

Filter-based Model Checking of Partial Systems*

Matthew B. Dwyer
Kansas State University
Department of Computing
and Information Sciences
234 Nichols Hall
Manhattan, KS 66506-2302
dwyer@cis.ksu.edu

Corina S. Pasareanu
Kansas State University
Department of Computing
and Information Sciences
234 Nichols Hall
Manhattan, KS 66506-2302
pcorina@cis.ksu.edu

1 ABSTRACT

Recent years have seen dramatic growth in the application of model checking techniques to the validation and verification of correctness properties of hardware, and more recently software, systems. Most of this work has been aimed at reasoning about properties of complete systems. This paper describes an automatable approach for building finite-state models of partially defined software systems that are amenable to model checking using existing tools. It enables the application of existing model checking tools to system components taking into account assumptions about the behavior of the environment in which the components will execute. We illustrate the application of the approach by validating and verifying properties of a reusable parameterized programming framework.

1.1 Keywords

Software verification and validation, model checking, assume-guarantee reasoning, filter-based analysis

2 INTRODUCTION

Modern software is, increasingly, built as a collection of independently produced components which are assembled to achieve a system's requirements. A typical software system consists of instantiations of generic, reusable components and components built specifically

for that system. This software development approach offers many potential advantages, but it also significantly complicates the process of verifying and validating the correctness of the resulting software systems.

Developers who wish to validate or verify correctness properties of software components face a number of challenges. By definition, reusable components are built before the systems that incorporate them, thus detailed knowledge of the context in which a component will be used is unavailable. Components are often designed to be very general, for breadth of applicability, yet configurable to the needs of specific systems; this generality may impede component verification. Typically, components are subjected to unit-level testing and are delivered with informal documentation of the intended component interface behavior and required component parameter behavior. For high-assurance systems this is lacking in a number of respects: *(i)* unit-level reasoning focuses solely on local properties of the component under consideration, *(ii)* informal documentation cannot be directly incorporated into rigorous reasoning processes, and *(iii)* system developers may have some knowledge about the context of component use, but no means of exploiting this information. In this paper, we describe an automatable approach to applying existing model checking tools to the verification of partial software systems (i.e., systems with some missing components) that addresses these concerns.

Model checking is performed on a finite-state model of system behavior not on the actual system artifacts (e.g., design, code), thus any application of model checking to software must describe model construction. In our approach, models for partial systems are constructed in two independent steps. First, a partial system is *completed* with a source code representation of the behavior of missing system components. This converts the open partial-system to a closed system to which model checking tools can be applied. Second, techniques from partial evaluation and abstract interpretation [19, 18] are

*This work was supported in part by NSF and DARPA under grants CCR-9633388, CCR-9703094, and CCR-9708184 and by NASA under grant NAG-02-1209.

applied to transform the completed source code into the input format of existing finite-state system generation tools. Finite-state models built in this way are safe, thereby insuring the correctness of verification results, but may be overly pessimistic with respect to the missing components' behavior. To enhance the precision of reasoning, we filter [16] missing components' behavior based on assumptions about allowable behavior of those components. Our approach supports model checking of systems with different kinds of missing components, including components that call, are called from, and execute in parallel with components of the partial system. This flexibility enables verification of properties of individual components, collections of components, and entire systems.

The work described in this paper extends the applicability of existing model checking techniques and tools to partial software systems and illustrates the practical benefits of the filter-based approach to automated analysis. We illustrate our approach and its benefits by verifying correctness properties, expressed in linear temporal logic (LTL) [24], of realistic generic, reusable components, written in Ada, using the SPIN model checker [20]. In principle, the approach described in this paper can be used in any setting that supports filter-based analysis [16].

In the following section we discuss relevant background material. Section 4 discusses abstractions used in constructing finite-state software models and Section 5 describes our approach to completing partial software systems. We then present our experiences and results from applying the analysis approach to a generic, reusable software component in Section 6. Section 7 describes related work and we conclude, in Section 8, with a summary and plans for future work.

3 BACKGROUND

In this section we overview LTL model checking, a variant of model checking for open systems called module checking, and a technique for refining model checking results using filter formulae. These ideas form the basis for our approach to constructing and checking finite-state models of partial software systems.

3.1 Model Checking

Model checking techniques [7, 20] have found success in automating the validation and verification of properties of finite-state systems. They have been particularly effective in the analysis of hardware systems [26] and communication protocols [20, 30]. Recent work has seen model checking applied to more general kinds of software artifacts including requirements specifications [2, 3], architectures [28], and implementations [13, 9]. In model checking software, one describes the software as

a finite-state transition system, specifies system properties with a temporal logic formula, and checks, exhaustively, that the sequences of transition system states satisfy the formula.

There are a variety of temporal logics that might be used for coding specifications. We use linear temporal logic in our work because it supports filter-based analysis and it is supported by robust, efficient model checking tools such as SPIN [20]. In LTL a pattern of states is defined that characterizes all possible behaviors of the finite-state system. We describe LTL operators using SPIN's ASCII notation. LTL is a propositional logic with the standard connectives $\&\&$, $\|\|$, \rightarrow , and $!$. It includes three temporal operators: $\langle \rangle p$ says p holds at some point in the future, $\square p$ says p holds at all points in the future, and the binary pUq operator says that p holds at all points up to the first point where q holds. An example LTL specification for the response property "all requests for a resource are followed by granting of the resource" is $\square(\text{request} \rightarrow \langle \rangle \text{granted})$.

SPIN accepts design specifications written in the Promela language and it accepts correctness properties written in LTL. User's specify a collection of interacting processes whose product defines the finite-state model of system behavior. SPIN performs an efficient non-empty language intersection test to determine if any state sequences in the model conform to the negation of the property specification. If there are no such sequences then the property holds, otherwise the sequences are presented to the user as exhibits of erroneous system behavior.

3.2 Module Checking

In computer system design, a closed system is a system whose behavior is completely determined by the state of the system. An open system (or module [22, 23]) is a system that interacts with its environment and whose behavior depends on this interaction. Given an open system and temporal logic formula, the *module checking* problem asks whether for all possible environments, the composition of the model with the environment satisfies the formula. Fortunately, for LTL, the module checking problem coincides with the basic model checking problem [22]. Often, we don't want to check a formula with respect to all environments, but only with respect to those that satisfy some assumptions. In the assume-guarantee paradigm [29], the specification of a module consists of two parts. One part describes the guaranteed behavior of the module; which we encode in the finite state system to be analyzed. The other part specifies the assumed behavior of the environment with which the module is interacting and is combined with the property formula to be analyzed.

