Slicing Abstractions*

Ingo Brückner[1], Klaus Dräger[2], Bernd Finkbeiner[2], and Heike Wehrheim[3]

[1] Carl von Ossietzky Universität, 26129 Oldenburg, Germany
ingo.brueckner@informatik.uni-oldenburg.de
[2] Universität des Saarlandes, Fachrichtung Informatik, 66123 Saarbrücken, Germany
{draeger,finkbeiner}@cs.uni-sb.de
[3] Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

Abstract. Abstraction and slicing are both techniques for reducing the size of the state space to be inspected during verification. In this paper, we present a new model checking procedure for infinite-state concurrent systems that interleaves automatic abstraction refinement, which splits states according to new predicates obtained by Craig interpolation, with slicing, which removes irrelevant states and transitions from the abstraction. The effects of abstraction and slicing complement each other. As the refinement progresses, the increasing accuracy of the abstract model allows for a more precise slice; the resulting smaller representation gives room for additional predicates in the abstraction. The procedure terminates when an error path in the abstraction can be concretized, which proves that the system is erroneous, or when the slice becomes empty, which proves that the system is correct.

1 Introduction

Much of the progress in automated software verification during the past decade has been driven by the invention of *predicate abstraction* together with methods like Craig interpolation that automatically find the right predicates [1,2,3,4,5,6,7]. Predicate abstraction reduces a potentially infinite state space to the finite set of valuations of a tuple of state predicates. In the *abstraction refinement* loop, one first builds an initial abstract model from some given set of predicates. Then the abstract model is verified, which may result in a proof of correctness (no counter example), a proof of incorrectness (an abstract counter example that can be concretized), or a spurious counter example (an abstract counter example that cannot be concretized). In the latter case, additional predicates are extracted from the proof of spuriousness, and the next iteration of the loop starts with the extended set of predicates.

The advantage of predicate abstraction is its *precision*: when successful, the refinement loop automatically produces a set of predicates that eliminates all

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org).

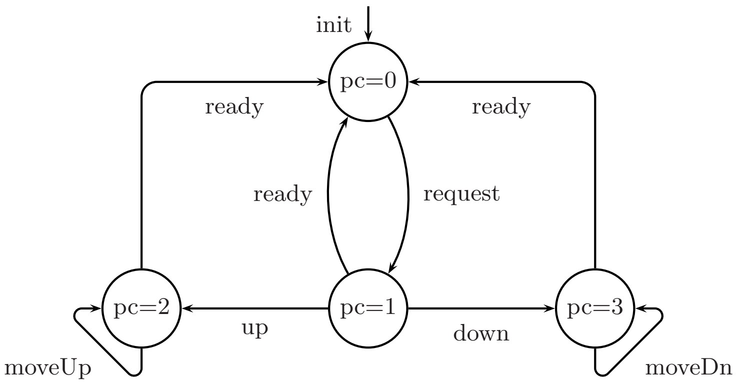


Fig. 1. Control flow graph of a simple elevator

spurious counter examples. On the other hand, the abstract systems generated by predicate abstraction tend to become prohibitively large: the size of the abstract system, and hence the complexity of the verification step of the loop, grows exponentially with the number of predicates.

In this paper, we address this problem by interleaving abstraction refinement steps with *slicing*. Slicing syntactically tracks the dependencies between variables and transitions in a system and completely removes irrelevant parts. While slicing alone cannot be used as a proof technique, it has the advantage that it never increases the size of the state space and may lead to significant reductions.

Figure 1 shows the control flow graph of a simple elevator example, which will be used in the following to illustrate our method. The elevator accepts a request for a certain floor, then moves up or down accordingly, and finally, after reaching the requested floor, is ready for a new request. The transitions of the elevator are specified in Table 1. The system variables include the program counter *pc*, the current floor *current*, the currently requested floor *req*, and a nondeterministic input variable *input* (*input* is constrained to be in the valid range $input \leq Max$ when the elevator is ready to receive its next request). We verify the correctness of the elevator by showing that the error condition $current > Max$ is never satisfied.

The verification of the elevator is shown in Figures 2 and 3. (We will refer to these figures throughout the paper to illustrate the individual steps.) Our procedure maintains an explicit representation of the abstract model. Rather than requiring, as in many other approaches, a simulation preorder between system and abstraction, we call an abstraction *sound* if the system is correct iff the abstraction has no concretizable error path. The process starts with a default abstraction shown as Step 1 of Figure 2: there are four abstract nodes, corresponding to the four different evaluations of *init* (a predicate characterizing the initial states) and *error* (a predicate characterizing error states). The edges in the abstraction allow for all possible paths between *init* and *error* states that are "minimal" in the sense that they do not visit a second *init* or *error* state. The advantage of this restriction is that redundant computation segments,

Table 1. Initial condition, error condition, and transitions of a simple elevator

init	$pc{=}0 \wedge current{\leq}Max \wedge input{\leq}Max$
error	$current{>}Max$
request	$pc{=}0 \wedge pc'{=}1 \wedge current'{=}current \wedge req'{=}input$
ready	$pc \geq 1 \wedge req{=}current \wedge pc'{=}0 \wedge current'{=}current \wedge req'{=}req \wedge input'{\leq}Max$
up	$pc{=}1 \wedge req > current \wedge pc'{=}2 \wedge current'{=}current \wedge req'{=}req$
down	$pc{=}1 \wedge req < current \wedge pc'{=}3 \wedge current'{=}current \wedge req'{=}req$
moveUp	$pc{=}2 \wedge req > current \wedge pc'{=}2 \wedge current'{=}current + 1 \wedge req'{=}req$
moveDn	$pc{=}3 \wedge req < current \wedge pc'{=}3 \wedge current'{=}current - 1 \wedge req'{=}req$

such as any downward movement of the elevator (which needs to be followed by an upward movement before an error can possibly be reached), are quickly eliminated from the abstraction.

