On Proving Safety Properties by Integrating Static Analysis, Theorem Proving and Abstraction *

V. Rusu† E. Singerman
SRI International, Menlo Park, California, USA

Abstract. We present a new approach for proving safety properties of reactive systems, based on tight interaction between static analysis, theorem proving and abstraction techniques. The method incrementally constructs a proof or finds a counterexample. Every step consists of applying one of the techniques and makes constructive use of information obtained from failures in previous steps. The amount of user intervention is limited and is highly guided by the system at each step. We demonstrate the method on three simple examples, and show that by using it one can prove more properties than by using each component as a stand-alone.

1 Introduction

Theorem proving [GM95, ORS+95, CCF+97][1] is a powerful and general way to verify safety properties of reactive systems, but its use in mechanical verification requires a serious amount of both insightful and labor-intensive manual guidance from the human verifier. Model checking [BCM+92, H91, LPY97] is largely automatic but it only addresses a limited class of essentially finite-state system. Abstraction [SUM96, DGG97, GS97, BLO98, CU98] can be used to translate an infinite-state system to a finite-state systems so as to preserve the property being verified. This can reduce the manual burden of the verification but the discovery of a suitable property-preserving abstraction takes considerable human ingenuity. Furthermore, when the abstracted system fails verification, this could either be because the abstraction was too coarse or because the system did not satisfy the property. It takes deep insight to draw useful information from such a failure. This paper addresses these problems by presenting a methodology for integrating static analysis [CC77, HPR97, BL], theorem proving, and abstraction that does not tax the patience and ingenuity of the human verifier. In this methodology

*This research was supported by National Science Foundation grant CCR-9509931. The first author is also supported by a Lavoisier grant of the French Ministry of Foreign Affairs.

†Postal address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, Email: rusu@csl.sri.com, Fax: +1-650-8592844

[1]Due to space limitations, we cite only a few of the relevant contributions in each domain.

From TACAS'99: Springer LNCS Vol. 1579, pp. 178-192, March 1999

1. The choice of the abstraction mapping can be guided by the subgoals in a failed proof attempt.
2. A failed verification attempt at the abstract level suggests either strengthened invariants or a more refined abstraction.
3. The iterative process, when it terminates, yields a counterexample indicating how the property is violated or a proof that the property is satisfied.

We also show that the combination of abstraction and theorem proving is strictly more powerful than verification based on theorem proving with systematic dynamic invariant strengthening techniques.

In our method, the verification starts with a one-time use of static analysis, generating true-by-construction invariants that are communicated to both the theorem-proving and abstraction components. The rest of the process involves a tight interaction between the prover and abstraction generator, in which each step makes constructive use of information obtained from failures in previous steps. The method assists the user in discovering relevant auxiliary invariants and suitable abstraction mappings while progressing towards a proof or a counterexample. Using this "small increments" approach the required amount of user ingenuity is reduced. Instead of having to rely on keen insight of the problem right from the start, the user gains increasing insight as she progresses in the verification task, enabling her to conclude eventually.

The rest of the paper is organized as follows. In Section 2 we present some basic terminology and an overview of the static analysis, theorem proving and abstraction techniques that we are using. Section 3 presents our approach for integrating these techniques, which we introduce through the verification of a simple example. Section 4 contains a formal comparison of the relative power of the theorem proving and abstraction techniques, together with an example demonstrating that our method is strictly more powerful than using each component as a stand-alone. We conclude in Section 5 and present some future work directions.

2 The Components

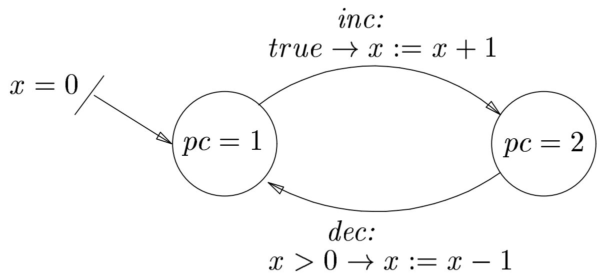
We use transition systems as a computational model for reactive programs. A transition system T consists of a finite set of typed variables V, an initial condition Θ and a finite set of guarded transitions $\mathcal{T}$. The variables can be either control or data variables; the control variables are of a finite type *Location*. Each transition $\tau \in \mathcal{T}$ is labeled and consists of a guard and an assignment. A state is a type-consistent valuation of the variables, and the initial condition Θ is a predicate on states. Each transition τ induces a transition relation ρ_τ relating the possible before and after states. The global transition relation of the system is $\rho_T = \bigcup_{\tau \in \mathcal{T}} \rho_\tau$.