3.3 Filter-based Analysis

Filters [16] are constraints used to incrementally refine a naively generated state space and help validate properties of the space via model checking. Filters can be represented in a variety of forms (e.g., as automata or temporal logic formulae) and are used in the FLAVERS static analysis system [14]. Filters in FLAVERS were originally developed to sharpen the precision of analysis relative to internal components of a complete software system that were purposefully modeled in a safe, but abstract manner. Filters serve equally well in refining analysis results with respect to external component behavior in the analysis of partial software systems [11].

In this paper, we encode filters in LTL formulae to perform assume-guarantee model checking. Given a property P and filters F_1, F_2, \dots, F_n that encode assumptions about the environment, we model check the combined formula $(F_1 \wedge F_2 \wedge \dots \wedge F_n) \rightarrow P$. We refer to the individual F_i as *filters* and to the combined formula as a *filter formula*.

We note that many specification formalisms support filter-based analysis, but some popular formalisms, such as CTL, do not. It may be possible to encode a filter into a CTL formula. In general, however, because multiple temporal operators cannot lie directly in the scope of a single path quantifier, there is no simple method for constructing a filter formula in CTL.

4 FINITE-STATE MODELS

In principle, model checking can be applied to any finite-state system. For non-trivial software systems we cannot render a finite-state system that precisely models the system’s behavior, since, in general, the system will not be finite-state. Even for finite-state software, the size a precise finite-state model will, in general, be exponential in the number of independent components, i.e., variables and threads of control. For these reasons, we would like to use finite-state system models that reflect the execution behavior of the software as precisely as possible while enabling tractable analysis. We use techniques from abstract interpretation to construct such models. In the remainder of this section, we introduce the notion of abstract interpretation, we then describe a collection of abstract interpretations that we use in model construction, and finally, we describe our approach to the selection of abstractions to be used for a given software system.

4.1 Safe Models for Verification

We say that a finite-state model of a software system is *safe* with respect to model checking of a property specification if model checking succeeds only when the property holds on the real system. For LTL model-

checking, which is fundamentally an all-paths analysis, any abstraction of behavior must preserve information about all possible system executions. This class of abstractions can be described as abstract interpretations (AI) [10] over the system’s execution semantics. These abstractions are similar to the kinds of approximations that are introduced into program representations (e.g., control flow graphs) used in compiler analyses [27]. When behaviors are abstracted in this way and then exhaustively compared to an LTL specification and found to be in conformance, one can be sure that the true executable system behaviors conform to the specification.

One of the strengths of AIs is that they guarantee the safety of information gathered by analyses that incorporate them. To achieve this we need to precisely define each AI. We formalize an AI as a Σ -algebra [18] which defines, for a *concrete* type signature in the source program, a type for the abstract domain of the AI and abstract definitions for the operations in the signature. Operationally we view a Σ -algebra as a data abstraction with a defined domain of values and implementations of operations over that domain. This operational view allows systematic abstraction by substituting abstract definitions for concrete definitions for abstracted program variables. Computation with abstract values and operations can then proceed in the same way as it would have for concrete values and operation.

Unlike the concrete operations in a program, we can define abstract operations to produce sets of values of the operation’s return type. This is a mechanism for encoding lack of precise information about variable values. This mechanism can be exploited by the partial evaluation capabilities discussed in Section 5. The partial evaluator will treat a set of values returned by an operation as equally likely possibilities and create a variant program fragment for each value (i.e., simulating non-deterministic choice). This allows subsequent analyses (e.g., model checking) to detect the presence of specific values that may be returned by the operation. The technical details of how this is performed is given in [18] for a simple imperative language; we have scaled those techniques up and applied them to a real Ada program in Section 6. Conceptually, one can think of partial evaluation as the engine that drives the systematic application of selected abstract interpretations to a given source program.

4.2 Sample Abstract Interpretations

Space constraints make it impossible to give the complete formalization of the AIs used in the example in Section 6. In this section, we describe the main idea of each AI and illustrate selected abstract operations.

The *point* AI is the most extreme form of abstraction.

Under this AI, a variable’s abstract domain has a single value, representing the lack of any knowledge about possible variable values. Abstract operations for the variable are defined as the constant function producing the domain value. Abstract relational operations that test the value of a variable are defined as the constant function returning the set $\{true, false\}$. While the point AI is extreme in its abstraction, it is nevertheless, not uncommon in existing FSV approaches—many state reachability analyses (e.g., CATS [32]) use this abstraction for all program variables.

Closely related to the point AI is the *choice* AI. This AI also encodes a complete lack of knowledge about possible program variables, but it does so in a different way. Abstract operations are defined to produce the set of all possible domain values. Abstract relational operations that test the value of a variable retain their concrete semantics. In most cases, the possible values reaching such a test will consist of the set of all domain values and the result of the test will be the set $\{true, false\}$. Exposing the distinct domain values to a partial evaluator gives it the opportunity to specialize program fragments with respect to possible variable values. For example, a control flow branch for each variable value can be introduced into the program, subsequent program fragments can be hoisted into each branch, and each fragment can be specialized to the branch variable value. Any tests of an abstracted variable within such a branch will have a single domain value flowing into it, thus, a more refined test result (e.g., *true* or *false*, but not both) may be computed. The resulting program model can be more precise than using the point AI, but care should be taken in applying the choice AI since it will result in a larger program model.

The *k-ordered data* AI provides the ability to distinguish the identity of k data elements, but completely abstracts the values of those elements. For $k = 2$, a 2-ordered data AI, any pair of concrete data values are mapped to abstract values d_1 and d_2 ; all other values are mapped to the *ot* value. Like in the choice AI, abstract operations are defined to return sets of values to model lack of knowledge about specific abstract values. For all operations, except assignment, the constant function returning $\{d_1, d_2, ot\}$ is used. For assignment the identity function is used. Relational test operations are slightly more subtle. Relational operations other than equality, and inequality, return $\{true, false\}$. Equality is defined as:

$$\text{Equal}(x, y) = \begin{cases} \{true\} & \text{if } x = y = d_1 \text{ or } d_2 \\ \{false\} & \text{if } x \neq y \\ \{true, false\} & \text{if } x = y = ot \end{cases}$$

Inequality is defined analogously.

A special case of the classic signs AI [1] is the

zero-pos AI which is capable of differentiating between valuations of a variable that are positive and zero. The abstract domain ranges over three values: *unknown*, *zero*, *positive*. For this AI we find it convenient to introduce the *unknown* value, which represents the fact that the variable can have either *zero* or *positive* value. Abstract operations for assignment of constant zero, increment and decrement by a positive value, and a greater-than test with zero are defined as:

$$\begin{aligned} \text{AssignZero}(x) &= \{zero\} \\ \text{IncByPositive}(x) &= \{positive\} \\ \text{DecByPositive}(x) &= \{unknown\} \\ \text{GTZero}(x) &= \begin{cases} \{false\} & \text{if } x = zero \\ \{true\} & \text{if } x = positive \\ \{true, false\} & \text{otherwise} \end{cases} \end{aligned}$$

Other operations are defined analogously.

