

A Survey of Tools for Model Checking and Model-Based Development

Elisabeth A. Strunk, M. Anthony Aiello, John C. Knight, Eds.

Technical Report CS-2006-17
Department of Computer Science
University of Virginia
June 2006

Preface

Model checkers and model-based development tools are becoming increasingly prominent in industrial practice. There are a wide variety of these tools available, with a number of different capabilities suited to different kinds of problems. We felt that the variety poses two problems, however. First, it is difficult for researchers outside of these specific domains to understand what capabilities exist in practice, and so to avoid duplicating theory that is already embodied in a tool. Second, while the variety is of great benefit to practitioners, it is intimidating to know which tool to choose for a particular problem when no comprehensive discussion comparing and contrasting the different tools is available. We felt that to help with the technology transfer of the state of the art in software engineering, a better characterization of the different tools and the problems they can help solve is necessary.

In the Spring of 2006, we led a graduate seminar attempting to study some of these tools, characterizing their capabilities and comparing them with other tools in their class. Each student in the class studied one tool in depth, reported back to the class on what they found, and created a set of simple exercises that the class completed to get a feel for the tool. Also, the group of students working on model checkers and the group working on model-based development tools each met to compare and contrast the tools they studied.

This report is the collection of final reports the students wrote for the class. It is by no means comprehensive: it covers only the tools the students studied. We hope, however, that it will provide a helpful starting point for the interested reader to learn more about what these tools can provide.

Elisabeth Strunk
Tony Aiello
John Knight
Charlottesville, VA 2006

Table of Contents

Model Checking Tools

SLAM and BLAST	5
<i>Vibha Prasad</i>	
An Introduction to the SPIN Model Checker	20
<i>Xiang Yin</i>	
KRONOS: A Verification Tool for Real-Time Systems	30
<i>Na Zhang</i>	

Model-Based Development Tools

Model-Based Development Using Simulink	40
<i>Ben Taitelbaum</i>	
Safety Critical Application Development Environment.....	45
<i>Kendra N. Schmid</i>	
<i>Perfect Developer</i> Tool Suite.....	53
<i>Michael Spiegel</i>	
SCRTool and the SCR Specification Language	58
<i>Patrick John Graydon</i>	

Model Checking Tools



SLAM and BLAST

Vibha Prasad

I. INTRODUCTION

SLAM and BLAST are both software verification tools that perform static analysis of C programs to find if the program satisfies a certain property. Both these tools are relatively new. SLAM was developed by Microsoft Research around 2000 and BLAST was developed by the University of California, Berkeley around 2002. Figure 1 shows the overview for SLAM/BLAST. These tools work on a C program and take the specification of the property to be checked as its input. This specification is written in the specification language specified by the tool. SLAM and BLAST either verify that the system is safe, i.e. the program satisfies the specified property or give an error trace that violates that property. The error trace can be directly mapped to the original C program.

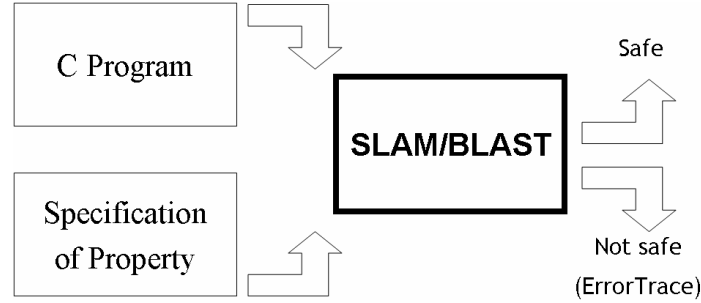


Fig. 1. Overview of SLAM and BLAST (from slides of [3])

This section covers the basic concepts that are a part of both SLAM and BLAST. Both the tools use the concepts of predicate abstraction, model checking, automated theorem proving and program analysis. The main objective behind using all these techniques together is to fully utilize the strengths of each of these techniques and to use their individual strengths to counter each others weaknesses.

A. Motivation

As the size of software increases, it becomes very hard to build and test. Large scale software is generally written by different groups of programmers, and integration testing becomes a major problem. This is of more concern in systems where the reliability of the software cannot be compromised. Such concerns have motivated the software industry to consider alternative techniques for software verification, especially those based on formal proofs. The recent success of SLAM particularly, has led the industry to recognize that formal methods may be just right for the job.

B. Property Checking

The property checking problem is: given a program P and a specific property, does the program P satisfy the given property? This problem has been found to be undecidable. The reason is as follows. A program can be represented with an infinite state space. The problem then translates to finding a finite set of predicates that support the specified property over the infinite state-space of the program. This problem is undecidable and the complete proof is given in [13]. The important thing to remember here is that any

algorithm which tries to find the solution of an undecidable problem is potentially non-terminating.

In algorithms for property checking, the programmer generally provides a partial specification of the software. This specification consists of only the property of interest. The code is then automatically checked for consistency with the specification. This is different from proving the “correctness” of the entire program, as the specifications are not complete and are only concerned with a specific property of the program.

There are some trade-offs in performing automatic property checking. The main one is between the soundness and completeness. An analysis is sound “if every true error is reported by the analysis [1]”. The analysis is complete “if every reported error is a true error [1]”. If the mechanism for property checking is too coarse, some errors will not be reported. Hence, the analysis is not sound. Conversely, if the approach is too conservative, false alarms may be reported. Then the technique is not complete. There are other trade-offs associated with the complexity of the analysis. For example, for the analysis to be precise, more computation has to be done and efficiency is compromised. This is usually unacceptable and some degree of precision is compromised for efficiency.

C. Predicate Abstraction

Predicate abstraction is a method for incrementally constructing conservative abstractions of the software. Predicates are used to abstract the data (variables in the program). Each predicate is represented by a *boolean variable* in the abstract program. A boolean variable represents an expression that evaluates to either true or false. At any point in the program (or program state), the set of predicates tell us about the relationship between the variables in the program in scope at that point. Typically, implementations use CEGAR (Counter-example guided abstraction refinement), (which is used by both SLAM and BLAST), to find predicates. The initial set of predicates is empty (or inferred from the specification of the property) and new predicates are added at each iteration. When the model checker finds an error in the abstraction, it has to be verified that the error is not a spurious error. Spurious traces may be present if the set of predicates is insufficient to prove the property as the abstraction is too coarse. At this point, new predicates are needed for further analysis. Thus, predicates are iteratively added to refine the abstraction until the given property is found to be true or a genuine error trace is found. The final set of predicates must be discovered iteratively.

D. Model Checking

Model checking is an automatic technique for verifying behavioral properties of a model of a system by exhaustively enumerating its states. The program (which can have an infinite number of states) is converted to a finite-state machine representation through abstraction. Then the problem reduces to the reachability analysis of an error state in this state graph.

The main advantage of using model checking is that it is fully automatic. Ideally, by proper selection of the model (abstraction), the properties of interest can be verified in one pass. Practically, the model is refined in many iterations as new predicates are added. Furthermore, model checking tools give an error trace for each error path in the program. This helps in locating and correcting the error in the program. Model checking tools can also compute invariants in the program.

The main drawback of model checking is that it is not scalable to very large systems unless the model is very abstract. The number of states in the finite state representation increases exponentially with the number of variables (“the state explosion problem”). Another drawback of model checking is that it operates only on models. Thus, deriving a model from the program becomes one of the key problems in model checking. A coarse abstraction may not be precise enough to prove the property, and the analysis of a detailed abstraction could be very time consuming.

The traditional approach to model checking has been to build a model of the system and verify it. The actual implementation is done after the model has been successfully verified. In the modern approach (followed by tools like SLAM and BLAST), the model is built from the implementation. That is, the implementation is done first. The software is then used to infer the model and check for properties. The main drawback of using the traditional approach is that the actual implementation might “stray” from the correct implementation of the model. In that case, the verified model is not equivalent to the actual implementation.

E. Automated Theorem Proving

Automated theorem proving deals with the development of computer programs to prove mathematical theorems. The main objective is to show that some statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses). Initially, a set of axioms and hypotheses are given and new inference steps are produced by following some rules of inference.

The advantages of automatic theorem proving is that it can handle unbounded domains naturally and it is a good implementation for certain problems (equality with uninterpreted functions, linear inequalities, combination of theories etc.). There are some disadvantages of using automatic theorem proving. Automatic theorem proving generally requires invariants for the analysis. This usually means that some human interaction is needed with the theorem prover. Also, automated theorem proving is very expensive in terms of computational complexity. By combining it with other techniques like predicate abstraction, a considerable difference can be achieved in terms of space and time requirements.

There are some issues raised by [19, 20] regarding the theorem provers used by symbolic software verification engines like SLAM and BLAST. Theorem provers are targeted for efficiency in mathematical reasoning. Some of these features may not be needed in program verification frequently. Similarly, programming language constructs like pointers, structures and unions are not supported by theorem provers. The verification tool encodes these constructs into axioms over some symbols to approximate the semantics of these features. This may interfere with the performance heuristics of the theorem prover often used during axiom-instantiation. Additionally, theorem provers do not support bit vectors and arrays, so operations like equality between bit vectors and integers are not supported. Another problem mentioned in [19, 20] is that when a query is not valid, the theorem provers do not provide concrete counterexamples. This means that the error trace does not contain concrete valuations to the variables in the program and only partial information is provided.

F. Program Analysis

Program analysis offers techniques for statically predicting the dynamic (run-time) behavior of a program at compile-time. Program analysis techniques make approximations about the program, i.e. a model of the program, automatically and operate on these approximations. As constructing a model is one of the key problems faced by model checking, program analysis can be used to overcome this.

Program analysis originated in optimizing compilers for analyzing things like constant propagation, live variable analysis, dead code elimination, loop index optimization etc. The main strengths of program analysis is that it works directly on code, is pointer-aware and the precision-efficiency trade-off is well studied. In spite of these advantages, program analysis is traditionally not targeted for checking temporal properties. However, using it with techniques like model checking and automated theorem proving can provide a powerful tool for software verification. For example, program analysis techniques can detect invariants in a program, which can be used by the automated theorem prover.

II. SLAM

A. Introduction

Model checking tools have been studied in academia for some time, but they were not considered a feasible solution by the software industry. SLAM has generated a lot of interest in the software industry as it successfully applied these concepts to industrial projects. Moreover, since SLAM has been developed by Microsoft Research, which is a part of the industry, and is now a part of a commercial product (SDV (Static Driver Verifier) in Windows), people in the industry have begun to show some confidence in formal verification techniques for software. However, as it is a part of SDV, SLAM is not publicly available at this point in time. Thus all the analysis done here is based on the available literature.