A computation of the transition system is an infinite sequence of states, in which the first state satisfies the initial condition and every two consecutive states are in the transition relation. The parallel (asynchronous) composition of transition systems is defined using interleaving in the usual manner. For a transition τ and a state predicate φ, the predicate $\widetilde{pre}_\tau(\varphi)$ characterizes all the states from which, after taking transition τ, the predicate φ holds: $\widetilde{pre}_\tau(\varphi): \quad \forall s^{'}.\rho_\tau(s,s^{'}) \supset \varphi(s^{'})$. Likewise, $post_\tau(\varphi)$ characterizes the states that can be reached by taking transition τ from some state satisfying φ: $post_\tau(\varphi): \quad \exists s.\rho_\tau(s,s^{'}) \wedge \varphi(s)$. These predicates are also defined globally for the transition system T: $\widetilde{pre}_T(\varphi): \quad \forall s^{'}.\rho_T(s,s^{'}) \supset \varphi(s^{'})$ and $post_T(\varphi): \quad \exists s.\rho_T(s,s^{'}) \wedge \varphi(s)$. In the sequel, we omit T when it is understood from the context.

We now briefly describe the static analysis, theorem proving and abstraction techniques we integrate in our approach. In should be stressed that the identity of the particular tools we use is not the main point here, but rather the way in which we integrate them. One could, for example, use POLKA [HPR97] as the static analysis tool, InVesT [BLO98] as the abstraction tool, etc.

2.1 Static Analysis

For automatically generating invariants we use a method similar to that suggested by [BLO98]. The analysis starts by computing local invariants at every control location: the local invariant of a control location l is the disjunction of $post_\tau(true)$, for all transitions τ leading to l. Then, the local invariants are propagated to other control locations of the system to obtain global invariants. For example, in the simple transition system illustrated below, static analysis yields the local invariant $\Box(pc = 1 \supset x \geq 0)$. Then, since $x \geq 0$ is preserved by transition *inc*, it is a global invariant.



2.2 Theorem Proving

We use PVS [ORS+95] for invariant strengthening [GS96, HS96]. Given a transition system T and a state predicate I, we say that I is *inductive* if $I \supset \widetilde{pre}(I)$. Obviously, if I is inductive and holds at the initial state of T, then I is an invariant of T. When I is not inductive, we can strengthen it by taking $I \wedge \widetilde{pre}(I)$ and check if the latter is inductive, that is, whether $I \wedge \widetilde{pre}(I) \supset \widetilde{pre}(I \wedge \widetilde{pre}(I))$; or equivalently, whether $I \wedge \widetilde{pre}(I) \supset \widetilde{pre}^2(I)$. In general, this procedure terminates if there exists an n such that

$$I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I) \supset \widetilde{pre}^{n+1}(I). \tag{1}$$

In this case, it follows that $I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I)$ is inductive: in particular, I is an invariant.

This technique can be implemented in PVS as follows. We use a simple invariance rule stating that I is an invariant if it is true initially and is preserved by all transitions. If I is inductive then applying the rule once would complete the proof. Otherwise, the prover presents a number of pending (unproved) subgoals: each subgoal results from the fact that I is not preserved by some transition. We then apply the invariance rule to the predicate obtained by taking the conjunction of I and all the unproved subgoals: this amounts to attempting to prove that $I \wedge \widetilde{pre}(I)$ is inductive. If there exists an n such that (1) holds, then repeating this process n times would eliminate all the subgoals and complete the proof. This leads to a fully automatic procedure (that is not guaranteed to halt).

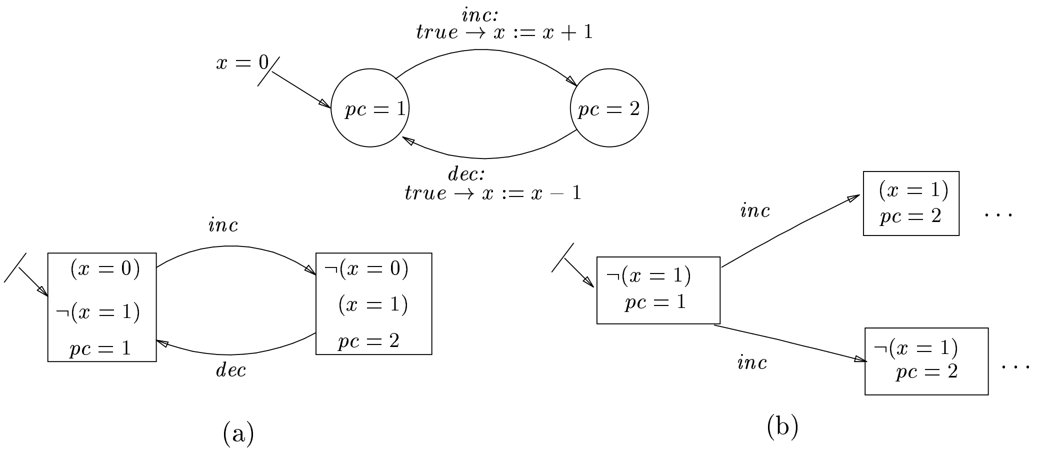
2.3 Abstraction

We use the abstraction technique described in [GS97]. The abstraction of a concrete transition system T relative to a finite set of state predicates $\mathcal{B} = \{B_1, \ldots, B_k\}$ called *boolean abstract variables*, is a transition system denoted $T/\mathcal{B}$. The states of the abstract system $T/\mathcal{B}$ are called *abstract states*; every abstract state is labeled with a valuation of the control variables of T and of the abstract variables. Let us now briefly describe how $T/\mathcal{B}$ is constructed.