It is also possible to define an AI that is safe with respect to a restricted class of LTL properties. One such abstraction is used commonly in verifying message ordering requirements in communication protocols. Wolper [31] has shown that reasoning about pairwise ordering questions over a communication channel that accepts large domains of values can be achieved using a domain of size three¹. This can be achieved when the data in the channel are not modified or tested by the program². We support reasoning for this class of questions through the use of a *2-ordered list* AI. This AI represents the behavior of a “list” of data items which are themselves abstracted by the 2-ordered data AI. Conceptually, the values of the abstracted list record whether a specific d_i has been inserted into the list and not removed yet. If both of the non-*ot* 2-ordered data are in the list their ordering is also recorded. No attempt is made to represent the number of *ot* elements in the list or their relative position with respect to the d_i values. The abstract list values are:

- some* : zero or more *ot* values
- d_1 : d_1 mixed with zero or more *ot* values
- d_2 : d_2 mixed with zero or more *ot* values
- $d_1 + d_2$: d_1 and d_2 mixed with zero or more *ot* values, with d_1 in front of d_2
- $d_2 + d_1$: d_2 and d_1 mixed with zero or more *ot* values, with d_2 in front of d_1

Technically, this AI is not safe for LTL (e.g., it does not allow for lists with multiple instances of d_1 values), however, the abstraction is safe for all system executions for which the d_i are inserted at most once into the list. Thus, the AI is safe for LTL formulae of the form:

¹Requirements involving more than a pair of data items can be handled by a simple scaling of the approach described here.

²This condition can be enforced using an approach that is similar to the restriction of errors discussed in Section 5.

```

 $\square(\text{return\_Insert}(d_1) \rightarrow \square(\text{!call\_Insert}(d_1))) \ \&\&$ 
 $\square(\text{return\_Insert}(d_2) \rightarrow \square(\text{!call\_Insert}(d_2))) \rightarrow P$ 

```

where P is an arbitrary LTL formula and the `call` and `return` prefixes indicate the program actions of invoking and returning from an operation. This is a filter-formula, as described in Section 3, that restricts checking of P to paths that are consistent with the information preserved by the the 2-ordered list AI.

We illustrate abstract list operations for **Inserting** elements at the tail and **Removing** elements at the head; other operations are defined analogously.

$$\text{Insert}(L, x) = \begin{cases} \{-, d_i\} & \text{if } x = d_i \wedge L = \text{some} \\ \{-, d_i + d_j\} & \text{if } x = d_j \wedge L = d_i \\ \{-, L\} & \text{otherwise} \end{cases}$$

$$\text{Remove}(L) = \begin{cases} \{(d_i, \text{some}), (ot, L)\} & \text{if } L = d_i \\ \{(d_i, d_j), (ot, L)\} & \text{if } L = d_i + d_j \\ \{(ot, L)\} & \text{otherwise} \end{cases}$$

Since list operations may both produce values and update the list contents we define abstract operations over tuples. The first component is the return value of the operation; $-$ indicates that no value is returned. The rest of the components define how the components of the AI should be updated based on the operation. In the next section we discuss the use of a 2-ordered abstraction in completing a partial system that enables verification of ordering properties of data-independent systems.

4.3 Abstraction Selection

Given a collection of program variables and a collection of AIs we must select, for each variable, the AI which will define its semantics in the finite-state model. We do not believe that this process can be completely automated in all cases. Our experience applying AIs in model construction, however, has left us with a methodology and a set of heuristics for selecting abstractions. In this methodology, we bind variables to AIs, then use additional program information to refine the modeling of a variable by binding it to a more precise AI.

Start with the point AI Initially all variables are modeled with the point AI.

Use choice for variables with very small domains

Variables with domains of size less than 10 that are used in conditional expressions are modeled with the choice AI.

Identify semantic features in the specification

The property to be checked includes, in the form of propositions, different semantic features of the program (e.g., valuations of specific program variables). These features must be modeled precisely by an AI to have any hope of checking the property.

Select controlling variables In addition to variables mentioned explicitly in the specification, we

```

d : Boolean;
x, y : Natural;
begin
...
if not d then
  x := 0;
end if;
...
if y > 0 and not d then
  P();
end if;
...

```

Figure 1: Example for AI Selection

consider variables on which they are control dependent. The conditional expressions for these controlling variables suggest semantic features that should be modeled by an AI.

Select variables with broadest impact When confronted with multiple controlling variables to model, select the one that appears most often in a conditional.

After the selection process is complete, we generate a finite-state model using the variable-AI bindings and check the property. Model checker output either proves the property or presents a counter-example whose analysis may lead to further refinement of the AIs used to model program variables.

To illustrate this methodology, consider the program fragment in Figure 1 which has variables d , x and y . Assume we are interested in reasoning about the response property $\square(x \text{ ISzero} \rightarrow \langle \rangle \text{call_P})$. The key features that are mentioned explicitly in this specification are values of variable x and calls to procedure P . We must model x with more precision than the point AI provides in order to determine the states in which it has value zero. An effective AI for x must be able to distinguish zero values from non-zero values; we choose the zero-pos AI. At this point we could generate an abstracted model and check the property or consider additional refinements of the model; we choose the latter for illustrating our example. Using control dependence information we can determine the variables that appear in conditionals that determine whether statements related to x and P execute. In our example, there are two such variables d and y . We could refine the modeling of both of these variables, but, we prefer an incremental refinement to avoid unnecessary expansion of the model. In choosing between these variables, we see that d appears in both conditionals and we choose to model it since it may have a broader impact than modeling y . Since, d is a boolean variable and the conditional tests for falsity, we choose for it to retain its concrete semantics. At this point, we would generate an abstracted model and check the property. If a true result is obtained then we are sure that the property holds on the program, even though the finite-state system only models two vari-

ables with any precision. If a false result is obtained then we must examine the counter-example produced by the model checker. It may reveal a true defect in the program or it may reveal an infeasible path through the model. In the latter case, we identify the variables in the conditionals along the counter-example's path as candidates for more binding to more precise AIs.

This methodology is not foolproof. It is based on a fixed collection of AIs and a given program variable may require an AI that is not in that collection. Our heuristics for choosing variables to refine may cause the generation of finite-state models that are overly precise, and whose analysis is more costly than is necessary. Nevertheless, this approach has worked well on a variety of examples and we will continue to improve it by incorporating additional AIs and mechanisms for identifying candidates for refinement. While not fully-automatable, this methodology could benefit from automated support in computing controlling variables and in the analysis of counter-examples. We are currently investigating how best to provide this kind of support.

5 COMPLETING SYSTEMS

A partial system is a collection of procedures and tasks³. We complete a partial system by generating source-code that implements missing components, which we call *contexts*. These context components are combined with the given partial system.