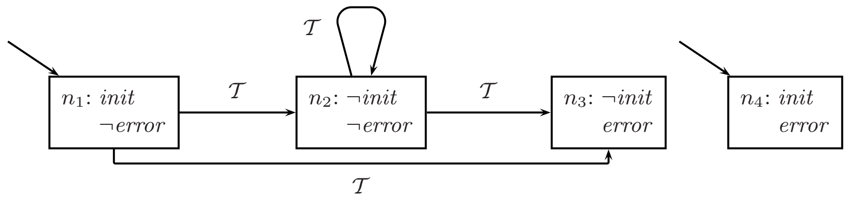
Predicates, obtained by Craig interpolation, are used to refine the abstraction locally, i.e., by splitting individual nodes. In parallel, slicing reduces the size of the abstraction by dropping irrelevant states and transitions from the model. (In Figures 2 and 3, components that are eliminated by slicing are shown in dashed lines.) The effects of abstraction and slicing complement each other. As the refinement progresses, the increasing accuracy of the abstract model allows for a more precise slice; the resulting reduction gives room for additional predicates in the abstraction. In the example, the procedure terminates after Step 6, when the slice becomes empty, proving that the system is correct.

2 Related Work

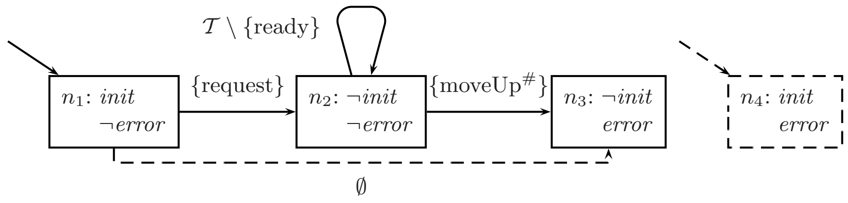
Abstraction. There is a rich literature on predicate abstraction and the abstraction refinement loop [1,2,3,4,5]. The key difference between our approach and classic predicate abstraction is that we use new predicates to split individual nodes, while predicate abstraction interprets every predicate in every abstract state. Our approach can be seen as a generalization of *lazy abstraction* [6,7], which incrementally refines the state space with new predicates as the control flow graph is searched in a forward manner to find an error path. New predicates in lazy abstraction only affect the subgraph reachable from the current node. Lazy abstraction thus exploits locality in branches of the control flow graph while our approach exploits locality in individual nodes of the abstraction.

Our abstraction process is similar to *deductive model checking* [8], which also refines an explicit abstraction by splitting individual nodes. While we only handle simple error conditions, deductive model checking provides rules for full linear-time temporal logic. The key difference is that deductive model checking is only partly automated and in particular relies on the user to select the nodes and predicates for splitting.

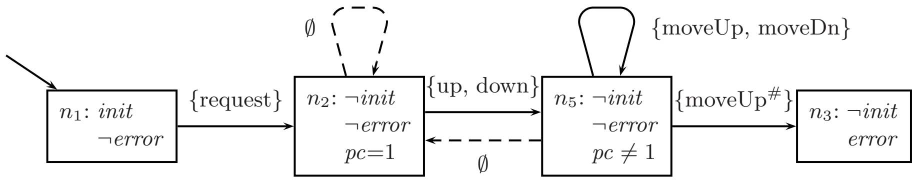
Slicing. Program slicing, introduced by Weiser [9], is a static analysis technique widely used in debugging, program comprehension, testing, and maintenance.


$\mathcal{T}$
n_1: *init* $\neg$*error*
$\mathcal{T}$
n_2: $\neg$*init* $\neg$*error*
$\mathcal{T}$
n_3: $\neg$*init* *error*
n_4: *init* *error*
$\mathcal{T}$

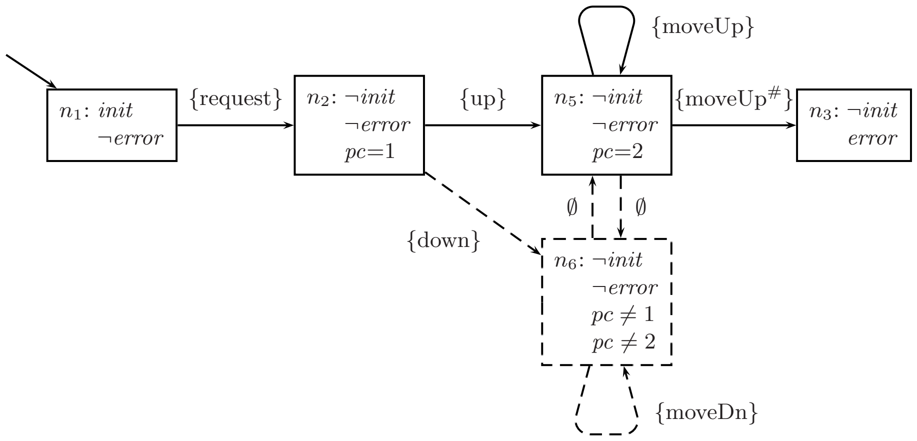
Step 1: Initial abstraction.


$\mathcal{T} \setminus$ {ready}
n_1: *init* $\neg$*error*
{request}
n_2: $\neg$*init* $\neg$*error*
{moveUp$^\#$}
n_3: $\neg$*init* *error*
n_4: *init* *error*
$\emptyset$

Step 2: After slicing. The transition relation moveUp on edge (n_2, n_3) simplifies to moveUp$^\#$ = pc=2 $\wedge$ req > $current$ $\wedge$ pc'=2 $\wedge$ $current'$=$current$ + 1.

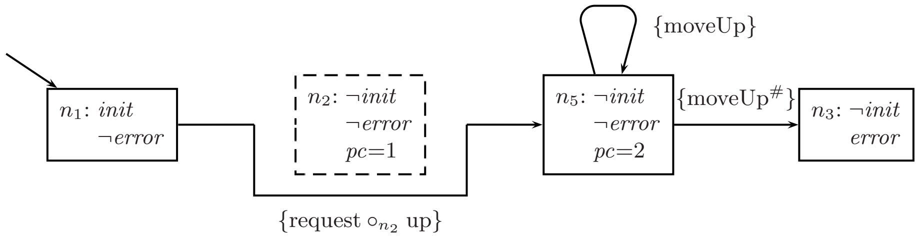

$\emptyset$
{moveUp, moveDn}
n_1: *init* $\neg$*error*
{request}
n_2: $\neg$*init* $\neg$*error* pc=1
{up, down}
$\emptyset$
n_5: $\neg$*init* $\neg$*error* $pc \neq 1$
{moveUp$^\#$}
n_3: $\neg$*init* *error*

Step 3: After splitting node n_2 with predicate pc=1 and slicing.