The initial abstract state is labeled with the initial control configuration of T and with the truth values of the abstract variables at the initial concrete state. Assume now that s^A is an abstract state, the abstract transitions going out of s^A are then generated. Every concrete transition τ, originating from a concrete state with the same control configuration as s^A, can give rise to several abstract transitions. Each of these transitions will have the same label as τ and lead to an abstract state obtained by computing (with PVS) the effect of τ (starting from s^A) on the control and abstract variables.

Consider, for example, the concrete system illustrated below. An abstraction relative to $B_1 : (x = 0)$ and $B_2 : (x = 1)$ generates the abstract system (a); while an abstraction only relative to B_2 yields the abstract system (b), of which only the initial portion is shown. Note that in the latter, simulating the concrete transition *inc* gives rise to two successors. This is because starting at the initial abstract state, where $\neg(x = 1)$ holds, the transition *inc* performing $x := x + 1$ can either lead to a state in which $(x = 1)$ is true, or to a state in which the latter is false. Note also that in the abstract system (a), the only state labeled $pc = 2$ is also labeled

$(x = 1)$; we say this abstraction "shows" the property $\Box(pc = 2 \supset (x = 1))$. On the other hand, the abstraction (b) does not show this property, since there exists an abstract state labeled $pc = 2$ and $\neg(x = 1)$.



To define the notion of "abstraction showing a property" we interpret the labelling of each abstract state s^A as a predicate $\pi(s^A)$ on the concrete variables; for instance, the predicate associated with the initial state of system (b) above is $(pc = 1) \wedge \neg(x = 1)$. Let $T/\mathcal{B}$ be the abstraction of a concrete system T relative to the set of abstract variables $\mathcal{B} = \{B_1, \ldots, B_k\}$, and φ be a state predicate. We say that an abstract state s^A *shows* φ if $\pi(s^A)$ implies φ. We say that $T/\mathcal{B}$ shows $\Box\varphi$, denoted $T/\mathcal{B} \models_{ABS} \Box\varphi$, if all abstract states show φ. The crucial feature of these boolean abstractions, which is true by construction, is that for every computation

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1}, \ldots$$

of the concrete system T, there exists an abstract trace

$$s_0^A \xrightarrow{a_0} s_1^A \xrightarrow{a_1}, \ldots$$

such that for $i = 0, 1, \ldots,$ the labels of the abstract and concrete transitions coincide, and the boolean values of the abstract variables in s_i^A and in s_i coincide. Consequently, boolean abstractions are useful for proving invariants, since

$$T/\mathcal{B} \models_{ABS} \Box\varphi \quad \Rightarrow \quad T \models \Box\varphi.$$

In general, an abstraction relative to a larger set of abstract variables can "show" more properties, because the prover has more information at its disposal when new abstract states are generated, therefore it can eliminate some of them, yielding a finer abstraction. Also, constructing an abstraction with some known invariants of the concrete system can assist in eliminating irrelevant abstract states.

3 The Integration

We introduce our approach for integrating the previously discussed static analysis, theorem proving and abstraction techniques. The general scheme is presented in Fig. 1.