The main goal of SLAM is to check C programs (system software) for temporal safety properties using model checking. Its main application domain has been device drivers in Windows. SLAM has three main components: c2bp (which evaluates a boolean abstraction of the program), bebop (which performs the reachability analysis of boolean programs) and newton (which verifies the feasibility of error paths). SLAM uses zapato as its theorem prover.

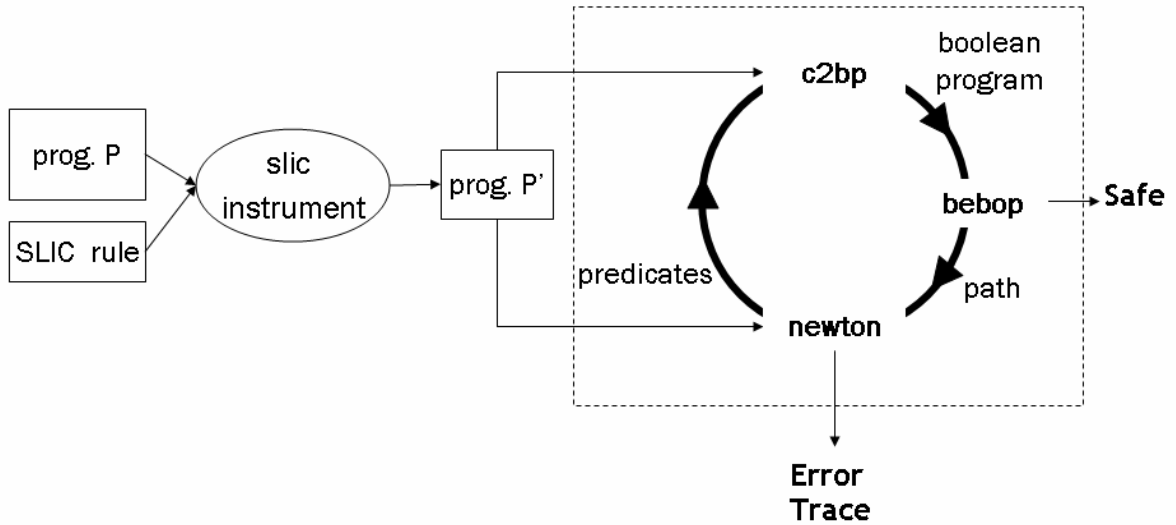


Fig. 2. Working of SLAM ([5])

The specification language used for SLAM is SLIC (Specification Language for Interface Checking). The safety properties to be checked are specified in SLIC. Specifications are described as state machines with a static set of state variables and a set of events and state transitions on the events. The SLIC instrument merges the program P and the C code for the SLIC specification to form a single program P'. This is shown in figure 2. The analysis is then done on the instrumented program P'. Any error path reachable in P' is also reachable in P.

Figure 2 shows the different components of SLAM and how they interact. c2bp takes an input C program (the instrumented program P') and a set of predicates (the SLIC specifications for the first iteration) and converts them to a *boolean program*. A boolean program is a program in which all the variables are of the type boolean. Next, bebop performs a reachability analysis on the boolean program created by c2bp to check if the label error is reachable in P'. If such a path is found, newton checks to see if this path is feasible in the original C program. If newton finds that the path is feasible, an error has been found and SLAM gives the error trace as the output. Otherwise, newton finds additional predicates

that explain the infeasibility. Newton uses the same interfaces to the theorem provers as c2bp for its analysis.

Computing a precise boolean expression (c2bp) is very expensive. It depends on the number of calls to the theorem prover and is linear in the size of the program P and exponential in the size of the set of predicates E . The c2bp algorithm performs modular abstraction of the program when it creates a boolean program. It abstracts each procedure in isolation. Then, within each procedure, it abstracts each statement in isolation. Thus there is no need for c2bp to do any control-flow analysis or find loop invariants.

Bebop is the model checker for boolean programs. It is the most time consuming component of the SLAM toolkit. The states are represented as bit vectors, thus the set of reachable states are represented as a set of bit vectors. This set of bit vectors is computed for every state. States are implicitly represented via BDDs. The complexity of the bebop algorithm is $O(E^{2n})$ where E is the size of interprocedural control flow graph and n is the maximum number of variables in the scope of any label. It tries to compute a fixpoint by iterating over the set of facts associated with each statement.

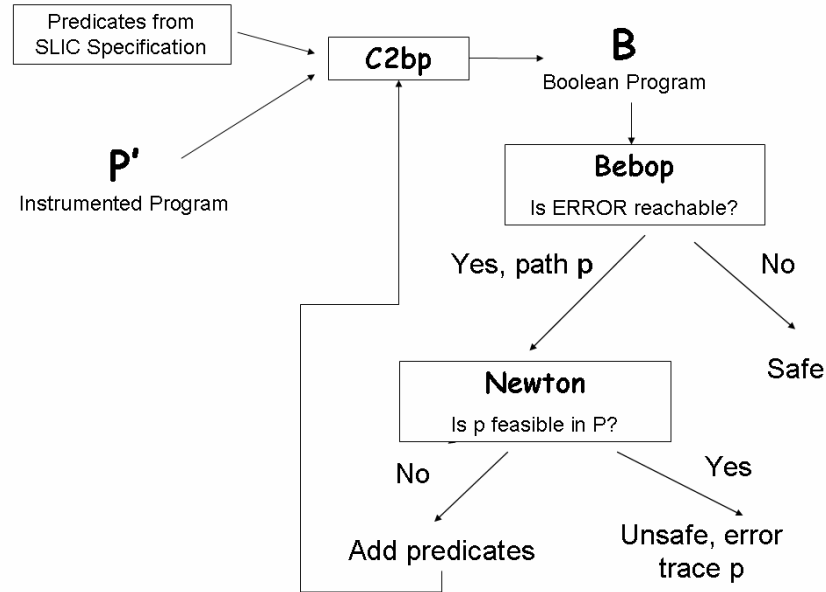


Fig. 3. Components of SLAM (from slides of [3])

When an error path p is provided by bebop, SLAM first uses newton to symbolically simulate the entire trace and determine whether it is spurious. If the trace is spurious, then newton searches for additional predicates which could eliminate the trace in a refined abstraction. If no new predicates are found, SLAM concludes that the spurious trace is caused by the imprecise abstraction done by c2bp. It then invokes another refinement method, called constrain. Constrain symbolically examines each step of the trace in isolation and attempts to refine the abstract transition relation in order to improve the accuracy of the abstraction using the predicates that are available. By default, both newton and constrain use the theorem prover zapato.

B. Capabilities

Currently, SLAM faces two major problems: 1) dealing with pointers, and 2) dealing with imprecision of alias analysis (as it does not consider data flow). Moreover, SLAM does not scale to very large programs currently.

The main problem encountered by SLAM is with pointers. Abstracting from a language with pointers (C) to one without pointers (boolean programs) is hard. Strictly speaking, C supports only call by value, but with pointers and the address-of operator, call-by-reference can be simulated. This creates a problem, as boolean programs support only call-by-value results. SLAM mimics call-by-reference with call-by-value-result.

SLAM has been shown to successfully check control-dominated properties. It successfully handles recursive and mutually recursive procedures.

[1] states three properties for a good model checker – soundness, completeness and usefulness. The analysis is sound “if every true error is reported by the analysis [1]”. SLAM is considered to be sound with respect to the initial assumptions it made (regarding the logical model of memory, aliasing etc.). The analysis is complete “if every reported error is a true error [1]”, that is, there are no false positives. This has not been found to be true for SLAM and SLAM is currently incomplete. The analysis is useful “if it finds error someone cares about [1]”. As SLAM has been successfully used in device drivers and is a part of SDV now, it is definitely useful.

SLAM claims to be very precise in detecting bugs. SLAM gives out an error trace which can be mapped to the original program directly. Although there is an error trace for each error cause, SLAM may overlook some error causes or report spurious error causes.

The results for SLAM have been encouraging. The largest driver processed by SLAM has approximately 60K lines of code. The largest abstraction analyzed by SLAM has several hundred boolean variables. SLAM claims to get results after 20-30 iterations. Based on this, SLAM also claims that counterexample-driven refinement terminates in practice, as SLAM has always terminated. Out of 672 runs in one set, 607 terminate within 20 minutes.

C. Problem Domain

SLAM was originally developed for addressing temporal safety properties in C programs. It has now been customized for the Windows product, SDV. Thus, it works well for specific domain problems like device drivers. Although the SLAM toolkit has also been tested for multi-threaded software libraries ([3]), the analysis done by the SLAM toolkit has been focused on the application domain of device drivers. Hence, not much is known about its capabilities in other domains. Since the SLAM toolkit is publicly unavailable, its evaluation and comparisons with other tools has only been studied by the Microsoft Research group.

III. BLAST

A. Introduction

BLAST (Berkeley Lazy Abstraction Software verification Tool) was developed at the University of California, Berkeley. The basic concept behind BLAST is the *abstract-check-refine* approach which is also followed in SLAM. Additionally, BLAST uses concepts of *Lazy Abstraction* ([13]) and *interpolation-based predicate discovery* ([9]) which will be discussed later in this section.

The overall methodology is similar to SLAM as shown in figure 4. The inputs to BLAST are a C program and the specification of the property to be checked. `spec.opt` merges these together and forms the instrumented program which contains the label for the error state. This instrumented program is then fed to `pblast.opt` which is the model checker for BLAST. If there are no paths to the specified error label, BLAST considers the system to be safe and generates a proof. Otherwise, it checks if this path is feasible using symbolic execution of the program. If the path is feasible, it generates an output trace. Otherwise, the model is refined further.