{moveUp}
n_1: *init* $\neg$*error*
{request}
n_2: $\neg$*init* $\neg$*error* pc=1
{up}
n_5: $\neg$*init* $\neg$*error* pc=2
{moveUp$^\#$}
n_3: $\neg$*init* *error*
$\emptyset$
$\emptyset$
{down}
n_6: $\neg$*init* $\neg$*error* $pc \neq 1$ $pc \neq 2$
{moveDn}

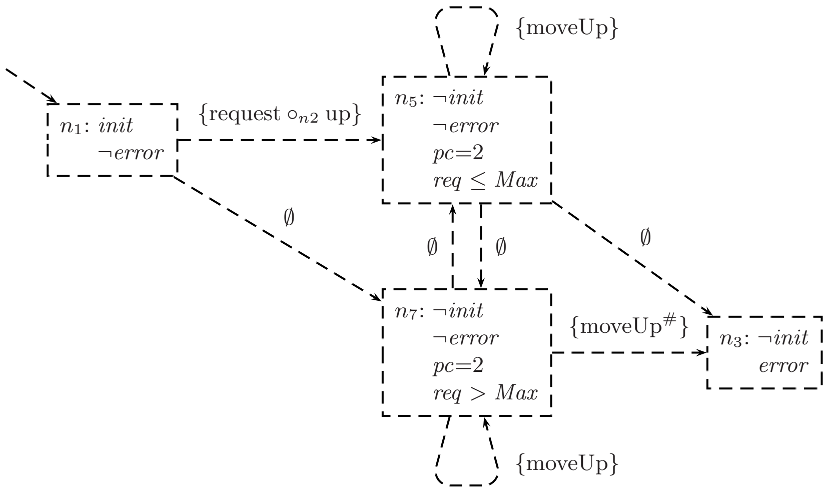
Step 4: After splitting node n_5 with predicate pc=2 and slicing.

Fig. 2. Steps 1–4 of the verification of the simple elevator. Components shown in dashed lines are deleted in the slice.



n_1: *init* $\neg$*error*
n_2: $\neg$*init* $\neg$*error* *pc*=1
n_5: $\neg$*init* $\neg$*error* *pc*=2
n_3: $\neg$*init* *error*
{moveUp}
{moveUp$^{\#}$}
{request $\circ_{n_2}$ up}

Step 5: Node n_2 is bypassed via transition relation request $\circ_{n_2}$ up.



n_1: *init* $\neg$*error*
n_5: $\neg$*init* $\neg$*error* *pc*=2 *req* $\leq$ *Max*
n_7: $\neg$*init* $\neg$*error* *pc*=2 *req* $>$ *Max*
n_3: $\neg$*init* *error*
{request $\circ_{n_2}$ up}
{moveUp}
{moveUp$^{\#}$}
$\emptyset$

Step 6: After splitting node n_5 with predicate *req* $\leq$ *Max* and slicing.

Fig. 3. Steps 5 and 6 of the verification of the simple elevator. Components shown in dashed lines are deleted in the slice. Since the slice after Step 6 is empty, the system is correct.

Essentially, slicing extracts the parts of a program which might affect some given slicing criterion (e.g. a variable at some point). Slicing has become one of the standard reduction techniques in finite-state model checking (for instance in SAL [10], Bandera [11], Promela [12], IF [13]). More recently, slicing has been used in automated abstraction refinement as a preprocessing step on abstract error paths (thus analyzing individual paths, not the full abstraction). *Path slicing* [14] removes irrelevant parts of the abstract error path before the path is passed to the theorem prover to verify if the path can be concretized.

Usually, the slice is determined by a dependency analysis on the control flow graph of the program. A more refined technique, taking additional information about the property under interest into account, is *conditioned slicing* [15]. Here, an assumption about the initial (forward conditioning) or final states (backward conditioning) is added in the form of a predicate, and slicing then only keeps the statements which can be executed from an initial state or which lead to a final state satisfying the predicate.

Closest to our work is the backward conditioning approach of [16] (used for program comprehension, not verification). Backward conditioning proceeds by a symbolic execution of the program and the use of a theorem prover to prune the execution paths which do not lead into a desired final state. The analysis is however always carried out on the concrete program, not its abstraction, and the technique will – due to its objective of program comprehension – preserve *all* paths to the given final states. A use of conditioned slicing in verification can be found in [17], where the condition is extracted from a temporal logic formula of the form $G(p \rightarrow q)$. The predicate p is used as a condition for forward conditioning, the technique is then building a refined program dependence graph (based on the control flow graph). A conditioning method operating on an abstraction of the program is presented in [18]. On this abstraction it can be determined under which conditions one statement might affect another (while for verification we need to find out whether some condition might hold at all or not).

3 Preliminaries: Transition Systems

We use a general representation of concurrent systems as transition systems, which can be defined using an assertion language based on first-order logic. In the following we denote the set of first-order formulas over a set of variables $\mathcal{V}$ by $Ass(\mathcal{V})$. A *transition system* $\mathcal{S} = \langle V, init, \mathcal{T} \rangle$ consists of the following components:

- V: a finite set of system *variables*. We define for each system variable $v \in V$ a primed variable $v' \in V'$, which indicates the value of v in the next state. We call the set $Ass(V)$ of assertions over the system variables the set of *state predicates* and the set $Ass(V \cup V')$ of assertions over the system variables and the primed variables the set of *transition relations*. For a state predicate φ, let φ' denote the assertion where each variable v is replaced by v'.
- $init(V)$: the *initial condition*, a state predicate characterizing all states in which the computation of the system can start.
- $\mathcal{T}$: the *transition set*. Each transition $\tau(V, V') \in \mathcal{T}$ is a conjunction $\tau(V, V') = \bigwedge_i g_i(V) \ \wedge \ \bigwedge_i t_i(V, V')$ of *guards* g_i and *transition relations* t_i. In the special case where, for a given set W of variables, τ is of the form $\tau(V, V') = \bigwedge_i g_i(V) \ \wedge \ \bigwedge_{v \in W}(v' = e_v(V))$, i.e., each variable in W is assigned a value defined over V, we say that τ is a *guarded W-assignment*. We assume that $\mathcal{T}$ always contains the idling transition $\tau_{idle} = \bigwedge_{v \in V} v = v'$.