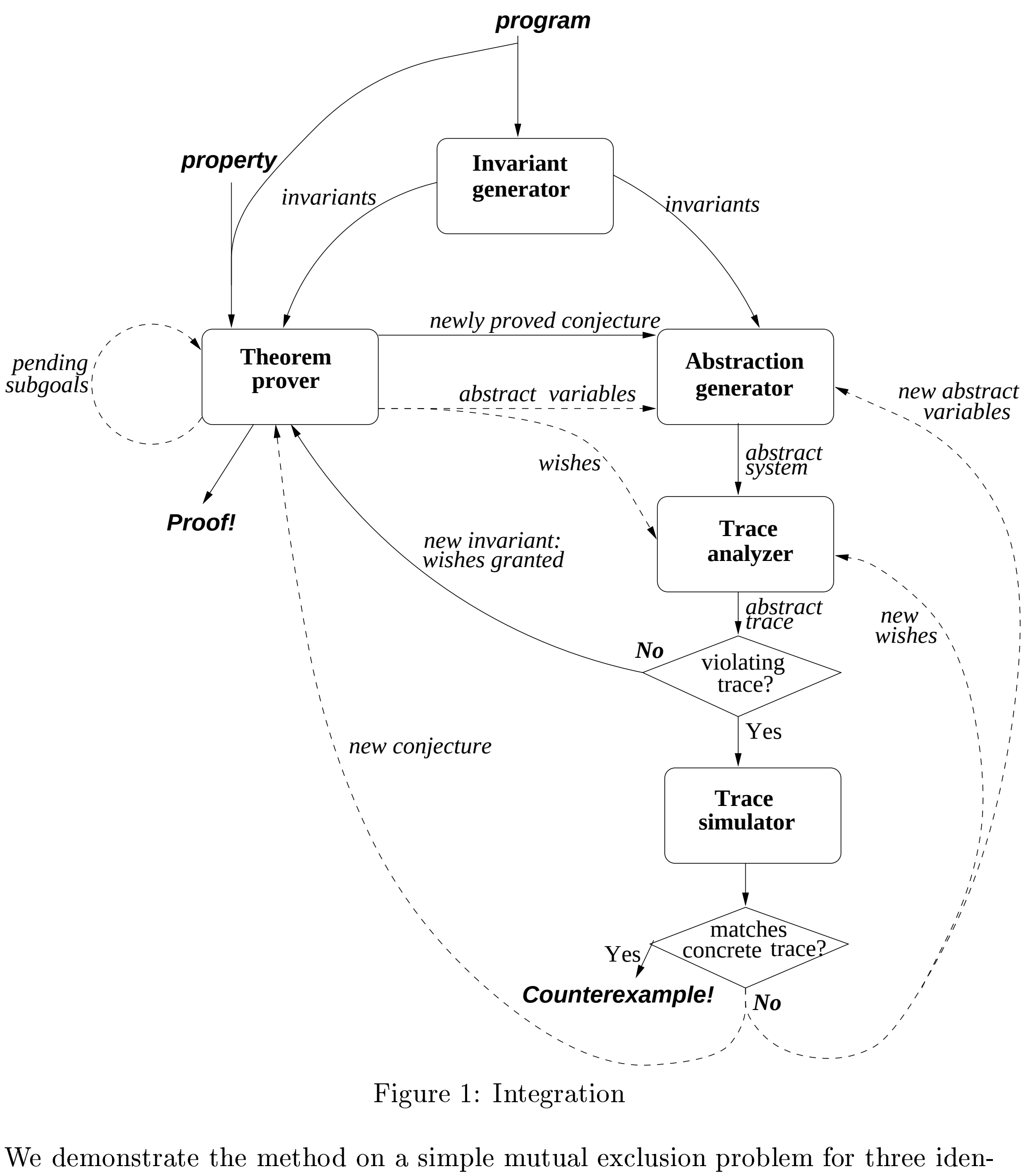
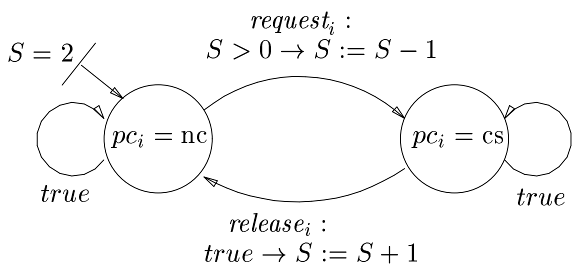


Figure 1: Integration

We demonstrate the method on a simple mutual exclusion problem for three identical processes (illustrated below), in which the semaphore S is a shared variable.



The property to be proved is that it is never the case that all *three* processes are in their critical sections simultaneously; this is expressed as $\Box I$ with

$$I : \neg((pc_1 = cs) \wedge (pc_2 = cs) \wedge (pc_3 = cs)).$$

The first step is to employ the *Invariant generator*. This yields the global invariant: $\Box(S \geq 0)$, which is fed to the *Theorem prover* and to the *Abstraction generator*, since it contains relevant information that may be useful in the sequel.

The next step is to apply theorem proving in trying to prove that I is inductive. In our case, I is not inductive, and therefore the proof is not completed. Rather, we are presented with three (symmetric) pending subgoals, resulting from transitions that do not preserve I. For example, the following subgoal is related to transition $request_3$ when the third process attempts to enter the critical section while the other two processes are already there:

Assuming:

$pc_1 = cs$

$pc_2 = cs$

$S > 0$

Prove:

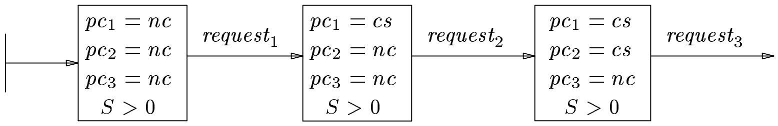
$\neg(pc_3' = cs)$

Obviously, the only way to prove this implication is to show that the assumption is contradictory; but I alone is too weak to prove it. The user now has two alternatives: either to remain in the prover and try to strengthen I, or to try to prove the pending subgoals using an abstraction. User-dependent decisions are represented in the diagram of Fig. 1 by dashed lines. Here, we choose the latter alternative. From the pending subgoals we identify the predicate $(S > 0)$ as a potentially relevant abstract variable and use the *Abstraction generator* to build the abstract system $T/\{(S > 0)\}$. The generated abstract system is then passed to the *Trace analyzer* together with a user-defined *wish*.