Stubs and drivers are defined to represent three different kinds of missing components; calling, called and parallel contexts. Calling contexts represent the possible behavior of the portions of an application that invoke the procedures of the partial system. Called contexts represent the possible behavior of application procedures that are invoked by the procedures and tasks of the partial system. Parallel contexts represent those portions of an application that execute in parallel with and engage in inter-task communication with the procedures and tasks of the partial system.

For simplicity, in our discussion and the examples in this section, we phrase our system models and properties in terms of events (i.e., actions performed by the software). It is often convenient to use a mixture of event and state-based descriptions in models and properties and we do so in the example in Section 6.

To begin construction of any system model one must have a definition of the events that are possible. For Ada programs, these events include: entry calls or accepts, calls or returns from procedures, designated statements being executed or variables achieving a specified value. We partition the events into those that are

³The approach in this section can easily be extended to support packages and other program structuring mechanisms.

```

procedure P() is
begin
  T.E();
  C;
end P;

task body T is
begin
  accept E;
  missing();
  accept E;
end T;

procedure stub() is
  choice : Integer;
begin
  loop
    case choice is
      when 1 => A;
      when 2 => B;
      when 3 => P();
      when 4 => null;
      otherwise => exit;
    end case;
  end loop;
end stub;

task body driver is
begin
  stub();
end driver;

task body T is
  choice : Integer;
begin
  accept E;
  -- call_missing;
  loop
    case choice is
      when 1 => A;
      when 2 => B;
      when 3 => null; ★
      when 4 => null;
      otherwise => exit;
    end case;
  end loop;
  -- return_missing;
  accept E;
end T;

task body driver is
  choice : Integer;
begin
  loop
    case choice is
      when 1 => A;
      when 2 => B;
      when 3 => T.E; C;
      when 4 => null;
      otherwise => exit;
    end case;
  end loop;
end stub;

```

Figure 2: Partial Ada System, Stubs and Drivers

internal to the components being analyzed and *external* events, which may be executed by missing components. Based on this partitioning we construct a stub procedure that represents all possible sequences of external events and calls on public routines and entries in the partial system.

Figure 2 illustrates, on the left side, a partial system consisting of procedure P and task T. The internal events are calls on the E entry of T and execution of C. Two external events are defined as A and B. The stub procedure and driver task are also given in the figure. Because there are no external entries there is no parallel context defined.

Existing model checking tools require a single finite-state transition system as input. To generate such a system from a source program with procedures requires inlining, or some other form of procedure integration. We describe the construction of a source-level model for a completed partial system as a series of inlining operations. We assume that there are no recursive calls in the system's procedures, stubs and drivers. Given this assumption, Figure 3 gives the steps to assemble a completed system. Applying these steps to the example gives the code on the right side of Figure 2.

The same stub procedure is used to model the behavior of all missing called components. To enable model checking based on assumptions about the behavior of

Input: collection of procedures, tasks and a description of the external alphabet

Output: a source level system without external references

Steps:

1. Generate stub procedure that non-deterministically chooses between the actions in the external alphabet and calls to the procedures and entries of the program components. It must also be capable of choosing to do nothing or to return.
2. Inline calls to procedures made by stubs.
3. Stubs may now contain (inlined) calls to task entries. For each task that calls a missing component specialize the stub so that any calls to that tasks entries are replaced by an error indication.
4. Calls to missing components from tasks are replaced by the stub routine. Indications of the call to and return from the missing component are inserted before and after the stub body.
5. The driver and parallel contexts, if any, are formed by inlining the stub body.

Figure 3: System Completion Algorithm

specific missing components we bracket the inlined stub with indicators of call and return events for the missing component (e.g., `call_missing` and `return_missing`).

The goal of the system completion process is to produce legal Ada source code so that subsequent tools can process the system. This is somewhat at odds with the fundamental lack of knowledge about event ordering in missing components. We model this lack of knowledge with non-determinism by introducing a new variable, `choice`, that is tested in the stub conditionals. This variable will be abstracted with a point AI and subsequent model construction tools will represent such conditionals with non-deterministic choice.

We must take care to insure that potential run-time errors are preserved in the completed system, since they contribute to the actual behavior of the software. For example, it is possible for system tasks to call stubs which in turn call system procedures containing entry calls on that task; this is a run-time error in Ada. To preserve this possible behavior we introduce an `error` event. In the example in Figure 2 the point at which such an event is introduced is marked by a `★`. This allows a user to test for the possibility of the run-time error or to filter the allowable behavior of missing components to eliminate the error (i.e., using `[]!error` as the property to be checked or as a filter). This conversion is done separately for each task and amounts to specialization of the stub body for the task by interpreting self-entry calls as the `error` event.

Completing a partial program does not yield a finite-state system. The next step is to selectively abstract program variables and transform dynamic program behavior to a static form.

5.1 Automating Model Construction

Our approach to automating the construction of safe finite-state models of software systems builds on recent work in abstraction-based program specialization [18]. Figure 4 illustrates the steps in converting Ada source to Promela which can be submitted to the SPIN model checker. First, the partial system is completed with a source-level model of its execution environment. We apply a source-to-source partial evaluation tool that transforms the program to a form that is more readily modeled as a finite-state system. Partial evaluation is a program transformation and specialization approach which exploits partial information about program data. Essentially it performs parts of a program’s computation statically; the result is a simplified program that is specialized to statically available data values. A wide-variety of source transformations can be applied to aid in finite-state model construction including; procedure integration, bounded static variation, and migration of dynamically allocated data and tasks to compile-time [19]. A novel feature of the approach we use is its ability to incorporate AIs for selected variable [18]; we use the variable-AI bindings discussed in Section 4.

After partial evaluation, a tool is applied to convert the resulting Ada to SEDL, an internal form used by the Inca [4] toolset⁴, which is then converted into Promela. The Promela is then submitted, along with an LTL specification, to SPIN which produces either an indication of a successful model check or a counter-example.

Aside from selection of AIs, this approach is completely automatable. At present, the system completion and the partial evaluation tools are not fully-implemented and were not used in the experiments described in Section 6; all other tools depicted in Figure 4 were run without user intervention. Stubs, drivers and the abstracted, specialized Ada for those experiments were constructed by-hand using the algorithms that are being implemented in our partial evaluation tool. We are implementing an approach to stub and driver generation using ideas from work on synthesis of program skeletons from temporal logic specification [25]. With this approach we will be able to encode filters on environment behavior directly into stubs and drivers, thereby eliminating the need for including those filters in the formula to be checked. It remains to be seen whether encoding filters in the transition system or in the formula to be checked results in better performance; we plan to explore this question in future work.

⁴Inca was previously referred to as the constrained-expressions toolset.