A *state* of $\mathcal{S}$ is a valuation of the system variables V. A *run* is an infinite alternating sequence $s_0, \tau_0, s_1, \tau_1, \ldots$ of states and transitions such that $init(s_0)$ holds and for all positions $i \geq 0$, $\tau_i(s_i, s_{i+1})$ holds.

We assume that the correctness criterion for $\mathcal{S}$ is given as an *error condition* $error(V)$, a predicate which characterizes all error states. We say $\mathcal{S}$ is *correct* if there is no run $s_0, \tau_0, s_1, \ldots$ of $\mathcal{S}$ that has a position $i \geq 0$ such that $error(s_i)$ holds.

4 Abstraction

Our abstractions are graphs where the nodes are labeled with sets of predicates and the edges are labeled with sets of transition relations.

Definition 1. *An* abstraction $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ *of a transition system* $S = \langle V, \mathit{init}, \mathcal{T} \rangle$ *consists of the following components:*

- *a finite set N of nodes,*
- *a set $E \subseteq N \times N$ of edges,*
- *a labeling $\nu : N \to 2^{Ass(V)}$ of nodes with sets of predicates, and*
- *a labeling $\eta : E \to 2^{Ass(V,V')}$ of edges with sets of transition relations.*

A node $n \in N$ of the abstraction is an *initial* node if its label $\nu(n)$ contains the initial condition *init*, and an *error node* if its label $\nu(n)$ contains the error condition *error*. In the following, let $I = \{n \in N \mid \mathit{init} \in \nu(n)\}$ denote the set of initial nodes, and $F = \{n \in N \mid \mathit{error} \in \nu(n)\}$ the set of error nodes.

A *path* of an abstraction is a finite alternating sequence $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$ of nodes and transitions such that for all $0 \leq i < k, \tau_i \in \eta(n_i, n_{i+1})$. An *error path* is a path $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$ such that $n_0 \in I$ is an initial node and $n_k \in F$ is an error node.

An abstract path $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$ is *concretizable* in $\mathcal{S}$ if there exists a finite sequence of states $s_0, s_1, \ldots, s_k$ such that for every position $0 \leq i \leq k$ and every state predicate $q \in \nu(n_i)$, $q(s_i)$ holds and for every position $0 \leq i < k$, $\tau_i(s_i, s_{i+1})$ holds. We call the alternating sequence of system states and transitions $s_0, \tau_0, s_1, \tau_1, \ldots, \tau_{k-1}, s_k$ the *concretization* of $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$. An abstract error path that is not concretizable is called *spurious*. An abstraction $\mathcal{A}$ of a transition system $\mathcal{S}$ is *sound* if there exists a concretizable error path in $\mathcal{A}$ if and only if $\mathcal{S}$ is not correct. Our abstraction refinement procedure starts with a sound initial abstraction and then preserves soundness in each transformation.

Definition 2. *The* initial abstraction $\mathcal{A}_0 = \langle N, E, \nu, \eta \rangle$ *of a transition system* $\mathcal{S} = \langle V, \mathit{init}, \mathcal{T} \rangle$ *consists of the following components:*

- $N = \{ie, \bar{i}e, i\bar{e}, \overline{ie}\}$
- $E = \{(i\bar{e}, \overline{ie}), (i\bar{e}, \bar{i}e), (\overline{ie}, \bar{i}e), (\overline{ie}, \overline{ie})\}$
- $\nu : ie \mapsto \{\mathit{init}, \mathit{error}\}, \bar{i}e \mapsto \{\neg\mathit{init}, \mathit{error}\}, i\bar{e} \mapsto \{\mathit{init}, \neg\mathit{error}\}, \overline{ie} \mapsto \{\neg\mathit{init}, \neg\mathit{error}\}$,
- $\eta : e \mapsto \mathcal{T}$ *for all* $e \in E$.

The initial abstraction is shown as Step 1 of Figure 2. As explained in the introduction, no concretization of a path in the abstraction visits an initial or error state twice. This is consistent with our definition of soundness, which only requires the existence of *some* concretizable error path. Given an error path that visits initial or error nodes multiple times, we can always construct an error path that visits both only once, by considering the segment between the last initial node and the first error node.

Proposition 1. *The initial abstraction $\mathcal{A}_0$ of a transition system S is sound.*

Proof. By definition, the concretization of an error path of $\mathcal{A}_0$ is the prefix of a run of $\mathcal{S}$ that leads to a state that satisfies the error condition. Hence, the existence of a concretizable error path implies that $\mathcal{S}$ is not correct. Suppose, on the other hand, that S is not correct, i.e., there exists a run $s_0, \tau_0, s_1, \tau_1 \dots$ such that $\mathit{error}(s_k)$ holds for some $k \in \mathbb{N}$. Let i be the greatest index between 0 and k such that $\mathit{init}(s_i)$ holds, and let j be the smallest index between i and k such that $\mathit{error}(j)$ holds. The subsequence $s_i, \tau_i, s_{i+1}, \tau_{i+1}, \dots, s_k$ defines a concretizable abstract path $n_i, \tau_i, n_{i+1}, \tau_{i+1}, \dots, \tau_{k-1}, n_k$ as follows: for all $l = i, \dots, k$, $n_l = ie \Leftrightarrow \mathit{init}(s_l) \wedge \mathit{error}(s_l)$; $n_l = \bar{i}e \Leftrightarrow \neg \mathit{init}(s_l) \wedge \mathit{error}(s_l)$; $n_l = i\bar{e} \Leftrightarrow \mathit{init}(s_l) \wedge \neg \mathit{error}(s_l)$; $n_l = \overline{ie} \Leftrightarrow \neg \mathit{init}(s_l) \wedge \neg \mathit{error}(s_l)$. Since $n_i \in I$ and $n_k \in F$, $n_i, \tau_i, n_{i+1}, \tau_{i+1}, \dots, \tau_{k-1}, n_k$ is an error path. □

Soundness of the abstraction is preserved by all of the following transformations on the abstraction. Due to lack of space, we prove this only for the most complex type of simplification, namely Simplify Transition.

5 Slicing

Slicing removes irrelevant components from the abstraction. We define four different types of slicing operations: Elimination of transitions, elimination of nodes, simplification of transition relations, and the introduction of bypass transitions.