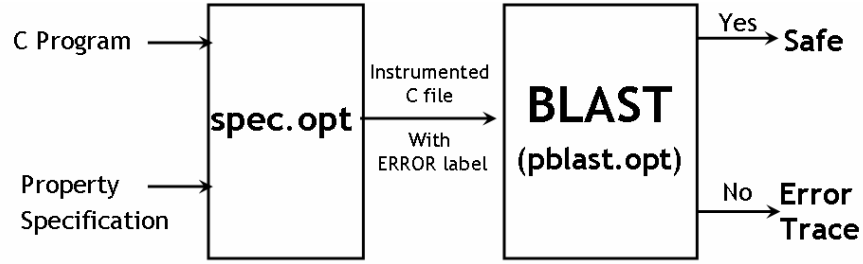


Fig. 4. Working of BLAST ([15])

The BLAST specification language has a very C-like syntax. This makes it easier to learn, especially for C programmers, compared to learning a new specification language. Figure 5 shows an example of the specification for the correct usage of the locking protocol. The specification essentially looks for certain patterns in the original program and inserts some checks and actions to be performed when these patterns are matched. In the example given in figure 5, whenever a function call to `FSMInit()` is detected, the statement `lockStatus = 0` will be inserted.

```

1 global int lockStatus = 0;
2
3 event {
4   pattern { FSMInit(); }
5   action { lockStatus = 0; }
6 }
7
8 event {
9   pattern { FSMLock(); }
10  guard { lockStatus == 0 }
11  action { lockStatus = 1; }
12 }
13
14 event {
15  pattern { FSMUnlock(); }
16  guard { lockStatus == 1 }
17  action { lockStatus = 0; }
18 }
  
```

Fig. 5. An example of Specification in BLAST ([14])

The architecture of BLAST is given in figure 6. The tool is written in OCaml. It uses CIL(CCured) as the front-end to check for type safety of the C program. CCured inserts run-time checks necessary to prevent all memory safety violations. The program is internally represented as control-flow automata (CFA). A CFA is a directed graph; its vertices correspond to the control points of the program and its edges correspond to program operations. Each edge is labeled either by a block of instructions that are executed along that edge, or by an *assume* predicate. The assume predicate represents the condition that must hold for the transition to take place. The abstraction is constructed during execution and is represented as the Abstract Reachability Tree (ART). BDDs are used for the internal representations. BLAST uses Simplify and vampire as theorem provers.

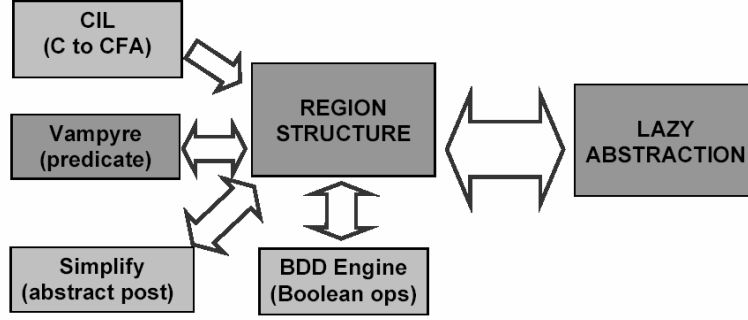


Fig. 6. The BLAST architecture (from slides of [16])

The abstract-check-refine approach (also used in SLAM) has three phases. In the “abstract” phase, a set of predicates is chosen to create an abstraction of the program. Each abstracted state can be represented by the set of truth assignments of the predicates. In the “check” phase, this abstraction is used to check the safety property. If the abstracted model is safe, the original program is considered to be safe. Otherwise, an error path is generated and it is checked whether this path is feasible in the original program. If the error trace corresponds to a spurious error, in the “refine” phase, the abstracted model is refined further by adding more predicates. In BLAST the abstract-check-refine loop is short-circuited ([13]) and the three phases are integrated tightly using lazy abstraction as shown in figure 7. The check phase drives the abstract phase, and hence only the abstract phase is shown.

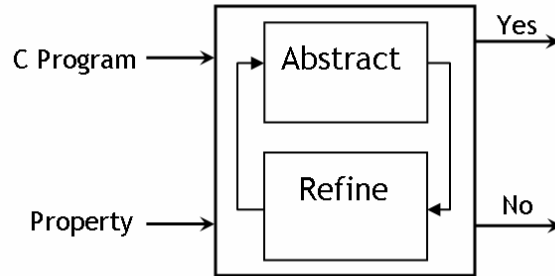


Fig. 7. Working of BLAST (from [15])

There are two principles involved in lazy abstraction: *on-the-fly abstraction* and *on-demand refinement*. On-the-fly abstraction is based on the observation that having the same level of precision may not be advisable for all the regions in the state space as some regions may be unreachable. In lazy abstraction, instead of abstracting the entire abstract model in the abstract phase, regions are abstracted only when they are needed by the check phase. On-demand refinement is based on the observation that after refinement, the model used in the previous iteration may be re-used. In SLAM, the entire model is built from scratch after refinement. In lazy abstraction, the regions in the state space that have been already proved to be safe are not refined again. In the refine phase, refinement is done starting at the earliest state at which the control point in the error path (based on the abstraction) does not have a corresponding point in the original program. This state is called the pivot state.

The lazy abstraction algorithm is composed of two phases: the forward-search phase and the backwards counterexample analysis. An abstract reachability tree (ART) is built during execution. It represents a portion of the reachable, abstract state space of the program. Each control point in the program can be

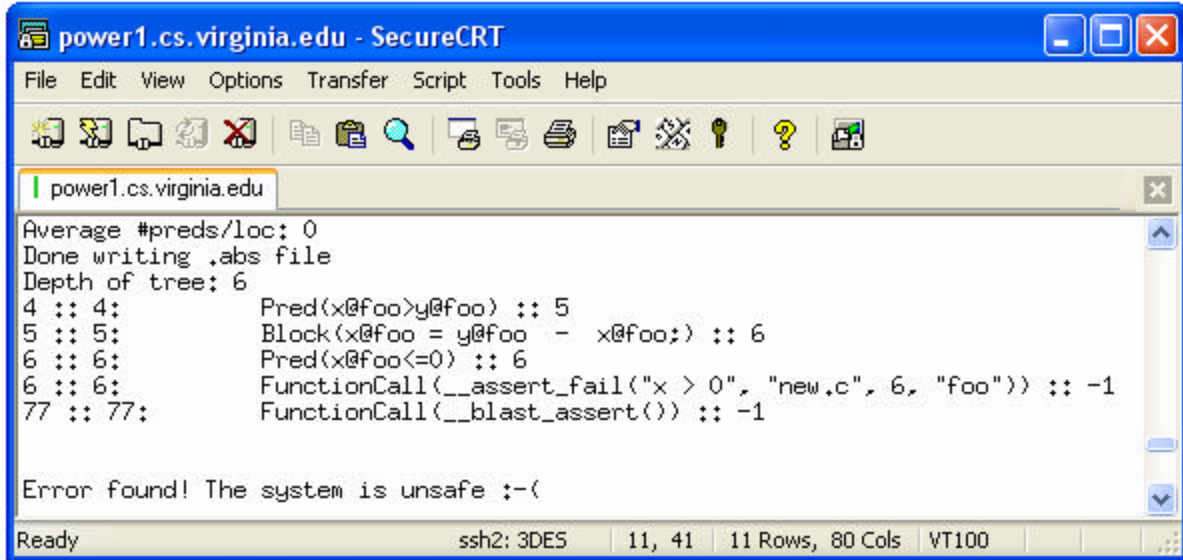
represented as a state. A set of states can be abstracted as a region. If a path to the error region is found, it might be the case that the region has some error states but those are not reachable in the original program. The abstraction is then refined for that region to find out whether the error trace is feasible in the original program.

In the forward-search phase, the abstract reachability tree (ART) is built incrementally. Each path in the tree corresponds to a path in the CFA. Each node of the tree is labeled by a vertex of the CFA and a formula, called the *reachable region*. Each edge is labeled either by a block of instructions or by an assume predicate. The reachable region is a boolean combination of the abstraction predicates. It represents what is known about the state of the program in terms of the predicates under consideration, after executing the instructions from the root node to the current node. The reachable region of a node is obtained from the reachable region of the parent node and the instructions on the edge from the parent node to the current node. If an error node is reachable in the tree, then the next step is the backwards counterexample analysis.

In backwards counterexample analysis, a theorem prover is called, to check whether the error is real or results from the abstraction being too coarse. If it is a spurious error, the theorem prover suggests new abstraction predicates which rule out that particular spurious counterexample. The program is then refined locally by adding the new abstraction predicates only in the smallest subtree containing the spurious error. The search then continues from the point that is refined (the pivot state), without touching the part of the reachability tree outside that subtree. By iterating over the two phases of forwards search and backwards counterexample analysis, different portions of the reachability tree will use different sets of abstraction predicates.

There are many advantages of the lazy abstraction approach. First, only the reachable part of the state space is abstracted, which is typically much smaller than the entire abstract state space. Secondly, different parts of the state space have different precisions. Thus, fewer predicates have to be processed at every point. Lastly, model checking is not repeated for those parts of the state space that are known to be error-free (from some coarser abstraction).

When an error trace is generated, it is simulated to check if the path is feasible in the original program. BLAST uses Interpolation-based predicate discovery for discovering new predicates. The predicates at any node (control point in CFA), can be computed by interpolating between the predicates at the previous node and the next node in the CFA. Interpolation-based predicate discovery finds the minimum set of predicates needed at a node (control point in CFA) to go from the previous node to the next node. Suppose A and B are two sets of predicates, and $A \rightarrow B$. Craig interpolant is a (minimum) set of predicates C , such that $A \rightarrow C$ and $C \rightarrow B$. The advantage of using this method is that just recording the interpolants along an execution path is enough. This reduces the storage requirement as less number of predicates are stored at every node.



```
power1.cs.virginia.edu - SecureCRT
File Edit View Options Transfer Script Tools Help

power1.cs.virginia.edu
Average #preds/loc: 0
Done writing .abs file
Depth of tree: 6
4 :: 4:      Pred(x@foo>y@foo) :: 5
5 :: 5:      Block(x@foo = y@foo - x@foo;) :: 6
6 :: 6:      Pred(x@foo<=0) :: 6
6 :: 6:      FunctionCall(__assert_fail("x > 0", "new.c", 6, "foo")) :: -1
77 :: 77:     FunctionCall(__blast_assert()) :: -1

Error found! The system is unsafe :-(<

Ready          ssh2: 3DES      11, 41   11 Rows, 80 Cols  VT100
```

Fig. 8. Error Trace in BLAST

B. Capabilities

BLAST is used for statically analyzing a C program. It uses lazy abstraction to reduce unnecessary abstraction refinement. This reduces the space and time requirements considerably. BLAST separates the internal data structures of the symbolic abstraction from the model checking algorithm. One of the reasons for this was to facilitate reuse of code to build model checkers for different front-ends (for other languages like java).