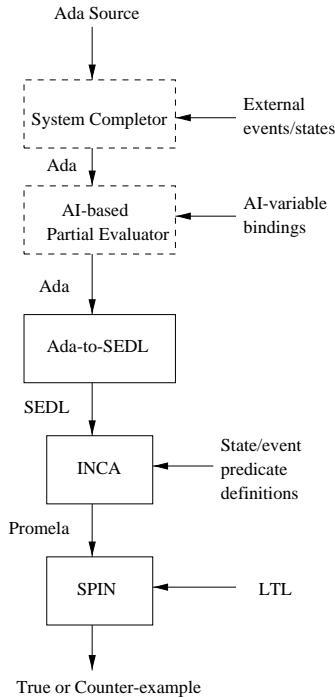


Figure 4: Model Construction Process

6 EXPERIENCES

In this section, we describe our experiences with applying the techniques described in this paper to model checking of a real partial software system. We begin with a description of this partial system.

6.1 Replicated Workers Computations

The replicated workers framework (RWF) is a parameterizable parallel job scheduler, where the user configures the computation to be performed in each job, the degree of parallelism and several pre-defined variations of scheduler behavior. An instance of this framework is a collection of similar computational elements, called workers. Each worker repeatedly accesses data from a shared work pool, processes the data, and produces new data elements which are returned to the pool. User's define the number of workers, the type of work data, and computations to be performed by a worker on a data item. A version of this framework, written in Ada [15], implements workers, the pool and a lock as dynamically allocated instances of task types.

Figure 5 illustrates the structure of the replicated workers framework and a sample of its interaction with a user application; procedure and entry calls are depicted with dashed and solid arrows, respectively, in the figure. Applications **Create** a collection of workers and a work pool and configure certain details of framework operation (e.g., whether the **Execute** routine operates as a **Synchronous** or **Asynchronous** invocation). A compu-

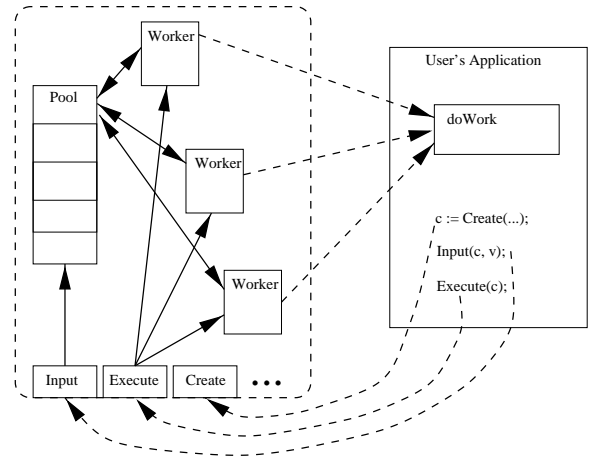


Figure 5: The Replicated Workers Framework

tation is initialized through calls to the **Input** routine and started by calling **Execute**. Communicating only by way of the workpool, the collection of workers cooperate to perform the desired computation and terminate their execution when complete. Detailed description of the behavior of the RWF is provided in [15].

The execution state of the RWF consists of the local control flow states of a single pool, a single lock, and each of the workers. In addition, each of these tasks maintains local data. The original Ada code for the pool task, on the left of Figure 6, has a boolean variable, **executeDone**, three natural variables, **numWait**, **numIdle**, and **workCount**, two linked lists, **WorkPool.List** and **newWork**, two variables of the work type, and an array of task accesses **workers**, which is accessed through the discriminant value **C**. The lock task has a single boolean variable. Each worker task has a boolean variable, **done**, three linked lists, a task access variable, and an integer variable. In addition to the internal state of the RWF, we need to consider the state of the context, represented by stub and driver code. The only data component of that state is the work value passed to **Input**, which we refer to as **driverInput**. We will see, below, that these variables are abstracted in a variety of different ways in the finite-state models used for system validation.

6.2 Building RWF Models

We use the approach described in Section 5 to produce finite-state systems that represent the behavior of the replicated workers framework. The framework is built of three active components: a task type (**ActivePool**) for the pool, a task type (**ActiveWorker**) for the worker, and a task which mediates access to a shared resource (**ResultLock**). The user is provided access to framework functionality through a collection of public procedures: a constructor **Create**, **Input** and

```

task body ActivePool is
  -- C : Collection is a discriminant
  executeDone : Boolean := TRUE;
  workCount,numWait,numIdle : Natural;
  work : WorkPool.List;
  workItem, resultItem : WorkType;
begin
  accept StartUp;
  WorkPool.Create(work);
  workCount := 0;
  Outer: loop
    loop
      select ***
        or accept Execute;
          C.done := FALSE;
          for i in 1 .. C.max loop ★
            C.workers(i).Worker.Execute;
          end loop;
          exit;
        *** end select;
      end loop;
    ***
    loop
      select ***
        or accept Put(newWork:in out WList) do
          Remove(newWork, workItem);
          for i in 1..Size(newWork) loop ★★
            Insert(work, workItem);
            workCount := workCount + 1;
            Remove(newWork, workItem);
          end loop;
          end Put;
          numIdle := numIdle + 1;
        *** end select;
        if numIdle = C.max and workCount=0 then
          executeDone := TRUE;
        end if;
        exit when numWait = C.max;
      end loop;
      if ExecSemantics = Synchronous then
        accept Complete;
      end if;
      C.done := TRUE;
    end loop Outer;
  end ActivePool;

```

```

type ZERO_POS is (zero, positive);
GEN1CollectionInfo.done : Boolean;

task body GEN1ActivePool is
  executeDone : Boolean := TRUE;
  workCount : ZERO_POS;
  choice : Boolean;
begin
  accept StartUp;

  workCount := zero;
  Outer: loop
    loop
      select ***
        or accept Execute;
          GEN1CollectionInfo.done := FALSE;
          GEN1ActiveWorker.Execute; ★
          GEN2ActiveWorker.Execute;
          GEN3ActiveWorker.Execute;
          exit;
        *** end select;
      end loop;
    ***
    loop
      select ***
        or accept Put() do
          if choice then ★★
            workCount := positive;
          end if;
        end Put;
        *** end select;
        if numIdle=3 and workCount=zero then
          executeDone := TRUE;
        end if;
        exit when numWait=3;
      end loop;

      accept Complete;

      GEN1CollectionInfo.done := TRUE;
    end loop Outer;
  end GEN1ActivePool;

```

Figure 6: Original Ada and Abstracted, Specialized Ada for ActivePool Task

Output routines, and a routine to **Execute** the computation. **ActiveWorkers** call user provided functions (**doWork**, **doResults**) to perform subcomputations on given work and result data.