5.1 Eliminating Transitions

The abstraction may contain transitions that are irrelevant because the predicates on the source and target nodes of the edge contradict the transition relation. Such transitions are eliminated in the slice:

Inconsistent Transition. Let $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ be an abstraction that contains a transition relation $\tau \in \eta(m, n)$ on some edge $(m, n) \in E$ such that the formula $\bigwedge_{q \in \nu(m)} q \wedge \tau \wedge \bigwedge_{q \in \nu(n)} q'$ is unsatisfiable. We remove τ, resulting in the abstraction $\mathcal{A}' = \langle N, E, \nu, \eta' \rangle$, where $\eta'(m, n) = \eta(m, n) \smallsetminus \{\tau\}$ and $\eta'(e) = \eta(e)$ for $e \neq (m, n)$.

Empty Edges. The removal of transition relations may result in edges with empty labels. Such edges can be removed, transforming $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ into the abstraction $\mathcal{A}' = \langle N, E', \nu, \eta|_{E'} \rangle$, where $E' = \{e \in E \mid \eta(e) \neq \emptyset\}$.

In Step 2 of the elevator example, all transition relations on the edge between nodes n_1 and n_3 are contradicted by the predicates on the nodes: there is no transition that leads directly from an initial state to an error state (the only transition enabled in the initial state is request, which does not modify *current*). As a result, all transitions are removed from the label, and the empty edge is removed from the abstraction.

5.2 Eliminating Nodes

Nodes are removed from the abstraction if they are either labeled with an inconsistent combination of predicates or do not occur on any error paths.

Inconsistent Node. Let $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ be an abstraction that contains a node $n \in N$ such that $\bigwedge_{q \in \nu(n)} q$ is unsatisfiable. We remove n, resulting in the abstraction $\mathcal{A}' = \langle N', E', \nu|_{N'}, \eta|_{E'} \rangle$, where $N' = N \setminus \{n\}$ and $E' = E \cap (N' \times N')$.

Unreachable Node. Let $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ be an abstraction that contains a node $n \in N$ which is unreachable from initial nodes or from which no error node can be reached. We remove n, resulting in $\mathcal{A}' = \langle N', E', \nu|_{N'}, \eta|_{E'} \rangle$, where $N' = N \setminus \{n\}$ and $E' = E \cap (N' \times N')$.

In Step 2 of the elevator example, the inconsistent node n_4 is removed (the conjunction $\mathit{init} \wedge \mathit{error}$ is unsatisfiable). Unreachable nodes are removed in Step 4 (node n_6), Step 5 (node n_2), and Step 6 (the entire abstraction).

5.3 Simplifying Transition Relations

The next slicing mechanism removes constraints from transition relations that are irrelevant for the existence of a concretizable error path. For this purpose we assign to each node $n \in N$ a set of *live* variables $L(n) \subseteq V$, containing all variables whose value may possibly affect the existence of a concretizable path from n to the error.

As usual in slicing, the set of live variables is computed by a fixpoint computation. Initially, the live variables of a node $n \in N$ are those appearing in its labeling $\nu(n)$ and in the enabling conditions of the transitions on outgoing edges: $L_0(n) = \mathit{vars}(\nu(n)) \cup \bigcup_{(n,m) \in E, \tau \in \eta(n,m)} \mathit{vars}(\mathit{enabled}(\tau))$.

Then, this labeling is updated according to dependencies through transition relations on edges. For a predicate q we let $\mathit{vars}(q)$ denote the set of its free variables. For a transition relation τ and a set of variables X we let $\mathit{depend}_\tau(X)$ denote the set of variables that potentially influence the value of variables in X when τ is taken: for $\tau = \bigwedge_i g_i(V) \ \wedge \ \bigwedge_i t_i(V, V')$, $\mathit{depend}_\tau(X) = W \cap V$, where W is the smallest set of variables such that $X' \subseteq W$, for all i, $\mathit{vars}(g_i) \subseteq W$, and for all i with $\mathit{vars}(t_i) \cap W \neq \emptyset$, $\mathit{vars}(t_i) \subseteq W$. The labeling is updated as follows until a fixpoint is reached: $L_{i+1}(n) = L_i(n) \cup \bigcup_{(n,m) \in E, \tau \in \eta(n,m)} \mathit{depend}_\tau(L_i(m))$.

Given the set of live variables for all nodes, we can simplify the transition relations by eliminating constraints over irrelevant variables. We assume that the conjunction of all such constraints is satisfiable (which can be achieved by a prior application of transformation Inconsistent Transition).

Simplify Transition. Let $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ be an abstraction and let $L(n) \subseteq V$ indicate the set of live variables for each node n. The simplification $\mathit{simplify}(\tau, m, n)$ of a transition τ on an edge $(m, n) \in E$ is obtained by removing from τ all conjuncts ϕ with $\mathit{vars}(\phi) \cap (L(m) \cup L(n)') = \emptyset$. In the special case of a guarded W-assignment τ, the simplification $\mathit{simplify}(\tau, m, n)$ is obtained by removing from τ all conjuncts $v' = e_v(V)$ with $v \notin L(n)$. Simplifying all transitions results in the new abstraction $\mathcal{A}' = \langle N, E, \nu, \eta' \rangle$ where $\eta'(m, n) = \{\mathit{simplify}(\tau, m, n) \mid \tau \in \eta(m, n)\}$ for all $(m, n) \in E$.

In Step 2 of the elevator example, node n_3 is labeled with the set $\{\mathit{pc}, \mathit{current}, \mathit{input}\}$ and nodes n_1 and n_2 are labeled with the full set of variables. As a result, the transition relation moveUp on the edge from n_2 to n_3 is simplified to moveUp$^{\#}$ by dropping the conjunct $\mathit{req}'{=}\mathit{req}$.

Proposition 2. *Let $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ be a sound abstraction of a transition system $\mathcal{S} = \langle V, \mathit{init}, \mathcal{T} \rangle$, and let $\mathcal{A}'$ be the result of applying* Simplify Transition. *Then $\mathcal{A}'$ is again a sound abstraction.*

Proof. We show that $\mathcal{A}$ has a concretizable error path iff $\mathcal{A}'$ has. The implication from $\mathcal{A}$ to $\mathcal{A}'$ is straightforward since *simplify* eliminates conjuncts from transition relations and thus every concretization of an error path in $\mathcal{A}$ is a concretization of the corresponding modified error path in $\mathcal{A}'$.