Currently, BLAST claims to handle all syntactic constructs of C, including pointers, structures, and procedures. However, integer arithmetic is modeled as infinite-precision arithmetic (no wrap-around), and a logical model of the memory is assumed. In particular, BLAST disallows casting that changes the layout pattern of the memory, disallows partially overlapped objects, and assumes that pointer arithmetic in arrays respects the array bound. Also, procedure calls are handled using an explicit stack and recursive functions are not supported.

```

power1.cs.virginia.edu
0 :: 0:      FunctionCall(__BLAST_initialize_tut3,i()) :: -1
0 :: 0:      Block(Return(0);) :: -1
-1 :: -1:    Skip :: 5
5 :: 5:      Block(x@main = 0;y@main = 0;) :: 7
7 :: 7:      Pred(true) :: 8
8 :: 8:      Block(x@main = x@main + 1;y@main = y@main + 1;) :: 7
7 :: 7:      Pred(true) :: 11
11 :: 11:     Pred(x@main>0) :: 12
12 :: 12:     Block(x@main = x@main - 1;y@main = y@main - 1;) :: 11
11 :: 11:     Pred(x@main<=0) :: 15
15 :: 15:     Pred(y@main!=0) :: 15
15 :: 15:     FunctionCall(__assert_fail("y == 0", "tut3.c", 15, "main")) :: -1
77 :: 77:     FunctionCall(__blast_assert()) :: -1
No new predicates found!
Writing out .abs file: tut3.abs
Maximum #preds/loc: 0
Average #preds/loc: 0
Done writing .abs file
Exception raised :(Failure("No new preds found !-- and not running allPreds ...")
Ack! The gremlins again!: Failure("No new preds found !-- and not running allPreds ...")
Ack! The gremlins again!: Failure("No new preds found !-- and not running allPreds ...")
Fatal error: exception Failure("No new preds found !-- and not running allPreds ...")

vp9g@power1
: /uf8/vp9g/blast_linux_exe/blast_lab ; █
Ready          ssh2: 3DES  25, 41  25 Rows, 91 Cols  VT100

```

Fig. 9. BLAST unable to detect all the predicates

BLAST is sound with respect to the assumptions made. Soundness here means that every true error is reported by the analysis. If the initial assumptions made by BLAST are considered (about alias analysis, no recursive functions etc.) then it is definitely sound. Figure 8 shows an example error trace from BLAST. Since the problem addressed here in BLAST is undecidable, it is possible that the algorithm may not terminate. However, BLAST claims to have terminated every time, sometimes after the addition of predicates by the user. However, the problem of finding enough predicates remains. There may be instances when BLAST fails to find enough predicates to prove the property. This is shown in figure 9. This happens because the BLAST predicate discovery engine may not be “smart” enough to discover enough predicates to prove the property and hence might fail. This situation can be corrected by human intervention by adding the predicates through a .pred file. However, in large programs, the predicates may not be intuitive for the user. There may be a certain class of programs where the predicate discovery engine fails to discover enough predicates and reports an error (false positives). Similarly, programs which manipulate their variables through aliasing of pointers might satisfy the given property, but since BLAST ignores those statements, a false acceptance (false negative) may be produced. Here also, new predicates may be added to specifically target the aliasing. In both these cases, adding new predicates may help. However, when manually adding predicates, the user should have some knowledge of the internal working of the tool and a thorough understanding of the program. All this is not desirable if it is expected that the tool be used in an industry setting by a junior software engineer.

Like SLAM, ZBLAST is targeted at the general programmers in the software industry. In order to encourage programmers to use BLAST for verification, an eclipse plug-in has been developed for BLAST

[8]). Thus software verification and software development can be done side by side. This is definitely aimed at boosting the popularity of BLAST among general programmers.

A Thread-modular Abstraction Refinement (TAR) algorithm ([10]) has been developed for extending BLAST for multithreaded programs. Also, an algorithm for race checking called CIRC has been developed and tested on nesC code ([7]). The current version of BLAST has options to check for race conditions.

C. Problem Domain

BLAST was developed for checking safety properties in C programs. BLAST has been used to verify several large C programs. It has also been used successfully in the domain of device drivers to check temporal safety properties. Most of these device drivers are from the Microsoft Windows DDK or from the Linux distribution ([12]). Blast has successfully verified and found violations of safety properties of large device driver programs up to 60,000 lines of code. The properties checked are similar to those studied in SLAM, mainly locking mechanisms and checking that a driver conforms to Windows NT rules for handling I/O requests. BLAST has found bugs in several drivers. They also claim to have proved that the other drivers correctly implement the specification.

IV. COMPARISON OF SLAM AND BLAST

SLAM and BLAST are based on similar concepts and have many characteristics in common. As has been already mentioned, both the tools perform static analysis and counter-example guided abstraction refinement (CEGAR) to extract a finite state model from a C program. Thus, both these tools face the problem faced by static analysis and property checking, i.e. the possibility of false alarms and non-termination. Both the tools verify safety properties in sequential C programs that are specified by the user. Both have been tested on device drivers and SLAM has been integrated in the product SDV(Static Driver Verifier) with Windows. [17] states that “BLAST is comparable to SLAM in scalability and precision”. It further mentions that there are other tools like MOPS which are much more scalable, but they trade precision for scalability.

Both SLAM and BLAST handle C language constructs (like pointers, structures, and procedures) and assume a logical model of the memory. Both the tools have options for including nondeterministic choices for calls to library functions.

Although both the tools were initially designed for sequential C programs, BLAST is being extended for multithreaded programs (TAR and CIRC). The concept behind SLAM has been used for a tool called Beacon, which checks for interface usage rules in multithreaded software libraries.

There are some differences between SLAM and BLAST. One key difference is the use of lazy abstraction in BLAST. Lazy abstraction allows predicate discovery to be done locally and on-demand and thus saves a lot of space and time. With respect to the specification language, BLAST has an advantage over SLAM. SLIC does not support type-state properties. It monitors only function calls and returns and so, is limited to the specification of interfaces. BLAST, however, considers type-state properties (CIL). Moreover, the BLAST specification language claims to have a C-like syntax which is easier to learn for programmers than learning a new specification language. However, there seem to be only minor differences in both the specification languages. In fact, both the specification languages have a look and feel of aspect-oriented languages.

The abstraction used in SLAM is a boolean program which is constructed before the execution, whereas, BLAST constructs the abstract reachability tree on-the-fly during execution from the CFA. However, internally both SLAM and BLAST use binary decision diagrams for their analysis.

Moreover, SLAM claims that it handles recursive and mutually recursive functions, whereas BLAST does not handle recursive functions as it maintains a stack for procedure calls.

V. COMPARISON WITH OTHER MODEL CHECKING TOOLS

There are many ways in which SLAM and BLAST are different from other model checking tools. In this seminar, we had the opportunity to look at SPIN and KRONOS, in addition to SLAM and BLAST. The comparison in this section will cover the key points in which these model checking tools differ.

First of all, the traditional approach to model-checking (followed by SPIN and KRONOS) has been to first create a model of a system, and once the model has been verified, move on to the actual implementation. SLAM and BLAST fall in the category of the “modern” approach in model checking. The user has already completed the implementation and wishes to verify the software. The objective then is to create a model from the existing program and apply model checking principles, such that the original program is verified. Although, SPIN now has support for abstracting the model automatically from C or java programs, originally, the model for both SPIN and KRONOS had to be specified manually by the user.

The traditional approach has its own drawbacks. Since the implementation is done after the model is verified, there may be a gap in the model that was verified and the actual implementation. This becomes a major problem as the abstraction becomes coarser. KRONOS works on a relatively high-level abstraction of the system. Hence there is no way of actually knowing if the model was correctly followed in the implementation of the system. Although, having a higher-level abstraction is necessary in KRONOS as it works on a timed automaton, and a finer abstraction would mean a very large number of states.

Also, the errors traced in SLAM and BLAST can be directly traced to the program, which is not the case in other model checking tools, SPIN and KRONOS. Errors detected by these tools can be traced to the model, but since the model may not have been implemented yet, nothing can be said about the actual program. This also raises the point of “visibility” of the model. In BLAST, the abstraction is stored as an internal symbolic representation, and hence cannot be directly manipulated by the user. Contrary to this, models in SPIN and KRONOS can be modified by the user directly. Again, if the model has been abstracted by the tool automatically and the user is allowed to make changes to the model directly, the model may no longer be a correct representation of the program.

Secondly, both SPIN and KRONOS have an associated specification language which is used to specify the model. This is different from the specification language in SLAM or BLAST, which is used to specify only the property to be checked. In this sense, SLIC and the BLAST specification language are “partial” specifications that specify the correct behavior of the program.

Thirdly, SLAM and BLAST are used for sequential C programs. This is a very important difference as compared to SPIN and KRONOS. SPIN focuses on interaction between processes, whereas KRONOS aims to verify real-time systems. Since the target systems are different for each of these tools, the levels of abstraction needed for the model are different. KRONOS has the coarsest abstraction of the system and potentially the widest gap between the abstraction and the implementation.

Fourthly, the user of SPIN and KRONOS is assumed to have prior knowledge of formal languages, and the user probably has to learn a new specification language to specify the model for the tool. On the other hand, SLAM and BLAST have been targeted to encourage general programmers to use model checking techniques for software verification. Hence, the user is not expected to have any background in formal languages or to learn a lot of new things. For this purpose, the languages for specifying the properties have a C-like syntax which is familiar to the programmers.

Additionally, SLAM and BLAST are used for checking safety properties only, whereas SPIN is more

developed and checks liveness properties also. In this sense, SLAM and BLAST are actually covering a subset of the functionality covered by SPIN. Although, we must remember that these tools are targeted for the software industry, hence the emphasis is not on having all the functionalities of a traditional model checker. KRONOS and SPIN cover some overlapping properties. However, since KRONOS works on timed automata, potentially many more functionalities can be checked compared to SPIN.

Finally, SPIN has been used for a wide variety of applications like call processing, ring protocol etc. KRONOS has been used to verify real-time systems including the classical CSMA/CD protocol. Although, BLAST can be extended to multithreaded programs, it can never reach the scale of KRONOS. It is yet to be seen if SLAM and BLAST can compete with SPIN, which is considered the standard model checking tool, in terms of functionality.