In validating the RWF implementation we assume that only one task creates and accesses each instance of the framework. This means that a single driver can be used to complete the system model; if the assumption is relaxed we would incorporate multiple drivers and a parallel component. We illustrate the analysis of a configuration of the RWF with three workers and **Synchronous** execution semantics. We will reason about local correctness properties of this system that are either internal to the RWF or related to the semantics of the RWF’s application interface. For this reason, the external alphabet is empty. The stub generated by the algorithm in Figure 3 in this case consists of choices among calls to the RWF procedures.

Defining the generic parameters and parameters to the **Create** call to be consistent with these assumptions enables program specialization to eliminate a number of program variables. In particular, the pool’s work variables and array of task accesses, and each worker’s three linked lists, task access variable, and integer variable are eliminated. Some of these variables were eliminated because their values are known to be constant, others are eliminated, by copy propagation, because they only transfer values between other variables. A significant number of variables, ranging over large domains, remain in the program so we apply the AIs, from Section 4, to the remaining variables to construct three different abstracted versions of the RWF system.

Model 1. This model is the most aggressively abstracted. The variables **numWait**, **numIdle**, **workCount**, **WorkPool.List**, **newWork** and **driverInput** are all abstracted to the point AI. Variables **executeDone**, **done**,

and the lock's boolean retain their concrete semantics. Our initial attempts to validate RWF properties did not use this model; we used model 2. We developed this model in order to see if any of the existing properties could be checked on a more compact model than 2. The results presented below confirm that this was possible.

Parameters passed to `doWork` and `doResult` routines, which are modeled with stubs, also require abstraction. In this model the input work parameter is abstracted with the point AI and the boolean output parameter uses the choice AI.

Model 2. Figure 6 gives the original Ada source and the abstracted, specialized Ada code for the `ActivePool` task of the RWF. Due to space limitations some details of the example are elided from the Figure, denoted by `***`, but the most interesting transformations remain. As with the first model, several local variables, `WorkPool.List`, `newWork` and `driverInput`, are abstracted to the point AI. Where those variables can influence branch decisions, non-determinism is used. Since there is no non-deterministic choice construct in Ada, we introduce a new variable `choice` that indicates, by convention, to the model construction tools that a non-deterministic choice of the value of the variable is desired. Of the remaining variables, `executeDone`, `numWait` and `numIdle` retain their concrete semantics and `workCount` is abstracted with a zero-pos AI. We note that that `numWait` and `numIdle` both act as bounded counters up to the number of workers, thus they have a relatively small impact on the size of the model. `ActiveWorker` tasks for this model are the same as for model 1.

Some details of the specialization process are illustrated in Figure 6. Knowledge of the number of workers is exploited to unroll the `*` loop and specialize its body. As a consequence of this, the resulting Ada contains only static task references (e.g., `GEN1ActiveWorker`). In fact, partial evaluation applied to this example converts all dynamically allocated data and tasks to a static form and all indirect data and task references to a static form. The `**` loop is not unrolled, rather, because of the zero-pos AI used in its body the specializer determines that there are only two possible values for `workCount` after the loop, unchanged and `positive`, and produces the conditional.

Model 3. Model 2 was insufficient for validation of ordering properties of work items in the RWF. We constructed a third RWF model that incorporated the 2-ordered data AI for `WorkType` data and the 2-ordered list abstraction for the `WorkPool.List` data. Even though this model uses a non-trivial domain for variables of `WorkType`, the resulting model did not explic-

itly require the modeling of the pool's local variables, since they only serve to transfer values between lists.

It is not possible to generate a compact, finite-state stub and driver that will input any sequence of work data to the partial system under analysis. For the properties we wish to check, such generality is not required of the stub and driver. In models 2 and 3 we require no information about the input sequence and consequently the point AI suffices. In this model, we require a finer abstraction. The stub and driver for this model incorporated the 2-ordered environment abstraction binding the `driverInput` variable with the 2-ordered data AI. Thus, input sequences are modeled as sequences of values from $\{d_1, d_2, ot\}$.

6.3 The Properties

We model checked a collection of correctness requirements of the replicated workers framework. The requirements were derived from an English language description of the framework and encoded as LTL formulae using patterns [12]. We expressed all the formulae in terms of event and state predicates that are converted automatically by the Inca toolset into propositions for use in defining LTL formulae for SPIN. An event refers to the occurrence of a rendezvous, a procedure call, or some other designated program statement. An event predicate is true if any task containing the specified event is in a state immediately following a transition on that event. State predicates define the points at which selected program variables hold a given value (e.g., states in which `workCount` is zero). We note the defining boolean expressions for encoding state predicates can be quite involved in some cases. For example, the Inca predicate definition for states where `workCount` is `zero`:

```
(defpredicate "workCountISzero"
  (in-task activepool-task (= workCount "zero")))
```

causes the generation of a disjunction of 123 individual state descriptions, i.e., one for each state of the `ActivePool` task in which `workCount` has the value `zero`. Our experience suggests that automated support for such definitions is a necessary component of any finite-state software verification toolset.

A selection of the specifications we checked are given in Figure 7. All model checks were performed using SPIN, version 3.09, on a SUN ULTRA5 with a 270Mhz UltraSparc IIi and 128Meg of RAM. Figure 8 gives the data for each of the model checking runs; the transition system model used for the run is given. We report the user+system time for running SPIN to convert LTL to the SPIN input format, to compile the Promela into a model checker, and to execute that model checker. The model construction tools were run on an AlphaStation 200 4/233 with 128Meg of RAM. The longest time taken

- (1) `[]((call_doResults:i && <>return_doResults:i) -> (!call_doResults:j) U return_doResults:i))`
Mutually exclusive execution of doResults.
- (2) `(<>call_Execute) -> (!(call_doWork) U call_Execute)`
No work is scheduled before execution.
- (3) `[]((return_Execute && (<>call_Execute)) -> (!call_doWork) U call_Execute))`
No new work scheduled after termination.
- (4) `[](call_Execute -> (!(return_Execute) U (done_w1 || done_w2 || done_w3 || workCountEQZero || [](!return_Execute))))`
Computation terminates when workpool is empty or worker signals termination.
- (5) `[](workCountGTZero && accept_ActivePool.Get && !(done_w1 || done_w2 || done_w3) -> <>call_doWork)`
If a worker is ready to get work, the workpool is not empty and the computation is not done, then work is scheduled.
- (6) `[]((call_Input(d1) && <>call_Input(d2)) -> (!ActivePool.Get(d2) U (ActivePool.Get(d1) || [](!ActivePool.Get(d2)))))`
Pool schedules work in input order.
- (7) `[](call_doWork:i(d1) -> [](!call_doWork:j(d1)))`
After a work item is scheduled, it will not be scheduled again.
- (8) `[](return_doWork:i(d1) -> [](!call_doWork:i(d1)))`
After a work item is processed, it will not be scheduled.