For the reverse direction assume $n_0, \tau_0, n_1, \tau_1, \ldots, n_k$ to be a concretizable error path in $\mathcal{A}'$ with concretization $s_0, \ldots, s_k$. Let $\widehat{\tau_i}$ be the corresponding non-simplified version of τ_i in $\mathcal{A}$. We inductively construct a concretization $\widehat{s_0}, \ldots, \widehat{s_k}$ of $n_0, \widehat{\tau_0}, n_1, \widehat{\tau_1}, \ldots, n_k$. For a state s and a set of variables $X \subseteq V$, we write $s|_X$ to stand for the valuation s restricted to X. We use the operator $\oplus$ to conjoin valuations over disjoint sets of variables. The construction of the concretization starts with $\widehat{s_0} = s_0$. $\widehat{s_0}$ can be written as $s_0|_{L(n_0)} \oplus t_0$ for some valuation t_0 of variables in $V \setminus L(n_0)$. Then we set $\widehat{s_1}$ to $s_1|_{L(n_1)} \oplus t_1$, where t_1 is a valuation of $V \setminus L(n_1)$ such that $\phi(t_0, t_1)$ for all removed conjuncts ϕ of $\widehat{\tau_0}$. Such a t_1 exists since we assumed that the conjunction of all ϕ is satisfiable and ϕ furthermore contains no variables from $\mathit{enabled}(\widehat{\tau_0})$. Then $\widehat{\tau_0}(\widehat{s_0}, \widehat{s_1})$ since ϕ does not constrain variables in $L(n_1)$ (definition of *depends*) and the enabledness of $\widehat{\tau_0}$ is independent of ϕ ($\mathit{vars}(\mathit{enabled}(\widehat{\tau_0})) \subseteq L(n_0)$). This construction can similarly be continued for all states. $\square$

5.4 Bypass Transitions

The following construction allows us to bypass (and, as a consequence, often eliminate) nodes in the abstraction. For a node n with an incoming transition τ_1 and an outgoing transition τ_2, we define the bypass relation $(\tau_1 \circ_n \tau_2)(V, V') = \exists V'' \,.\, \tau_1(V, V'') \wedge \nu(n)(V'') \wedge \tau_2(V'', V')$. If $W = L(n)$ is the set of live variables of n and τ_1 is a guarded W-assignment $\tau_1(V, V') = \bigwedge_i g_i(V) \wedge \bigwedge_{v \in W}(v' = e_v(V))$, then $\tau_1 \circ_n \tau_2$ can be simplified to $(\tau_1 \circ_n \tau_2)(V, V') = \bigwedge_i g_i(V) \wedge \nu(n)[e_v/v](V) \wedge \tau_2[e_v/v](V, V')$.

Bypass Transition. Let $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ be an abstraction and let $\tau \in \eta(m, n)$ be a transition on some edge $(m, n) \in E$. Transition τ can be modified to bypass node n, resulting in the new abstraction $\mathcal{A}' = \langle N, E', \nu, \eta' \rangle$, where $E' = E \cup \{(m, n') \mid (n, n') \in E\}$, $\eta'(m, n) = \eta(m, n) \setminus \{\tau\}$ and $\eta'(m, n') = \eta(m, n') \cup \{\tau \circ_n \tau_2 \mid \tau_2 \in \eta(n, n')\}$.

In Step 5 of the elevator example, node n_2 is bypassed via request $\circ_{n_2}$ up. As a result, n_2 becomes unreachable and is eliminated.

6 Abstraction Refinement

We first introduce the refinement step for a given predicate and node and then discuss how both can be obtained automatically by error path analysis.

6.1 Node Splitting

Given some new predicate q, we split an abstract node labeled φ into two new nodes, one labeled $\varphi \cup \{q\}$, the other $\varphi \cup \{\neg q\}$.

Node split. Let $\mathcal{A} = \langle N, E, \nu, \eta \rangle$ be an abstraction of a transition system $\mathcal{S} = \langle V, \mathit{init}, \mathcal{T} \rangle$, and let $n \in N$ be some abstract node and $q(V)$ some predicate. The *node split* of $\mathcal{A}$ with respect to n and q is the new abstraction $\mathcal{A}' = \langle N', E', \nu', \eta' \rangle$, where

- $N' = N \cup \{n'\}$ where n' is a fresh node $n' \notin N$;
- $E' = \bigcup_{e \in E} \mathit{edgesplit}(e)$, where

$$\mathit{edgesplit}(e) = \begin{cases} \{(n,n),(n,n'),(n',n),(n',n')\} & \text{if } e = (n,n), \\ \{(m,n),(m,n')\} & \text{if } e = (m,n), m \neq n, \\ \{(n,m),(n',m)\} & \text{if } e = (n,m), m \neq n, \text{ and} \\ \{e\} & \text{otherwise,} \end{cases}$$

- $\nu'(m) = \begin{cases} \nu(n) \cup \{q\} & \text{if } m = n \\ \nu(n) \cup \{\neg q\} & \text{if } m = n', \text{ and} \\ \nu(m) & \text{otherwise;} \end{cases}$
- $\eta'(e') = \eta(e)$ for all $e' \in \mathit{edgesplit}(e)$

The elevator example involves several node splits. For instance, in Step 3, the split of node n_2 with predicate $pc = 1$ adds the new node n_5.

6.2 Error Path Analysis

The verification process terminates as soon as the abstraction has a concretizable error path (in which case the system is incorrect) or no error paths at all (in which case the system is correct). The refinement process is therefore driven by the analysis of spurious error paths.

Our technique is based on *Craig interpolation.* For a given pair of formulas $\varphi(X)$ and $\psi(Y)$, such that $\varphi \wedge \psi$ is unsatisfiable, a Craig interpolant $\Upsilon(X \cap Y)$ is a formula over the variables common to φ and ψ such that Υ is implied by φ and $\Upsilon \wedge \psi$ is unsatisfiable. Craig interpolants can be automatically generated for a number of theories, including systems of linear inequalities over the reals combined with uninterpreted function symbols [19].