REFERENCES

- [1] Thomas Ball, Sriram K. Rajamani. "The SLAM Project: Debugging System Software via Static Analysis", POPL 2002, January 2002.
- [2] Thomas Ball, Sriram K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces", SPIN 2001, Workshop on Model Checking of Software, LNCS 2057, May 2001, pp. 103-122.
- [3] Thomas Ball, Sagar Chaki, Sriram K. Rajamani. "Parameterized Verification of Multithreaded Software Libraries". TACAS 2001, LNCS 2031, April 2001, pp. 158-173.
- [4] Thomas Ball, Sriram K. Rajamani. "The SLAM Toolkit". CAV 2001.
- [5] The slides from the PLDI 2003 tutorial. <http://research.microsoft.com/slam/>.
- [6] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "Checking memory safety with Blast". In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, LNCS 3442, pages 2-18, Springer-Verlag, 2005.
- [7] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "Race checking by context inference". In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, ACM Press, 2004.
- [8] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "An Eclipse plug-in for model checking". In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC 2004)*, pages 251-255, IEEE Computer Society Press, 2004.
- [9] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar and Kenneth L. McMillan. "Abstractions from Proofs". In ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, 2004.
- [10] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. "Thread-modular Abstraction Refinement". In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, LNCS 2725, Springer-Verlag, pages 262-274, 2003.
- [11] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. "Software Verification with Blast". In *Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN)*, LNCS 2648, Springer-Verlag, pages 235-239, 2003.
- [12] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre and Westley Weimer. "Temporal-Safety Proofs for Systems Code". In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, LNCS 2404, Springer-Verlag, pages 526-538, 2002.
- [13] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar and Gregoire Sutre. "Lazy Abstraction". In ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, pages 58-70, 2002.
- [14] The BLAST Group. "BLAST User's Manual". October 18, 2005.
- [15] The slides from the SPIN 2005 tutorial. <http://embedded.eecs.berkeley.edu/blast/>.
- [16] Wai Sum Mong. "Lazy Abstraction on Software Model Checking". CSC2108 – project report. <http://www.cs.toronto.edu/~arie/csc2108conf/mong.pdf>
- [17] Hao Chen and Jonathan S. Shapiro. "Exploring Static Checking for Software Assurance". In *Proceedings of 2004 IEEE Symposium on Security and Privacy*, 2004.
- [18] Hao Chen, Drew Dean and David Wagner. "Model Checking One Million Lines of C Code". In *Proceedings of Network and Distributed System Security (NDSS 2004)*, February 2004.

- [19] Byron Cook, Daniel Kroening and Natasha Sharygina. “Accurate Theorem Proving by Program Verification”. ETH Technical Report 464.
- [20] Byron Cook, Daniel Kroening and Natasha Sharygina. “Cogent: Accurate Theorem Proving for Program Verification”. In Proceedings of Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Springer-Verlag Volume 3576/2005.
-

An Introduction to the SPIN Model Checker

CS851 – Tools for Model Checking and Model-based Development Project Report: Tool Description and Analysis

Xiang Yin
xyin@cs.virginia.edu

I. INTRODUCTION

Model checking is a method to verify the correctness of software designs. It is an automated technique that, given a logical property and a model of a system, systematically checks whether this property holds for that model. Thus, when the system itself cannot be verified exhaustively, we can build a simplified model of the system that preserves its underlying design characteristics but at the same time avoids known sources of complexity. The model can then be verified using model checking techniques. Model checking is based on the idea of exhaustive exploration of the reachable state space of a system model. Therefore, currently, it can only be applied to systems which have a finite state space, with bounded states. In practice, this is a severe limitation.

SPIN is among the most popular model checking tools. The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. It has been available freely from the web since 1991, and continues to evolve over many years. SPIN is written in ANSI standard C, and is portable across all versions of Unix, Linux, cygwin, Plan9, Inferno, Solaris, Mac, and Windows. Now it is one of the most widely used model checkers and has a fairly broad group of users in both academia and industry. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM [5].

SPIN is designed for analyzing the logical consistency of concurrent or distributed asynchronous software systems, and is specially focused on proving the correctness of process interactions. Hence typical SPIN models attempt to abstract as much as possible from internal sequential computations. SPIN has been used to detect design errors in distributed applications such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc., ranging from high-level abstract descriptions to low-level detailed codes.

II. CAPABILITIES

A. Summary

In SPIN, the system models are described in a modeling language called PROMELA (Process Meta Language). The language is intended to make it easier to find good abstractions of system designs. Emphasis in this language is on the modeling of process synchronization and coordination, not on computation. It allows for the dynamic creation of concurrent processes while process interactions can be specified with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these.

Given a model system specified in PROMELA, SPIN can either perform simulations of the system's execution or it can generate a verifier in C programming language that performs an

efficient verification to check the logical consistency of the specification. SPIN reports on deadlocks, unspecified receptions, unexecutable code, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. The verifier can also be used to verify the correctness of system invariants, it can find non-progress execution cycles and acceptance cycles, and it can verify correctness properties expressed in next-time free linear temporal logic formulae.

B. Architecture

The Basic architecture of SPIN is illustrated in Figure 1.

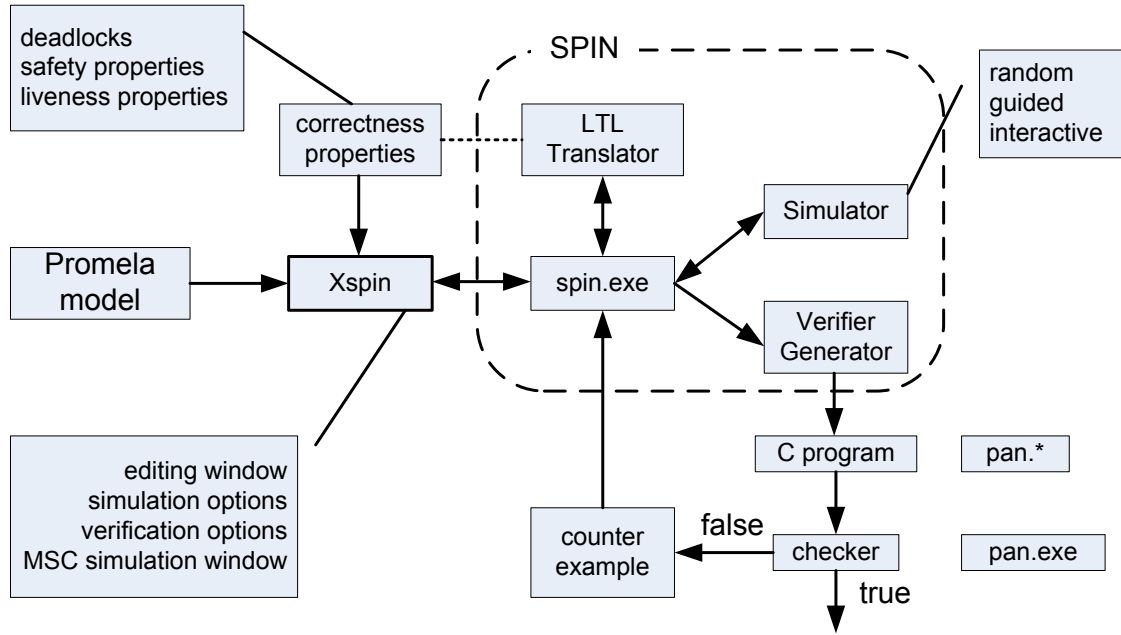


Fig. 1. SPIN architecture [3][4]

SPIN can be initiated from command line or optionally using its graphical interface XSPIN. The user needs to specify a high-level model of a concurrent system, or distributed algorithm, in PROMELA. It can be done either manually or with the help of some mechanical tools. Besides SPIN's built-in correctness requirements (such as absence of deadlock, unreachable code, race conditions), other correctness properties of the model that needs to be checked can be specified as system states or process invariants, as linear temporal logic requirements (LTL), as formal Büchi Automata, or more broadly as general omega-regular properties in the syntax of never claims [5].

Optionally there is a LTL translator that can automatically generate PROMELA correctness claim from a LTL formula. After fixing syntax errors, interactive or random simulation can be performed through the SPIN simulator so that the user can gain some basic confidence that the model has the intended properties. Then, SPIN can generate an optimized on-the-fly verification program in C from the specification of the high-level model and the correctness properties. This verifier is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and it can then be executed to perform an exhaustive or partial verification. If any counter example to the correctness claims is detected, it can be fed back into the SPIN simulator. Guided simulation on the error trail should be inspected in detail to determine the cause of the correctness violation. The model can then be modified to prevent the violation. Once a correctness property has been shown to hold, it is often possible to then reduce the complexity of

that model by using the now trusted property as a simplifying assumption. The simpler model may then be used to prove other properties.

C. Graphical Interface

The graphical interface XSPIN is not a must, but it can ease the whole process of SPIN model checking. XSPIN operates independently from SPIN itself. It provides a clean view of the commands and many options for performing simulations and verifications in SPIN. It executes SPIN commands in the background, in response to user selections on the graphical front-end. Nonetheless, it supplies a significant added value by providing graphical displays of message flows, time sequence diagrams, the finite-state machine corresponding to a PROMELA model, etc.

D. Simulation

As shown above, SPIN has two principal modes of operation: simulation and verification. Simulation is the step-by-step trace of a system execution. Verification requires exhaustive search, whereas simulation does not and therefore can deal with larger state spaces. Since SPIN is targeted at concurrent systems, in the system model there may be several different possible processes enabled at each point of execution, and each process may have several different possible actions enabled at each point of execution. SPIN assumes a non-deterministic scheduling policy. According to how these non-deterministic choices are resolved at the execution time, the SPIN simulation can be of the following three forms: random, interactive, or guided. Simulation is just like testing and debugging. It is used to gain basic confidence that the system behaves as intended. However, just as in all forms of testing, it can only indicate errors and can never show the absence of errors. The most important use of simulation in SPIN is to inspect an error trail (or counter example) to establish, and then remove its cause, when verification finds a property violation.

E. Verification

Perhaps the most important feature of SPIN is that it can generate optimized verifiers from a user-defined PROMELA model. SPIN does not attempt to verify properties of a model directly, with any generic built-in code. By generating a verifier that can be compiled and run separately, we usually can achieve a significant gain in performance [2].