Figure 7: LTL Specifications

to convert completed Ada to SEDL was for model 3; it took 66.3 seconds. Generating Promela from the SEDL can vary due to differences in the predicate definitions required for different properties. The longest time taken for this step was also for model 3; it took 16.4 seconds.

6.4 Discussion

All specified properties were known to hold on the RWF implementation we analyzed. For specifications 1-3 no filters were required to obtain true results. The remaining specifications required some form of filter. Properties 6-8 required the filter in Section 4 to insure the safety of the model check results under the 2-ordered AI incorporated in the transition system. We discuss the filters for properties 4-5 below. Specifications 1 and 7 are short-hand for collections of specifications for all i and j between 1 and 3, where i and j are different worker tasks ids. The model checks times were equal for the different versions of each specification; one such time is given in Figure 8.

Modeling missing components using the permissive stubs and drivers described in Section 5 has the advantage of yielding safe models for the system configuration considered. Its drawback is that it may not precisely describe the required behavior of missing components. This is the reason that model checks for specifications 4

Property	Time	Result	Model
(1)	0.1, 1:44.4, 2:14.9	true	1
(2)	0.1, 2:35.8, 0.1	true	1
(3)	0.1, 2:45.0, 3:48.0	true	2
(4)	0.5, 2:35.0, 1:58.5	false	2
(4f)	0.7, 3:44.3, 8.8	true	2
(5)	0.2, 6:55.2, 2.4	false	2
(5f)	0.3, 6:47.0, 0.1	true	2
(6)	0.9, 31:56.5, 60.1	true	3
(6f)	1.3, 36:58.7, 13:08.1	true	3
(7)	0.1, 31:42.9, 59.9	true	3
(7f)	0.1, 33:33.2, 15.45.7	true	3
(8)	0.1, 31:30.9, 1:00.9	true	3
(8f)	0.1, 33:56.3, 14:43.9	true	3

Figure 8: Performance Data

and 5 failed. To boost precision in analyzing these properties, we code assumptions about the required behavior of missing components (e.g. `doWork`) as filter-formulae that are then model checked.

Analysis of the counter example provided by SPIN for specification 4 showed that `doResults` calls made from `GEN1ActiveWorker` can call the `ActivePool.Finished` entry. This is because the stub routine allows `doResults` to perform any computation. API documentation for the RWF warns users against calling RWF operations from `doWork` and `doResults`. If we assume that users heed this warning, we can define two filters that eliminate such calls. The resulting filter formula, (4f), is:

```
[[](!call_stubRWF_w) && [](!call_stubRWF_r) ->
([](call_Execute -> (!(return_Execute) U
(done_w1 || done_w2 || done_w3 ||
workCountISZero || [](!return_Execute)))))]
```

where `call_stubRWF_w` and `call_stubRWF_r` are rather large conjunctions of the events that correspond to the calls of RWF operations from stubs inlined at `doWork` and `doResult` call-sites within workers. The generation of these propositions is relatively simple using Inca's predicate definition mechanism. The same filters were used for specification (5f).

The use of filters in properties (6-8f) was required since the AIs incorporated in the model were only guaranteed to be safe under the assumption of a single `Insert` of each work datum into the abstracted `WorkPool.List`. Even though the unfiltered versions of those properties returned a true result, those results cannot be trusted. It may be the case that the AI caused certain possible system executions to be excluded from the analysis. If such an execution violated the specified property then a true result might be returned when there is a defect in the system. To insure that this is not the case, we checked the following filter formula (6f):

```
[[](return_Insert(d1) -> [](!call_Insert(d1))) &&
[](return_Insert(d2) -> [](!call_Insert(d2)))) ->
```

```

[]((call_Input(d1) && <>call_Input(d2)) ->
  (ActivePool.Get(d2) U
  (ActivePool.Get(d1) || [](!ActivePool.Get(d2)))))

```

Properties (7-8f) only referred to `d1` so they only include the filter that restricts `Inserts` of `d1`.

6.5 Lessons Learned

Our experience using filters for model checking with this example is consistent with previous work on filter-based verification [11, 14, 28]. In many cases, no filters are necessary and when necessary relatively few filters are sufficient to achieve the level of precision necessary for property verification. Model checking of properties for our sample system was fast enough to be usable in a practical development setting. We note that the the second component of model check time in Figure 8 is the sum of the time for SPIN to compile Promela input to a C program and the time to compile that C program. The bulk of this cost in all cases was compiling the C program. The reader should not interpret these times as an inherent component of the cost of using SPIN. The Inca tools that we use to generate Promela code encode local task data into the control flow of a Promela task rather than as Promela variables. This can cause a dramatic increase in the size of the C program generated and consequently the compile times are significant. Further study is required to determine whether a more direct mapping to Promela would yield significant reductions in these times.

It is interesting to note that the addition of filters can in some cases reduce analysis cost (e.g., property (4f)) while in others it can dramatically increase analysis cost (e.g., property (6f)). Conceptually, analysis cost can be reduced because paths through the finite-state model that are inconsistent with the filters are not considered during model checking. Analysis cost can be increased, on the other hand, because the effective state space (i.e., the product of the model and the property) is significantly larger. Further study is needed to understand the situations in which reduction or increase in analysis cost can be expected when using filters.

Our methodology for incorporating AIs into finite-state models yields aggressively abstracted transition systems. Nevertheless, as one might expect, even the relatively small changes in the abstractions we incorporated into our three models dramatically change the space, and consequently the time, required for model checking. Checks for properties (1) and (2), using model 1, required on the order of 1000 states to be searched, whereas checks for properties (6-8), using model 3, required on the order of 100000 states to be searched. We believe that constructing compact transition systems, while retaining sufficient precision to enable successful model checks, requires that AIs be selected and incor-

porated independently for properties that refer to the same set of propositions. In our experiments, the cost of model construction is not the dominant factor in analysis time, so the benefits of independently abstracted models may yield an overall reduction in analysis time.

It is important to note that these observations are based on checking of local properties of a cohesive partial system with a relatively narrow and well-defined interface. In principle, it may be necessary to include a very large number of filters which may dramatically increase model check cost. We believe that application of the approach described in this paper is most sensible at points in the development process where unit and integration-level testing is currently applied. In this context, we believe that the sub-systems under analysis will be similar to the RWF system (i.e., highly cohesive and loosely coupled to the environment). Further evaluation is necessary to determine this conclusively and to study the cost of filter-based model checking for partial systems that are strongly coupled to their environment.

7 RELATED WORK

The work described in this paper touches on model checking of software systems, model checking of open or partial systems and abstractions in model checking. In Section 3, we have already discussed the bulk of the related work.