In order to obtain the new predicate, we use a variation of a standard error path *cutting* technique [20] from predicate abstraction, which splits the path into two subsequences such that the new predicate is an interpolant for the first-order formulas corresponding to the first and second parts. To ensure that the new predicate affects as many error paths as possible, we focus on minimal spurious sub-paths:

For a spurious error path $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$, we call a sub-path $n_i, \tau_i, n_{i+1}, \tau_{i+1}, \ldots, \tau_{j-1}, n_j$ with $0 \leq i < j \leq k$ *minimal* if the sub-path is not concretizable but both $n_{i+1}, \tau_{i+1}, \ldots, \tau_{j-1}, n_j$ and $n_i, \tau_{i+1}, \ldots, n_{j-1}$ are concretizable.

We translate error paths to first-order formulas in the following way. Let, for each $i \in \mathbb{N}$, V_i be a set of fresh variables such that for each $v \in V$, V_i contains a corresponding fresh variable $v_i \in V_i$. Given a finite path $\boldsymbol{p} = n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$ in an abstraction $\mathcal{A}$ (such that $\tau_i \in \eta(n_i, n_{i+1})$ for all $0 \leq i < k$), we define two first-order formulas

$$\Gamma_1(\boldsymbol{p}) = \nu(n_0)(V_0) \wedge \tau_0(V_0, V_1) \wedge \nu(n_1)(V_1) \wedge \tau_1(V_1, V_2) \wedge \ldots \wedge \nu(n_{k-1})(V_{k-1}),$$
$$\Gamma_2(\boldsymbol{p}) = \tau_{k-1}(V_{k-1}, V_k) \wedge \nu(n_k)(V_k).$$

We analyze a given spurious error path $n_0, \tau_0, n_1, \tau_1, \ldots, \tau_{k-1}, n_k$ in two steps:

1. We find a minimal sub-path $\boldsymbol{p} = n_i, \tau_i, n_{i+1}, \tau_{i+1}, \ldots, \tau_{j-1}, n_j$. This determines the node $n = n_{j-1}$ which will be split.
2. We compute the interpolant of $\Gamma_1(\boldsymbol{p})$ and $\Gamma_2(\boldsymbol{p})$. The interpolant $\Upsilon(V_{j-1})$ defines the new predicate $q = \Upsilon(V)$ on which we split node n.

After Step 2 of the elevator example, we obtain the abstract error path $\boldsymbol{p} = n_1, \text{request}, n_2, \text{moveUp}^{\#}, n_3$. The error path is minimal, since both $n_1, \text{request}, n_2$ and $n_2, \text{moveUp}^{\#}, n_3$ are concretizable. Hence, n_2 is selected for the split. The interpolant of $\Gamma_1(\boldsymbol{p})$ and $\Gamma_2(\boldsymbol{p})$ is the predicate $pc_1 = 1$, which is implied by $\Gamma_1(\boldsymbol{p})$ (it occurs in request(V_0, V_1)) and contradicts $\Gamma_2(\boldsymbol{p})$ ($pc_1 = 2$ occurs in moveUp$^{\#}(V_1, V_2)$).

7 Experiments

We have implemented the new model checking procedure as a small prototype tool named SLAB (for *Sl*icing *ab*stractions). SLAB is implemented in Java (JRE 1.5) and relies on Andrey Rybalchenko's *CLP-Prover* [21] for satisfiability checking and interpolant generation. In Table 2, we give running times of SLAB for a collection of standard benchmarks. Our experiments were carried out on an Intel Pentium M 1.80 GHz system with 1 GByte of RAM. For comparison, we also give the running times of the *Abstraction Refinement Model Checker* ARMC [22] and the *Berkeley Lazy Abstraction Software Verification Tool* BLAST [23] where applicable. Our benchmarks include a finite-state system (Deque), an infinite-state discrete system (Bakery), and a real-time system (Fisher).

Table 2. Experimental results for SLAB vs. ARMC and BLAST: number of iterations of the refinement loop and running times in seconds on the benchmarks Deque (with $5, \ldots, 9$ cells), Bakery (with $2, \ldots, 5$ processes), and Fisher (with $2, 3, 4$ processes). (BLAST is not applicable to the real-time system Fisher.)

	SLAB		ARMC	BLAST
specification	iterations	time (s)	time (s)	time (s)
Deque 5	6	1.34	3.80	2.23
Deque 6	6	1.92	27.65	5.64
Deque 7	8	2.70	255.63	13.64
Deque 8	8	3.15	1277.85	36.63
Deque 9	10	4.80	timeout	90.17
Bakery 2	29	6.30	2.56	9.26
Bakery 3	47	33.53	24.97	1943.17
Bakery 4	71	128.53	988.69	timeout
Bakery 5	96	376.56	timeout	timeout
Fisher 2	42	9.26	3.37	N/A
Fisher 3	335	126.05	339.21	N/A
Fisher 4	2832	2605.85	timeout	N/A

Deque. The *Deque* benchmark is an abstract version of a cyclic buffer for a double-ended queue. We model the cells of the buffer by n flags, where *true* indicates a currently allocated cell. Initially, all but the first flag are *false*. Adding or deleting an element at either end is represented by toggling a flag under the condition that the values of the two neighboring flags are different: $(\mathit{true}, \mathit{true}, \mathit{false}) \leftrightarrow (\mathit{true}, \mathit{false}, \mathit{false})$ and $(\mathit{false}, \mathit{true}, \mathit{true}) \leftrightarrow (\mathit{false}, \mathit{false}, \mathit{true})$. The error condition is satisfied if there are no unallocated cells left in the buffer.

Bakery. The *Bakery* protocol [24] is a mutual exclusion algorithm that uses *tickets* to prevent simultaneous access to a critical resource. Whenever a process wants to access the shared resource, it acquires a new ticket with a value v that is higher than that of all existing tickets. Before the process accesses the critical resource, it waits until every process that is currently requesting a ticket has obtained one, and every process that currently holds a ticket with a lower value than v has finished using the resource. An error occurs if two processes access the critical resource at the same time.

