes in Computer Science, volume 407, 1989.
15. R.P. Kurshan and K. McMillan. A Structural Induction Theorem for Processes. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, 1989.
16. N. Lynch and M. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proc. of the 6th ACM Symposium on Principles of Distributed Computation*, pages 137–151, 1987.
17. Jørgen Søgaard–Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-Assisted Simulation Proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification (5th International Conference, CAV'93, Elounda, Greece, June/July 1993)*, Lecture Notes in Computer Science, volume 697, pages 305–319. Springer Verlag, 1993.
18. J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer Verlag, 1993.