SPIN uses finite automata based model checking. Each process of the checked model is translated into a finite automaton, and the global behavior of the concurrent system is obtained by computing an asynchronous interleaving product of automata, one automaton per asynchronous process behavior. The checked property, representing the violations of correctness properties, is translated into a property automaton (i.e., the never-claim). SPIN checks the given model against the given property by calculating the intersection of the corresponding automata. A non-empty intersection means the possibility of violating a correctness requirement. The verification procedure is based on the reachability analysis of the automata, using an optimized depth-first-search or breadth-first-search graph traversal method (breadth-first-search is only supported by SPIN version 4.0 or later). Since it indeed requires an exhaustive search, there is a state-space explosion problem, just as in all other current model checking techniques. A number of special-purpose algorithms are used to avoid a purely exhaustive search procedure or to reduce the state space (e.g., partial order reduction, state compression, hash compaction, and sequential bitstate hashing). Also, the verification procedure is done on-the-fly, namely, the state space need only be built up to the point where the property can be verified, which means that it avoids the need to preconstruct a global state graph. Therefore if the state space is too large to fit in the SPIN model checker, at least we can have some partial results to look at.

In SPIN, verification is subdivided into two aspects: safety and liveness. Verification of safety is usually done first since it is more critical, and it is easier.

1) Safety Properties

By safety properties, we mean “nothing bad ever happens”. For instance, invariant (e.g. x is always less than 5) and deadlock freedom (the system never reaches a state where no actions are possible) are both safety properties. By default, SPIN will check a set of basic safety properties such as absence of deadlock and unreachable code. It will also check that any user-defined process assertions or invariants cannot be violated.

SPIN check a safety property by trying to find a trace leading to the “bad” thing. If there is not such a trace, the property is satisfied.

2) Liveness Properties

By liveness properties, we mean “something good will eventually happen”. For instance, termination (the system will eventually terminate) and response (if action X occurs then eventually action Y will occur) are both liveness properties. By default, SPIN can check that the system can only terminate in user-defined valid end-states. The PROMELA language includes two types of labels that can be used to define two complementary types of liveness properties: acceptance and progress. In the syntax of linear temporal logic (LTL), an acceptance property corresponds to formulae of the type $\Box\Diamond p$, where p is a user-defined accepting state. When checking for acceptance cycles, the verifier will complain if there is an execution that visits infinitely often an acceptance state. The violation of a progress property corresponds to formulae of the type $\Diamond\Box\neg p$, with p as a user-defined progress state. When checking for non-progress cycles, the verifier will complain if there is an infinite execution that does not visit a progress state infinitely often.

SPIN check a liveness property by trying to find an infinite loop in which the “good” thing does not happen. If there is not such a loop, the property is satisfied.

3) LTL Properties

Many safety and liveness properties can be expressed, and verified, without the use of a formal logic. For instance, the properties can be specified as system or process invariants (using assertions). However, SPIN can be used as a full LTL model checking system, supporting all correctness properties expressible in linear temporal logic (LTL).

For example, the formula $\Box(\text{req} \rightarrow \Diamond\text{ack})$ asserts that at any point in the execution, if a request was made, an acknowledgement is eventually reached. SPIN versions 2.7 and later include a translation algorithm that converts LTL formulae like this into PROMELA never-claims. Never-claims formalize the potential violations of a correctness property, i.e., behavior that should never happen. More specifically a never-claim can be used to represent an automaton, and it is this capability that is exploited by the LTL translator. Although the expressive power of LTL is smaller than that of never-claims, the use of LTL can be simpler and more direct.

F. Model Extraction

There are two basic ways of working with SPIN in system design. The first method is to use the tool to construct verification models that can be shown to have all the required system properties. Once the basic design of a system has been shown to be logically sound, it can be implemented with confidence. It is the traditional way to perform model checking. A second method, which fits in modern model checking techniques, is to start from an implementation and to convert critical parts of that implementation mechanically into verification models that are then analyzed with SPIN.

Automated model extraction tools have been built to convert programs written in mainstream

programming languages such as Java and C into SPIN models. For example, Modex is a tool that mechanically extracts PROMELA models from C programs, and Bandera can extract from Java codes. Various techniques like slicing algorithms have been developed to provide an appropriate level of abstraction. Types and actions in C that cannot be translated to PROMELA (e.g. floating point, pointer) can be embedded into the PROMELA model and compiled into the verifier since the latest SPIN version 4.0. This makes it possible to directly verify implementation level software specifications, using SPIN as a driver and as a logic engine to verify high level temporal properties

III. PROBLEMS TO WHICH THE TOOL IS SUITED

Since SPIN is designed for asynchronous process systems, the problems to which the tool is most suited are proving correctness, especially temporal related properties for concurrent systems, specifically data communication protocols. The obvious applications include generic distributed algorithms (e.g., the leader election algorithm, mutual exclusion algorithms), communications network design problems, or protocol design problems. Some applications of SPIN to real-life problems are the Cambridge ring protocol, the IEEE logical link control protocol LLC 802.2, and fragments of larger protocol applications such as XTP and TCP/IP [1].

In the SPIN literature, we can also see that SPIN has been applied to the verification of data transfer protocols, bus protocols, address registration protocols, error control protocols, requirements analysis, controllers for reactive systems, distributed process scheduling algorithms, fault tolerant systems, hardware-software co-design, asynchronous hardware designs, multiprocessor designs, local area network controllers, microkernel design, operating systems code, railway signaling protocols and circuitry, rendezvous algorithms, security protocols, flood surge control systems, feature interaction problems, Ethernet collision avoidance techniques, self-stabilizing protocols, and so on [1].

Note that, although SPIN can be and has been applied to hardware design, it is designed for checking software systems. It can be applied on complex distributed software systems, provided that the system states are bounded finite after abstraction. Inspiring applications of SPIN in the last few years include the verification of the control algorithms for the new flood control barrier built in the late nineties near Rotterdam in the Netherlands; the logic verification of the call processing software for a commercial data and phone switch, the PathStar switch that was designed and built at Lucent Technologies; and the verification of a number of space missions including Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, etc [5].

As discussed above, SPIN can be used to verify safety properties and liveness properties in these kinds of systems. These properties are mostly temporal properties. As an illustration, take a look at the SPIN model in Figure 2. This is correct PROMELA and it corresponds to Peterson's algorithm for the mutual exclusion problem. It has the shared variable `turn` of type `bool` ($= \{0, 1\}$) and `flag`, an array of bits. The model spawns two processes `user` with `pid` 0 and 1.

We now want to express the safety property that the two processes are never simultaneously in the critical section. The easiest way to do this is to introduce a third shared variable `ncrit` of type `byte`, to count the number of processes in the critical section. So, `ncrit` is incremented when a process enters the critical section and it is decremented when it leaves the critical section. When a process is in the critical section, it is verified that `ncrit = 1` by means of assertion.


```

/* Peterson's solution to the mutual exclusion problem */
bool turn, flag[2];
byte ncrit;

active [2] proctype user()
{
    assert(_pid == 0 || _pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);

    ncrit++;
    assert(ncrit == 1);    /* critical section */
    ncrit--;

    flag[_pid] = 0;
    goto again
}

```

Fig. 2. Peterson's solution to the mutual exclusion problem in PROMELA

SPIN will not find any error in this case since this is a correct mutual exclusion solution. However, in case it is not a correct algorithm and the two users can both enter the critical section at the same execution point, `ncrit` will be increased to 2. Then SPIN will complain about a possible assertion violation to indicate the violation of the mutual exclusion property and give out an error trail.

Besides checking the temporal properties for distributed systems, SPIN can also do some work (albeit limited) on functional properties. A good example would be the optimization problems. Given a model *M* of the problem in PROMELA, with:

- 1) certain costs (or time) associated with states/transitions
- 2) a global cost is updated when a transition is taken or a state is reached.

We want to find the optimal schedule to an end-state with minimum cost. (A typical example of this optimization problem would be the famous traveling salesman problem.) The intuitive way to do this in SPIN is to:

- 1) verify that *M* is error-free
- 2) find the optimal schedule as shown in Figure 3

```

min = guess of (worst case, maximum) cost
do
    verify ◇ (cost ≥ min) holds for M
    if (error) min = cost fi
while (error)

```

Fig. 3. Finding optimal schedule in SPIN [4]

We guess an initial value of minimal cost (which should be larger than the optimal one) and then use SPIN to check the property “eventually cost will be larger than min”. If there is a path to a final state for which the cost is less than min, SPIN will generate an error trail leading to this state. This error trail corresponds to a better schedule. We then modify the minimal cost to

represent this trail and repeat the whole process. The iteration stops when SPIN cannot find error trail, which means no better schedule exists. Thus we know we get the optimal minimal cost.

With the development of the mechanical model extraction for the implementation code to a PROMELA model, SPIN is also suited to ease verifying the properties for some existing implementations. A good example would be the logic verification of the call processing software in the PathStar switch, mentioned above. The application was based on model extraction from the full and unmodified ANSI-C code of the implementation, which was checked for compliance with a group of roughly 20 features formalized in linear temporal logic (e.g., call waiting, conference calling, etc.) [5].

IV. PROBLEMS TO WHICH THE TOOL IS NOT SUITED

The basic limitations (state space explosion) of model checking techniques still apply to SPIN. It can only work with finite and bounded states, with machine limitations. When it comes to checking the model and properties with infinite or unbounded states (e.g. floating point verification), we must use other formal methods. There are experiments with symbolic model checkers that should be able to deal with infinite state spaces, but that is for the future.

SPIN is targeted to check temporal properties, and PROMELA has limited support of internal sequential computations (limited data types and limited operations). It is not powerful enough to verify many algebraic properties or full correctness with respect to the formal specification. In this case, theorem proving is a better choice.

Another limitation of SPIN is that it is not timed. All the timing relations in SPIN are qualitative, not quantitative. This means that properties such as “between event A and event B at least 5 time units should pass” can not be expressed and checked in SPIN. Also, we cannot have an estimate of the execution, either BCET or WCET. There are extensions to SPIN that add real-time features by implementing some sort of dense time in the model. However, SPIN is still well suited for modeling the untimed aspects of the protocol processes and for expressing the relevant (untimed) properties. When serious about verifying timing constraints, one should use a dedicated real-time model checker like KRONOS or UPPAAL, which adopts timed automata in modeling. Perhaps it is better to use SPIN to check the correctness of the model, with real time abstracted, while use other real-time model checker to check the timing constraints.