Fisher. Fisher's algorithm, as described in [25], is a real-time mutual exclusion protocol. Access to a resource shared between n processes is controlled through a single integer variable *lock* and real-time constraints involving two fixed bounds $C1 < C2$. Each process uses an individual (resettable) clock c to keep track of the passing of time between transitions. Each process first checks if the lock is free, then, after waiting for no longer than bound C1, sets *lock* to its (unique) id. It then waits for at least C2, and if the value of the lock is unchanged, accesses the critical resource. When leaving, it frees up the lock. As in the previous

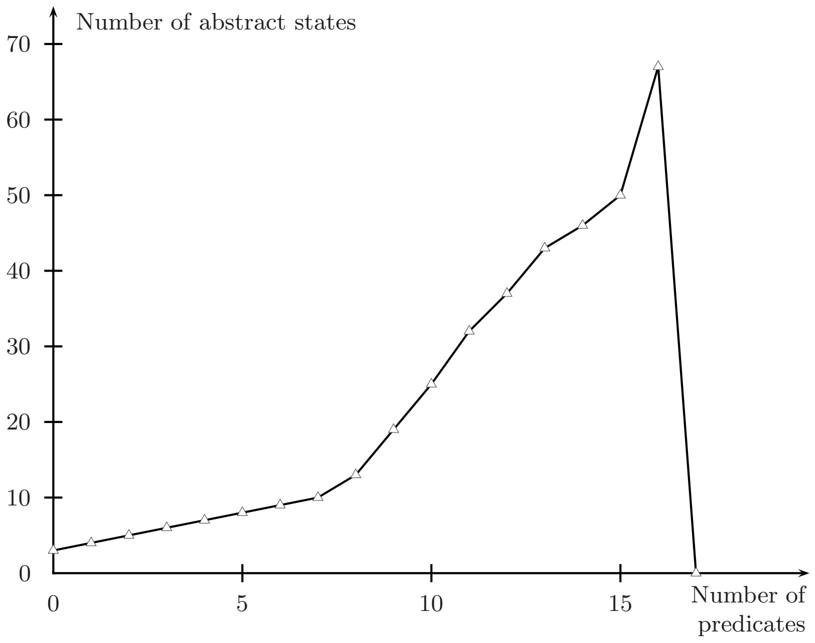


Fig. 4. Relation of the number of abstract states and the number of predicates in intermediate abstractions during the verification of the Bakery protocol with three processes

benchmark, an error occurs if two processes access the critical resource at the same time.

On our benchmarks, SLAB outperforms both ARMC and BLAST, and scales much better to larger systems. It appears that the abstract state space constructed by SLAB grows much more slowly in the number of predicates than the (fully exponential) state space considered by standard predicate abstraction: Figure 4 depicts the relation between the number of predicates and the number of abstract states in intermediate abstractions from the verification of the Bakery protocol with three processes.

8 Conclusions

We have presented a new model checking procedure for infinite-state concurrent systems that combines automatic abstraction refinement with slicing. Our experiments show that the two methods indeed complement each other well: As the refinement progresses, the increasing accuracy of the abstract model allows for a more precise slice; because the size of the resulting abstraction grows more slowly (in the number of predicates) than in standard predicate abstraction, our approach scales to larger systems.

Similar to lazy abstraction [6,7], the new approach exploits the inherent *locality* of the system. While lazy abstraction incrementally adds new predicates during a traversal of the control flow graph, ensuring that the additional predicates affect only the currently traversed sub-branch of the control flow graph, our approach refines individual nodes of the abstraction.

The state set that is partitioned according to a new predicate is thus not identified by a particular control flow location, as in lazy abstraction, but changes *dynamically* as the abstraction refinement progresses: with increasing precision of the existing abstraction, the state sets partitioned by new predicates become smaller and smaller. As a result, the abstraction process is independent of a particular control structure and can be applied to any transition system, including those with concurrent or infinite control.

References

1. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
2. Colón, M.A., Uribe, T.E.: Generating finite-state abstractions of reactive systems using decision procedures. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 293–304. Springer, Heidelberg (1998)
3. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Proc. SPIN 2001, pp. 103–122. Springer, New York (2001)
4. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
5. Das, S., Dill, D.L.: Counter-example based predicate discovery in predicate abstraction. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, Springer, Heidelberg (2002)
6. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL 2002, pp. 58–70. ACM Press, New York (2002)
7. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
8. Sipma, H.B., Uribe, T.E., Manna, Z.: Deductive model checking. Formal Methods in System Design 15(1), 49–74 (1999) (Preliminary version appeared). In: Proc. 8$^{\text{th}}$ Intl. Conference on Computer Aided Verification. LNCS, vol. 1102, pp. 208–219. Springer-Verlag, Heidelberg (1996)
9. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press, Los Alamitos (1981)
10. Ganesh, V., Saidi, N.S.H.: Slicing SAL. Technical report, SRI International (1999), `http://theory.stanford.edu/`
11. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V., Robby, W.T.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 73–89. Springer, Heidelberg (2006)
12. Millett, L., Teitelbaum, T.: Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. Software Tools for Technology Transfer 2(4), 343–349 (2000)

13. Bozga, M., Fernandez, J.C., Ghirvu, L., Graf, S., Krimm, J.P., Mounier, L.: IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In: Wing, J.M., Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 307–327. Springer, Heidelberg (1999)
14. Jhala, R., Majumdar, R.: Path slicing. In: Proc. PLDI 2005, pp. 38–47. ACM Press, New York (2005)
15. Canfora, G., Cimitile, A., Lucia, A.D.: Conditioned program slicing. Information and Software Technology Special Issue on Program Slicing 40, 595–607 (1998)
16. Fox, C., Danicic, S., Harman, M., Hierons, R.M.: Backward Conditioning: A New Program Specialisation Technique and Its Application to Program Comprehension. In: IWPC, pp. 89–97. IEEE Computer Society, Los Alamitos (2001)
17. Vasudevan, S., Emerson, E.A., Abraham, J.A.: Efficient Model Checking of Hardware Using Conditioned Slicing. ENTCS 128(6), 279–294 (2005)
18. Hong, H., Lee, I., Sokolsky, O.: Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In: SCAM, pp. 25–34. IEEE Computer Society, Los Alamitos (2005)
19. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)
20. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL 2004, pp. 232–244. ACM Press, New York (2004)
21. Rybalchenko, A.: CLP-prover (2006), `http://mtc.epfl.ch/~rybalche/clp-prover/`
22. Podelski, A., Rybalchenko, A.: ARMC: the logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, Springer, Heidelberg (2006)
23. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
24. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM 17(8), 435–455 (1974)
25. Manna, Z., Pnueli, A.: Clocked transition systems. Technical Report STAN-CS-TR-96-1566, Computer Science Department, Stanford University (1996)