A wish is a transition-related state property to be checked on the abstract system which, if shown correct, would enable to eliminate an unproved subgoal. The transition to which a wish refers is that who gave rise to the corresponding subgoal.

Formulating a wish is straightforward. For example, a wish corresponding to the subgoal above is: "for every abstract transition labeled $request_3$, if the origin abstract state is labeled $pc_1 = cs$ and $pc_2 = cs$ then it is also labeled $\neg(S > 0)$".

The role of the *Trace analyzer* is to find an abstract state that violates the wish. If there is no violating state, then the wish is *granted* and this information is passed back to the prover, allowing to complete the corresponding subgoal. In our example, however, there exists a violating abstract state and the *Trace analyzer* returns the following abstract trace (starting from the initial abstract state) leading to it:



This means that either mutual exclusion is not guaranteed by the program, or that the abstraction is too coarse. To decide between these two we must check whether this violating trace can be matched by a concrete computation.

This task is performed by the *Trace simulator*, by simulating the transitions of the violating trace on the concrete system. It checks whether after every transition the valuation of the abstract variables in the concrete and abstract systems coincide. If this is the case, then we have a *counterexample*. Here, it is not the case, since a miss-match is detected in the third abstract state: according to the concrete computation, $S = 0$ should hold, but in the abstract system, $S > 0$ holds. Thus, the abstraction is too coarse. In this situation, the simulator outputs a warning message indicating what "went wrong" in the abstraction; this information is obtained by computing the $\widetilde{pre}$-images of the abstract variables on the violating trace. In our example, the message suggests that the abstraction "believes" that initially $S > 2$ holds.

The user has now two options to pursue. The first is to do another abstraction relative to a larger set of abstract variables (obtained by adding the new ones suggested by the trace simulator as "responsible" for the miss-matches). For example, $S > 2$ is a new relevant abstract variable. The second option is to formulate a *conjecture* and try to prove it in the theorem prover. A conjecture is an auxiliary invariant that would assist in generating a finer abstraction. In our case, an obvious conjecture is $\Box(S \leq 2)$. If it was proved, then taking it into account when the next abstraction is computed would eliminate some abstract traces (e.g., the previous violating trace).

We pursue the latter alternative. The proof of $\Box(S \leq 2)$ does not succeed in one invariant strengthening step. From the new unproved subgoals we extract two new abstract variables: $(S \leq 2)$ and $(S \leq 1)$. We compute the abstract system $T/\{(S > 0), (S \leq 2)(S \leq 1)\}$, which is fine enough to grant our original wishes. Armed with this information the prover eliminates the (original) unproved subgoals and completes the proof of mutual exclusion.

As another example, we consider a version of the alternating bit protocol taken from [GS97] (see Fig. 2 below).

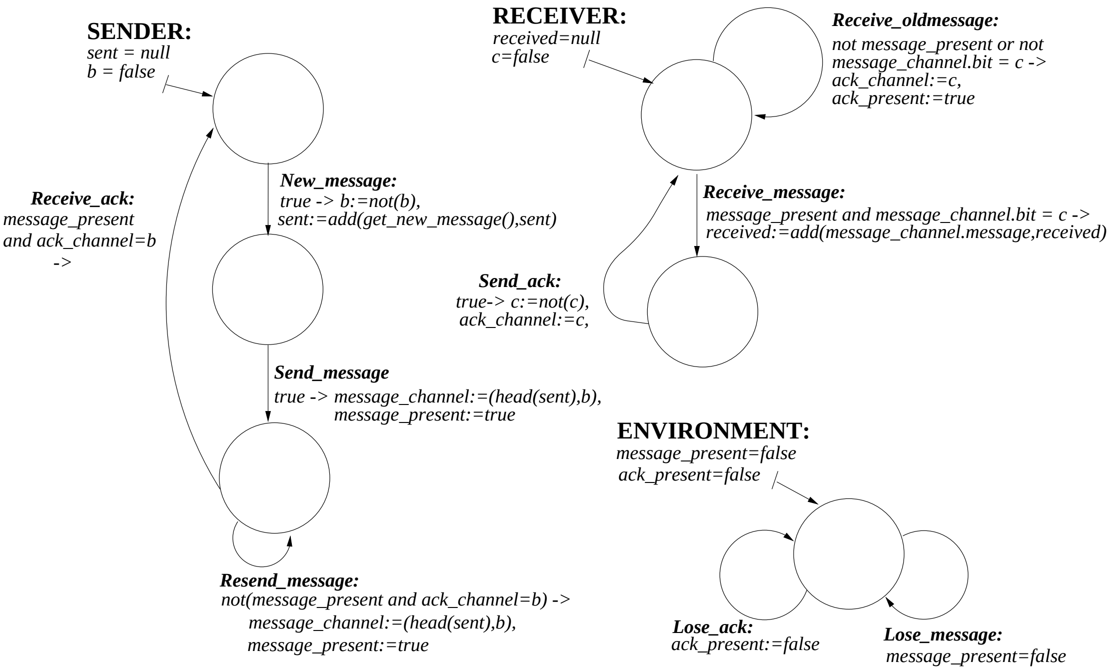


Figure 2: An Alternating bit Protocol.