V. TOOLS COMPARISON

In this section, we briefly compare and contrast the model checking tools we have discussed in the class: SPIN, KRONOS, and SLAM/BLAST. In brief, SPIN focuses on proving the correctness of process interactions; KRONOS is dedicated to validate real time properties, while SLAM/BLAST is specifically targeted at C programs, especially device driver protocols. Since all formal model checkers aim to provide a notation for specifying system design choices, a notation for expressing correctness requirements, and a methodology for establishing the logical consistency of the design choices and the matching correctness requirements, we compare and contrast these model checkers from these aspects. I will also mention other existing model checkers when necessary.

A. Model Specification

In SPIN, the system models are described in a C program like modeling language, called PROMELA. A PROMELA model can be generated either manually from the high-level system design, or mechanically from the low-level implementation with the help of certain model extraction tools. Mechanical extraction can reduce possible errors in generation of system models. Currently there are model extraction tools that support main stream languages such as C and Java.

In KRONOS, each process or component of the system is modeled as a timed automaton, and the whole system model is their product automaton. At the current stage of KRONOS, it does not support mechanical generation for timed automata models, so that they have to be constructed manually by the user. Both SPIN and KRONOS make no assumption on the implementation language, so that their system models can be derived directly from system design decisions with or without any specific kind of implementation. However, it is not the case for SLAM and BLAST. SLAM and BLAST do not have a separate modeling language like SPIN and KRONOS. They are particularly designed to verify C programs and hence only work for C programs at this time. The input system models for SLAM/BLAST are direct C programs. They do have certain internal representation of the abstract system model, such as boolean program (for SLAM) and symbolic abstraction structure (for BLAST). But users are not allowed to modify the abstraction directly as they do with the system models in SPIN and KRONOS.

As for the design decisions that can be conveyed in the system model, these tools also have very different focuses. The emphasis in the PROMELA model for SPIN is the abstraction of synchronizations and interactions among concurrent processes, with limited support of internal sequential computations. PROMELA resembles the programming language C, so that it is usually straightforward to derive a PROMELA model from the system design. PROMELA supports some finite data types, but since it intentionally abstracts out the implementation details of sequential computations, it does not support many elements in modern languages, such as pointers, floating-point types, etc. Timed automata in KRONOS are dedicated to real-time properties. Timing constraints are expressed as predicates on the values of clocks, which is used to model time elapsed between events. Theoretically timed automata can also express any untimed property for concurrent systems. However more human effort needs to be put to derive the model. KRONOS does not support any data type except the integer clock used to express timing properties. Hence usually the timed automata in KRONOS are used to express very abstract high-level system designs. Among other current model checkers, there is one called UPPAAL which also adopts timed automata as the system model. UPPAAL has partial support for data types associated with the timed automata and continues to evolve over these years. So in my view it is more powerful than KRONOS. For SLAM and BLAST, they work for sequential C programs, with partial support for C features like function calls, pointers, floating-point types, and so on. However, they do not look at process interactions and thus do not aim at concurrent systems as SPIN and KRONOS do.

B. Property Specification

Users also need to specify the correctness properties that they want to check for the system model. In SPIN, there are some built-in properties such as deadlocks and unreachable code that can be checked. The user can also specify other correctness properties as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Büchi Automata, or more generally as never claims. Usually users will use assertions, system states, and LTL formula since they are simple to understand and powerful enough for most properties. In KRONOS, properties can be specified either in timed temporal logic (TCTL) as a logical approach or in timed automata as a behavior approach. The case of SLAM/BLAST is still different. They use specifications with C-like syntax, and insert them into the original C programs, just like aspect-oriented programming. BLAST also supports type-state properties with CIL as front-end to parse C programs.

From the above comparisons, we can see that SPIN and KRONOS are targeted at the formal model checking community. They have formal specification for system models and correctness properties and they require people to have knowledge of the specification language or logic they adopt. Once mastered, they will be powerful tools to perform various verifications. On the other

hand, SLAM and BLAST are targeted at C programmers. They don't require users to learn much about the specification language and the underlying logic as far as they know C programming. In turn, their uses are narrowed in the scope of sequential C programs.

C. Capabilities and Limitations

The verification methodologies of all these model checkers are similarly based on exhaustive exploration of the reachable state space of the system model. The detailed search algorithms are different, though. SPIN uses depth-first and breadth-first search on the finite automata; KRONOS uses backward and forward analysis on the timed automata; and SLAM/BLAST searches on its internal reachability tree. Since this course is not focused on the algorithms they adopt, we compare their verification methodologies by their capabilities and limitations.

SPIN is designed for asynchronous process systems. It detects design errors (especially temporal related correctness properties) for concurrent systems, ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges. With built-in properties, SPIN can check deadlocks, unspecified receptions, unexecutable code, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. Other properties that can be specified by invariants (assertions), system states (i.e. non-progress cycles, acceptance cycles), and LTL formula all fit into the category of safety properties (nothing bad ever happens) or liveness properties (something good will eventually happen). However, these properties are untimed, or in other words, of qualitative, not quantitative timing relations. Moreover, SPIN only has limited support for internal sequential computation. KRONOS uses timed automata as its basis so that theoretically it can express and verify all those properties expressible by SPIN. Nevertheless, it introduces the clock into system states which makes the verification of untimed temporal properties much less efficient than SPIN. Therefore KRONOS is usually more suitable to verify real-time properties and timing constraints with dense time. In practice, the large number of clocks is a severe constraint for KRONOS. Since KRONOS does not have data types, it is even worse than SPIN in supporting internal sequential computation. SLAM and BLAST, however, are dedicated to deal with sequential C programs. They manage the data types and computations in C pretty well, but they have limited support for concurrent properties. They almost only check for sequential safety properties such as the ordering or locking/unlocking. Currently BLAST provides some algorithms to check for race conditions in multithreaded C programs. Perhaps it is the best to use SPIN to check the correctness of untimed temporal properties or concurrent systems, use KRONOS to check the real-time constraints, and use SLAM/BLAST to check safety properties in sequential C.

Note that all these model checkers do not support most high-level constructs in modern languages, which made the generation of the system model a bit painful. In view of this, there is a model checker called Bogor. The good thing about Bogor is that its modeling language (BIR) supports many commonly used high-level constructs and is extensible. Furthermore, it is allowed to introduce new abstract data types so that one can have a customized abstract machine for each specific application domain. It might be efficient because then the modeling language can move closer to the abstraction level of the system description used for that particular application domain.

D. Other Issues

All these model checkers suffer from the state explosion problem. The abstract system model must be finite and bounded states, with machine limitations. All of them have implemented a set of optimization algorithms or reduction algorithms to mitigate this problem and to make verification run more effectively, i.e. partial order reduction and bitstate hashing in SPIN, lazy abstraction in BLAST. We haven't established experiment results to compare and contrast the

memory management and efficiency of these tools, though.

SLAM/BLAST does not require the user to learn additional modeling language and any underlying logic. As stated above, they have less steep learning curves than SPIN and KRONOS, although their uses are limited to certain C programs. It is the powerful modeling language and property specification that make SPIN and KRONOS more general model checkers.

As for ease of use, SPIN provides a powerful graphical interface XSPIN, which can run SPIN commands in background and display graphical displays of, for instance, message flows. When it comes to examine an error trail, a graphical display will help to locate the cause of the error more quickly and more efficiently. Also, both SLAM and BLAST provide GUI, and BLAST even has an eclipse plug-in. Unlike SPIN and KRONOS, the error trace for SLAM/BLAST maps back to the original program, not any abstract model. KRONOS does not have strong points in this aspect. It only uses textual input and output. (There are third party tools that implemented some sort of GUI for KRONOS.)

For industrial applications, as we have discussed in section 3, SPIN has been applied to trace logical design errors in distributed systems design, such as operating systems, distributed algorithms, communication network design problems, protocol design problems, railway signaling protocols, flood control systems, call processing systems, mission critical systems, etc. KRONOS has been used to analyze real-time communication protocols like CSMA/CS, FDDI, Philips audio control protocol, timed asynchronous circuits and so on, most of which are associated with real-time constraints. SLAM and BLAST are mainly used for device driver verifications.

REFERENCES

- [1] Holzmann, G. J., "The Model Checker SPIN", *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.
- [2] Holzmann, G. J., *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2003.
- [3] Ruys, T. C., "SPIN Beginners' Tutorial", SPIN 2002, April 2002.
- [4] Ruys, T. C. and G. J. Holzmann, "Advanced SPIN Tutorial", SPIN 2004, April 2004.
- [5] "On-the-fly, LTL Model Checking with SPIN", <http://spinroot.com/>, January 22, 2006.

KRONOS: A Verification Tool for Real-Time Systems

Na Zhang

I. INTRODUCTION

KRONOS [1] is developed by Sergio Yovine. He is working at VERIMAG, a leading research center in embedded systems in France.

KRONOS is a tool developed with the aim to assist the user to validate complex real-time systems. *Real-time systems* are systems that must perform a task within strict time deadlines. Embedded controllers, circuits and communication protocols are examples of such time-dependent systems. These systems are often part of complex safety-critical applications such as aircraft avionics, which are very difficult to design and analyze, but whose correct behavior must be ensured because failures may have severe consequences. Hence, real-time systems need to be rigorously modeled and specified in order to be able to formally prove their correctness with respect to the desired requirements.

KRONOS checks whether a real-time system modeled by a timed automaton [2] satisfies a timing property specified by a formula of the temporal logic TCTL [3]. KRONOS implements a symbolic model-checking algorithm, where sets of states are symbolically represented by linear constraints over the clocks of the timed automaton.

II. MATHEMATICAL FOUNDATIONS

In KRONOS, components of real-time systems are modeled by timed automata and the correctness requirements are expressed in the real-time temporal logic TCTL.

Timed automata are automata extended with a finite set of real-valued clocks, used to express timing constraints. Clocks can be set to zero and their values increase uniformly with time. At any instant the value of a clock is equal to the time elapsed since the last time it was reset. A transition is enabled only if the timing constraint associated with it is satisfied by the current values of the clocks. A timed automaton models the behavior of a single process or component of the system. The locations of the automaton correspond to the different control points of the process. A transition from one location to another location corresponds to the execution of a statement. Timing constraints such as propagation delays, execution times and response times, are expressed as predicates on the values of the clocks.

As a quick view of timed automaton, take CSMA/CD protocol [4] as an example. The behavior of process Sender_i is depicted in Figure 1. Figure 2 shows the behavior of the bus medium in the system.