There have been some recent efforts to apply model checking techniques to abstracted software systems (e.g., [13, 30]). In that work, ad-hoc abstraction was performed by hand transforming source code into models suitable for analysis. While automating the selection of abstractions is a very difficult problem, application of abstractions is relatively well-understood. Unlike ad-hoc methods, our work builds off the solid semantic foundations and rich history of existing abstractions that have been developed in the twenty year history of abstract interpretation [10]. Furthermore, we explore the use of partial evaluation techniques (e.g., [21]) as a means of automating application of those abstractions.

Our use of filters to refine a model of the environment is similar to other work on compositional verification. These divide-and-conquer approaches, decompose a system into sub-systems, derive interfaces that summarize the behavior of each sub-system (e.g., [6]), then perform analyses using interfaces in place of the details of the sub-systems. This notion of capturing environment behavior with interfaces also appears in recent developments on theoretical issues related to model checking of partial systems (e.g., [22, 23]). There has been considerably less work on the practical issues involved with finite-state verification of partial systems. Aside from our work with FLAVERS, discussed in Section 3,

there are two other recent related practical efforts.

Avrunin, Dillon and Corbett [5] have developed a technique that allows partial systems to be described in a mixture of source code and specifications. In their work, specifications can be thought of as assumptions or filters on a naive completion of a partial system given in code. Unlike our work, their approach is targeted to automated analysis of timing properties of systems.

Colby, Godefroid and Jagadeesan [8] describe an automatable approach to completing reactive partial system. Unlike our approach, their work is aimed at producing a completed system that is executable in the context of the VeriSoft toolset [17]. Their system completion acts as a controlling environment that causes the given partial system to systematically explore its behavior and compare it to specifications of correctness properties. To produce a tractable completion, they perform a number of analyses to determine which portions of the partial system can be influenced by external behavior, for example, tests of externally defined variables are modeled with non-deterministic choice. This is equivalent to abstracting all external data with a point AI, which happens by default in our approach. Our use of filters allows restriction of external component behavior which is not possible in their approach. Both approaches are sensitive to elimination, or abstraction, of program actions that may cause run-time errors; in our case this is manifested by modeling self-entry calls as **error** actions.

8 CONCLUSIONS

We have described an automatable approach to safely completing the definition of a partial software system. We have shown how a completed system can be selectively abstracted and transformed into a finite-state system that can be input to existing model checking tools. We have illustrated that this approach strikes a balance between size and precision in a way that enables model checking of system requirements of real software components. Finally, we have shown how to refine the representation of system behavior, in cases where the precision of the base representation is insufficient, to enable proof of additional system requirements.

There are a number of questions we plan to investigate as follow on work. In the work described in this paper, we encode filter information into the model check on-the-fly, an alternate method is to encode it directly into the finite-state system. We are currently comparing these two approaches in order to characterize the circumstances in which one approach is preferable to the other. In this paper, we consider individual abstractions, encoded as AI, but we know of systems where the desired abstraction is a composition of two AIs. We are

investigating the extent to which construction of such compositions can be automated. Finally, we are continuing development of the tools that make up our approach and we plan to evaluate their utility by applying them to additional real software systems. Along with a completed toolset, we plan to produce a library of abstractions that users can selectively apply to program variables. This will allow non-expert users to begin to experiment with model checking of source code for real component-based software systems.

9 ACKNOWLEDGEMENTS

Thanks to John Hatcliff and Nanda Muhammad for helping to specialize versions of the replicated workers framework by hand. Thanks to James Corbett and George Avrunin for access to the Inca toolset. Special thanks to James for responding to our request for a predicate definition mechanism with an implementation. Thanks to the anonymous referee who gave very detailed and useful comments on the paper.

10 REFERENCES

- [1] S. Abramsky and C. Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, 1987.
- [2] R. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *Software Engineering Notes*, 21(6):156–166, Nov. 1996. Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering.
- [3] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, June 1993.
- [4] G. Avrunin, U. Buy, J. Corbett, L. Dillon, and J. Wileiden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, Nov. 1991.
- [5] G. Avrunin, J. Corbett, and L. Dillon. Analyzing partially-implemented real-time systems. In *Proceedings of the 19th International Conference on Software Engineering*, May 1997.
- [6] S. C. Cheung and J. Kramer. Checking subsystem safety properties in compositional reachability analysis. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Mar. 1996.
- [7] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, Apr. 1986.
- [8] C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing open reactive programs. In *ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. ACM Press, June 1998. to appear.

- [9] J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), Mar. 1996.
- [10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
- [11] M. Dwyer. Modular flow analysis for concurrent software. In *Proceeding of the 12th IEEE Conference on Automated Software Engineering*, Nov. 1997.
- [12] M. Dwyer, G. Avrunin, and J. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, Mar. 1998.
- [13] M. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In *LNCS 1301*, pages 244–261. Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sept. 1997.
- [14] M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. *Software Engineering Notes*, 19(5):62–75, Dec. 1994. Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering.
- [15] M. Dwyer, M. Craig, and E. Runquist. An application-independent concurrency skeleton in Ada-95. In *Proceedings of the TRI-Ada'96 Conference*, Dec. 1996.
- [16] M. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the 1st International Workshop on Verification, Abstract Interpretation and Model Checking*, Oct. 1997.
- [17] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Jan. 1997.
- [18] J. Hatcliff, M. B. Dwyer, and S. Laubach. Staging static analysis using abstraction-based program specialization. In *Proceedings of the Joint International Symposium of the 10th Programming Languages, Implementations, Logics and Programs and the 7th Algebraic and Logic Programming Conferences (PLILP/ALP)*, Sept. 1998.
- [19] J. Hatcliff, M. B. Dwyer, S. Laubach, J. Mayans, and N. Muhammad. Automatically specializing software for finite-state verification. Technical Report 98-4, Kansas State University, Department of Computing and Information Sciences, 1998.
- [20] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
- [21] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
- [22] O. Kupferman and M. Vardi. Module checking. In *Proceedings of the Seventh International Workshop on Computer Aided Verification*, July 1996.
- [23] O. Kupferman and M. Vardi. Module checking revisited. In *Proceedings of the Eighth International Workshop on Computer Aided Verification*, July 1997.
- [24] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
- [25] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, Jan. 1984.
- [26] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [27] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [28] G. Naumovich, G. Avrunin, L. Clarke, and L. Osterweil. Applying static analysis to software architectures. In *LNCS 1301*. The 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sept. 1997.
- [29] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1985.
- [30] J. Wing and M. Vaziri-Farahani. Model checking software systems: A case study. *Software Engineering Notes*, 20(4):128–139, Oct. 1995. Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.
- [31] P. Wolper. Specifying interesting properties of programs in propositional temporal logics. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, St. Petersburg, Fla., Jan. 1986.
- [32] M. Young, R. Taylor, D. Levine, K. Nies, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):64–106, Jan. 1995.