There are three processes: sender, receiver and environment. The sender generates messages, records them in the *sent* list, then sends them to the receiver over the communication medium *message_channel*. The latter is modeled as a one-place buffer that can hold a message and a bit. The receiver records successfully received messages in the *received* list and sends an acknowledgment through the one-place buffer *ack_channel*. The environment can lose messages and acknowledgements by setting the boolean flags *message_present* and *ack_present* to $False$. This causes the sender/receiver respectively to retransmit. The safety property to be proved, is that the (unbounded) lists *sent* and *received* always differ by at most one message: $\Box(sent = received \vee sent = tail(received))$.

The first step, static analysis, yields two invariants that are fed to the prover and to the abstraction generator. The next step is theorem proving, and since the property is not inductive, the proof is not completed. There are three pending subgoals, all of which are related to transitions that update the *sent*/*received* lists.

For example, we have to prove that at the origin of transition *receive_message*: $sent = tail(received)$. We take this predicate as an abstract variable, and formulate the above as a wish. (We also used two other similar abstract variables and corresponding wishes which are omitted here.)

After the abstraction has been computed, the trace analyzer returns a violating trace in which a *receive_message* transition is taken from the *initial* abstract state. From the trace simulator we get a warning message indicating that the problem occurred because the transition *receive_message* should not have been enabled initially, and that the predicate "responsible" for this is the conjunct $message\_channel.bit = c$ in the guard of the transition.

The obvious choice now is to take this predicate as a new abstract variable and to redo an abstraction. Still, the second abstraction does not grant our wishes; a new violating trace is detected and another abstract variable is suggested by the same mechanism described above. The third abstraction grants all original wishes, and then the prover completes the proof.

In [GS97] the same protocol is analyzed by an abstraction relative to a set of sub-formulas of the guards, and human inspection of the generated abstract system is necessary to conclude that the protocol is indeed correct. Our approach is different: the abstract variables are suggested to the user by the failures of previous proof attempts and abstractions; the analysis of the abstract system is automatic and it issues information to the user; and in the end we obtain a complete rigorous proof.

Our method can be automated in significant proportion. Indeed, all the components in the diagram (Fig. 1) perform automatic tasks, and user intervention is basically limited to choosing between abstraction and theorem proving. In both cases, the user is assisted in providing the relevant abstract variables, wishes and conjectures by the pending subgoals in the prover and by the warning messages issued by the trace simulator. The method is incremental: progress is made in each step, as every relevant abstract variable and conjecture reduces the search space; and the user gains insight of the problem while progressing towards a proof or a counterexample. Finally, we show in the next section that by integrating the components it is possible to prove more properties than by automatic invariant strengthening or automatic abstraction as stand-alones.

4 Integration is More Powerful

We now define the class of safety properties that can be proved to be invariant by the automatic invariant strengthening technique described in Section 2.2. For a transition system T and $n = 0, 1, \ldots$ consider the set

$$INV_n(T) = \{\Box I \mid I \wedge \widetilde{pre}(I) \wedge \ldots \wedge \widetilde{pre}^n(I) \supset \widetilde{pre}^{n+1}(I)\} \tag{2}$$

Definition 4.1 *The class $INV(T)$ of safety properties that can be proven by $\widetilde{pre}$-invariant strengthening is $\bigcup_{n \geq 0} INV_n(T)$.*

Next, we define a particular class of properties that can be shown by the abstraction mechanism described in Section 2.3. Given a state predicate I, we consider the set of predicates

$$AV(I) = \{I, \widetilde{pre}(I), \ldots, \widetilde{pre}^n(I), \ldots,\}$$

Definition 4.2 *The class $ABS(T)$ of safety properties that can be shown by $\widetilde{pre}$-abstraction is the class of properties $\Box I$ for which there exists a finite subset $\mathcal{B} \subset AV(I)$ such that $T/\mathcal{B} \models_{ABS} \Box I$.*

It should be stressed that choosing the abstract variables from $I, \widetilde{pre}(I), \ldots, \widetilde{pre}^n(I)$ is not arbitrary: the guards of transitions, which in many cases allow to generate useful control abstractions [GS97] are just sub-formulas of these predicates.

Note that both $\widetilde{pre}$-invariant strengthening and $\widetilde{pre}$-abstraction are fully automatic techniques. Under the assumption that the same "reasoning power" is used for both $\widetilde{pre}$-invariant strengthening and $\widetilde{pre}$-abstraction (for example, both use the same theorem prover), the following result holds.

Theorem 4.3 *A safety property can be proved by $\widetilde{pre}$-invariant strengthening iff it can be shown by $\widetilde{pre}$-abstraction.*

Proof (sketch). First, for every $n = 0, 1, \ldots,$ define the finite set of predicates $AV_n(I)$ as

$$AV_n(I) \;=\; \{I, \widetilde{pre}(I), \ldots, \widetilde{pre}^n(I)\}.$$

Then, the set $ABS_n(T)$ of safety properties that can be shown by abstraction relative to a subset of $AV_n(I)$ is

$$ABS_n(T) \;=\; \{\Box I \mid \exists n \geq 0,\; \mathcal{B} \subseteq AV_n(I) \; s.t. \; T/\mathcal{B} \models_{ABS} \Box I\}.$$

Thus, by Definition 4.2, $ABS_n(T) = \bigcup_{n \geq 0} ABS_n(T)$. Next, recall that by Definition 4.1, the class of properties that can be proved by $\widetilde{pre}$-invariant strengthening is $\bigcup_{n \geq 0} INV_n(T)$. Finally, it is not difficult to prove that

$$ABS_n(T) \subseteq INV_n(T) \subseteq ABS_{n+1}(T)$$

and the result follows. □

In our method, when trying to prove a safety property $\Box I$, the abstract variables and conjectures are also variants of sub-formulas of $AV(I)$. As is shown in the following example, however, our method is strictly more powerful than the fully automatic techniques of $\widetilde{pre}$-invariant strengthening and of $\widetilde{pre}$-abstractions.

The example is a mutual-exclusion algorithm taken from [BGP97], and is based on the same principle as the well-known Bakery Algorithm: using "tickets" to control access to the critical section. The program is illustrated in Fig. 3: two global variables t_1 and t_2 are used for keeping record of ticket values, and two local variables a and b control the entry to the critical sections.

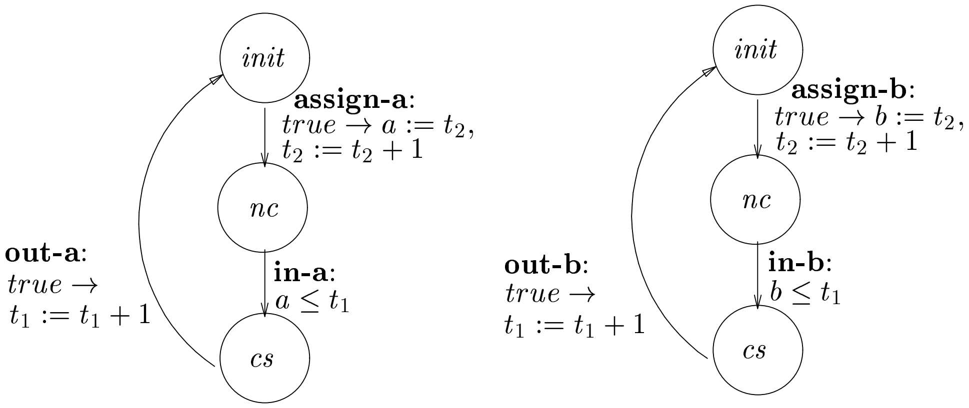


Figure 3: The Ticket Protocol.

The mutual-exclusion property is formulated as $\Box I$ where

$$I : \neg(pc_1 = cs \wedge pc_2 = cs). \tag{3}$$

We employ our method to prove this property. Static analysis generates the local invariants $pc_1 = cs \supset a \leq t_1$ and $pc_2 = cs \supset b \leq t_1$, which are then passed to the theorem prover and to the abstraction generator. Theorem proving yields two unproved subgoals, from which we identify the predicates $(a \leq t_1)$ and $(b \leq t_1)$ as relevant abstract variables (note that these predicates are simply the guards). The wish associated with the transition **in-a** is: "any abstract state labeled $pc_1 = nc$, $pc_2 = cs$ is also labeled $\neg(a \leq t_1)$ ". That is, the guard $(a \leq t_1)$ should prevent the first process from entering its critical section while the second is already there. A similar wish is associated with the transition **in-b**.

The first abstraction does not grant these wishes. A violating trace is produced by the trace analyzer and fed to the trace simulator, which identifies it as not corresponding to a concrete computation; thus, the abstraction is too coarse. By computing $\widetilde{pre}$-images of the abstract variable $(a \leq t_1)$, the system outputs a warning message indicating that the error occurred since the abstraction "believes" that initially: $t_1 \leq t_2 - 1$.

The user now has two options. The first is to add $t_1 \leq t_2 - 1$ as a new abstract variable and do another abstraction. The second is to formulate a conjecture and try to prove it. Choosing the former alternative is reasonable since it would undoubtedly result in a finer abstraction. When it is not too difficult to come up with a conjecture, however, the latter is preferred. This is because a proved (stronger) conjecture usually eliminates more violating traces in further abstractions, and therefore significantly reduces the number of iterations.

In our example this is the case, since it is easy to see that whenever both processes are at their *init* location, the stronger relation $t_1 = t_2$ (rather than $t_1 \leq$

$t_2 - 1$) should hold (this is true initially, and any loop that goes back to the *init* locations increases both t_1 and t_2 by one). So, we formulate the conjecture

$$\Box(pc_1 = \mathit{init} \wedge pc_2 = \mathit{init} \supset t_1 = t_2). \tag{4}$$

In the prover, (4) is proved by three iterations of invariant strengthening. We then use it in a second abstraction (also relative to $(a \leq t_1)$ and $(b \leq t_1)$). This time, the wishes are granted, and the prover can discharge the unproved subgoals and complete the proof.

An interesting conclusion can be drawn from this simple example. While the conjecture (4) can be proved by invariant strengthening, this is *not* the case for the mutual-exclusion property $\Box I$ itself. As shown in [BGP97], backwards analysis for this property does not converge, and hence (3) cannot be proved by $\widetilde{pre}$-invariant strengthening. Therefore, by Theorem 4.3, mutual exclusion cannot be shown by $\widetilde{pre}$-abstraction, either.

Moreover, it is not difficult to prove that even an abstraction relative to any finite set of *sub-formulas* of $\widetilde{pre}$-images of I (such as the guards of the transitions) cannot show (3). The reason for this is that to prove (3) it is important to know when $t_1 = t_2$ holds, but the $\widetilde{pre}$-images of I express only weaker relations between t_1 and t_2. (In the example we have obtained this information by proving the conjecture (4) instead of $\Box(pc_1 = \mathit{init} \wedge pc_2 = \mathit{init} \supset \neg(t_1 \leq t_2 - 1))$ as suggested by the system.)

This demonstrates a typical use of the methodology, in which the detailed feedback from the system together with moderate amount of user ingenuity yields the relevant auxiliary invariant. This is in contrast to an ordinary theorem proving process, in which the user usually has to invest much more effort to come up with suitable auxiliary invariants.

5 Conclusion and Future Work

As an attempt to address the problem of the significant user ingenuity that is required to come up with appropriate auxiliary invariants or with suitable abstraction mappings, we have presented a new methodology for integrating static analysis, theorem proving and abstractions. The key features of our approach are

- It is *incremental*: each step is based on information obtained from failures of previous steps. When the iterative process terminates, it yields a proof or a counterexample.
- It is *goal-directed*: abstractions are guided by a subgoals in a failed proof attempt.
- It is *partially automatic*: each component performs an automatic task, the user chooses which component to invoke at each step and how to apply it.

- User input is *highly guided* by information provided by the system.
- It is *general*, in principle, and not dependent on a particular implementation of the components.

For the experiments described in the paper we have used Pvs [ORS+95] for theorem proving and the Invariant Checker [GS97] for static analysis and abstraction. We are currently building a tool that would incorporate Smv [BCM+92] for trace analysis and simulation, and would also offer a connection to other static analysis tools [HPR97] as well as more general abstraction techniques [BLO98].

Acknowledgments. We wish to thank John Rushby and Natarajan Shankar for valuable comments, Sam Owre for lending us help with Pvs, and Hassen Saidi for assisting us with the Invariant Checker.

References

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142-170, 1992.

[BGP97] T. Bultan, R. Greber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Proc. of the 9th Conference on Computer-Aided Verification, CAV '97*, LNCS 1254, pages 400–411.

[BL] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. To appear in *Formal Methods in System Design.*

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Constructing abstractions of infinite state systems compositionally and automatically. In *Proc. of the 10th Conference on Computer-Aided Verification, CAV '98*, LNCS 1427, pages 319–331.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symposium on Principles of Programming Languages, POPL '77*, pages 238–252.

[CCF+97] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual Version 6.1. Technical Report RT-0203, INRIA, July 1997.

[CU98] M.E. Colòn and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. of the 10th Conference on Computer-Aided Verification, CAV '98*, LNCS 1427, pages 293–304.

[DGG97] D. Dams, R. Gerth and O. Grümberg. Abstract interpretation of reactive systems. *ACM Transactions in Programming Languages and Systems*, 19(2):253-291, 1997.

[GM95] M. Gordon and T.F. Melham. *Introduction to the HOL system.* Cambridge University press, 1994.

[GS96] S. Graf and H. Saidi. Verifying invariants using theorem proving. In *Proc. of the 8th Conference on Computer-Aided Verification, CAV '96*, LNCS 1102, pages 196–207.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of the 9th Conference on Computer-Aided Verification, CAV '97*, LNCS 1254, pages 72–83.

[H91] G.J. Holzmann. *Design and validation of communication protocols.* Prentice Hall, 1991.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[HS96] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe, FME '96*, LNCS 1051, pages 662–681.

[LPY97] K. G. Larsen, P. Petersson, and W. Yi. UPPAAL: Status & Developments. In *Proc. of the 9th Conference on Computer-Aided Verification, CAV '97*, LNCS 1254, pages 456–459.

[ORS+95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107-125, 1995.

[SUM96] H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *Proc. of the 8th Conference on Computer-Aided Verification, CAV '96*, LNCS 1102, pages 208–219.