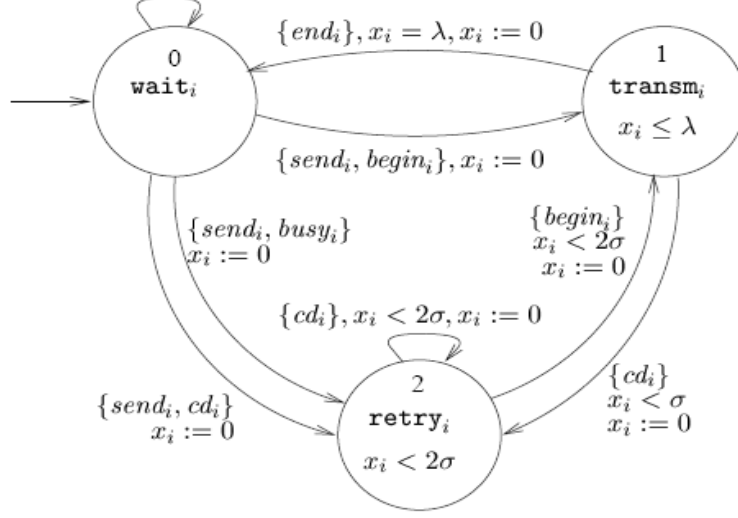


Figure 1: timed automaton for sender_i in CDMA/CD protocol

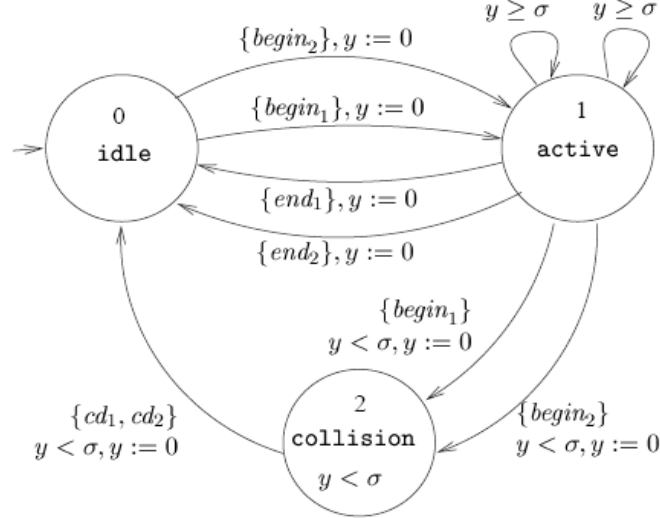


Figure 2: timed automaton for bus in CSMA/CD protocol

III. VERIFICATION

KRONOS checks whether a timed automaton satisfies a TCTL-formula. The *model-checking* algorithm is based upon a symbolic representation of the infinite state space by sets of linear constraints. Basically, KRONOS provides a specification framework that integrates both logical and behavioral approaches to verification. The details of these two approaches will be discussed in the section B and C.

A. Input of KRONOS tool

Figure 3 shows the textual description of the timed automaton of the bus in the KRONOS input language.

```

#locs 3
#trans 9
#clocks X1
#sync BEGIN1 END1 BUSY1 CD1
loc: 0
prop: WAIT1
invar: TRUE
trans:
TRUE => SEND1 BEGIN1; X1:=0; goto 1
TRUE => SEND1 BUSY1; X1:=0; goto 2
TRUE => SEND1 CD1; X1:=0; goto 2
TRUE => CD1; X1:=0; goto 0
loc: 1
prop: TRANSM1
invar: X1<=808
trans:
X1=808 => END1; X1:=0; goto 0
X1<26 => CD1; X1:=0; goto 2
loc: 2
prop: RETRY1
invar: X1<=52
trans:
X1<=52 => BEGIN1; X1:=0; goto 1
X1<=52 => BUSY1; X1:=0; goto 2
X1<=52 => CD1; X1:=0; goto 2

#locs 3
#trans 9
#clocks Y
#sync BEGIN1 BEGIN2 END1 END2
      BUSY1 BUSY2 CD1 CD2
loc: 0
prop: IDLE
invar: TRUE
trans:
TRUE => BEGIN1; Y:=0; goto 1
TRUE => BEGIN2; Y:=0; goto 1
loc: 1
prop: ACTIVE
invar: TRUE
trans:
Y<26 => BEGIN1; Y:=0; goto 2
Y>=26 => BUSY1; ; goto 1
TRUE => END1; Y:=0; goto 0
Y<26 => BEGIN2; Y:=0; goto 2
Y>=26 => BUSY2; ; goto 1
TRUE => END2; Y:=0; goto 0
loc: 2
prop: COLLISION
invar: Y<26
trans:
Y<26 => CD1 CD2; ; goto 0

```

Figure 3: the input file for KRONOS [1]

B. Logical approach

For the logical approach, KRONOS implements model-checking algorithms that check whether the system satisfies a property given as a formula in TCTL. These algorithms can be classified in two general categories according to the strategy applied to explore the state-space: backward analysis and forward analysis. *Backward analysis* algorithms perform a backward search of the reachability graph to compute the set of predecessors of a given set of states. *Forward analysis* algorithms construct the set of successors by performing a forward exploration of the graph.

Given a system modeled as a network of synchronizing timed automata, KRONOS can verify whether it satisfies the correctness criteria specified as formulas of the timed temporal logic called TCTL.

- **Reachability**

Many interesting properties can be stated in terms of the reachability of a given set of states. An example is safety properties: a system satisfies the safety criteria if the system never enters into a state belonging to the set of unsafe states. In TCTL, a safety property is expressed as follows:

$$init \Rightarrow \neg \exists \diamond unsafe$$

where *init* is the predicate characterizing the set of initial states, and the symbol “ $\exists \diamond$ ” means *some state along some execution*.

Another equivalent way of expressing the same property is the following:

$$init \Rightarrow \forall \square safe$$

where the symbol “ $\forall \square$ ” means *every state along every execution*. In words, the above formula says that all the states reachable from the set of initial states are *safe*.

- **Non-Zenoness**

Another important property that has to be shown concerns the *divergence* of time. A good timed model must allow time to elapse without bound. Executions along which time converges are called *Zeno*. A state is called *Non-Zeno* if whatever the system does at the state, it does not prevent time to diverge. Detecting Zeno behaviors is very important: it might be the case that the

system meets the safety criteria by just stopping the flow of time. Clearly, such behaviors are not acceptable.

It has been shown that all the states reachable from the set of initial states are Non-Zeno if and only if the following formula evaluates to true:

$$init \Rightarrow \forall \square \exists \Diamond_{=1} \text{ true}$$

The subscript “=1” means that we are only interested in states that are reachable in a time equal to one. In other words, the above formula states that *every state reachable from init can let time pass one time unit*.

- **Bounded response**

One important property of real-time systems is that they have to respond in bounded time to requests issued by their environment. For instance, a typical requirement is the following: every request from a client has to be served or rejected in at most 5 time units. This property is specified in TCTL as follows:

$$request \Rightarrow \forall \Diamond_{\leq 5} (served \vee rejected)$$

where *request*, *served* and *rejected* are predicates that characterize the set of states corresponding to the reception, the acceptance and the rejection of a request, respectively. The symbol “ $\forall \Diamond$ ” means *some state along every execution*. The subscript “ ≤ 5 ” means that we are only interested in states that are reachable in a time less than or equal to 5.

C. Behavioral approach

For the behavioral approach, KRONOS provides an algorithm that constructs an automaton in which time has been abstracted away in such a way that the causal relationship between events is preserved.

Behavioral equivalences have proven useful for verifying the correctness of concurrent systems. They provide a means for comparing the behavior of two systems, both represented as labeled graphs. The theoretical foundations of the behavioral approach can be found in [15]. Several tools for verifying whether two processes are behaviorally equivalent have been developed. One such tool is ALDEBARAN [5]. Details of this tool are beyond the scope of this report.

D. Summary

The main features provided by KRONOS include:

- KRONOS is able to construct the automaton on-the-fly, that is, while performing the verification. In the current version, that is, version 2.2, this is only possible in the case of reachability properties. The developers are implementing an algorithm that allows on-the-fly verification for full TCTL.
- KRONOS provides both backward analysis and forward analysis. *Backward analysis* algorithms perform a backward search of the reachability graph to compute the set of predecessors of a given set of states. *Forward analysis* algorithms construct the set of successors by performing a forward exploration of the graph. The default analysis is backward analysis.
- If KRONOS finds that the property is indeed not verified by the system, it will output a counter-example, that is, an example of a finite execution that violates the property. The counter-example is written down into the file `_path.reach` and `_trace.reach`. By analyzing the paths leading to the state where the property is not satisfied, we can find the reason and modify the protocol correspondingly.
- KRONOS performs the forward analysis using a breadth-first strategy. A depth-first strategy can be invoked by executing KRONOS with the option `-DFS`.
- The verification of more complex systems should require the use of other features and options provided by the KRONOS. For example, the optimization of the number of clocks in the model, the use of on-the-fly and partial-order techniques, and binary decision diagrams. All these features have an important aspect in common: they try to reduce the size of the state space explored during the

verification. They have been implemented in order to improve both the amount of memory and time required by the verification algorithms.

IV. DISCUSSION

In this section, we discuss the problems that KRONOS is suited and not suited.

As we mentioned above, KRONOS is a software tool built with the aim of assisting designers of real-time systems to verify whether their designs meet the specified requirements. One major objective of KRONOS is to provide a verification engine that can be integrated into design environments for real-time systems in a wide range of application areas.

Some examples of application domains where KRONOS has already been used are: real-time communication protocols such as CSMA/CD, FDDI protocols, Philips audio control protocol, timed asynchronous circuits, and hybrid systems.

KRONOS has also been applied to analyze real-time systems modeled in several process description formalisms such as ATP [8], AORTA [9], ET-LOTOS [10], and TARGOS [11]. Several projects are currently under development to connect KRONOS to specification languages such as SHIFT [12], DIADEM [13], and TCES [14] (Timed Condition/Event Systems.)

While KRONOS can be applied to many areas, it suffers some constraints. Specifically, the problems that KRONOS is not suited include:

A. *The large number of clocks*

KRONOS has been used to verify the FDDI protocol, which has up to 25 clocks. This exceeds the clock-space dimension of similar examples treated in the literature. Therefore, if the number of clocks of a protocol is too large, then KRONOS can not handle it.

B. *Maximum simulation states*

Maximum simulation states and transitions are constraints, although this is true of any model checker.

C. *The constraints of the model*

Since the tool is based on the timed automaton, any system that can not be represented by timed automaton is not suitable for KRONOS. For example, the timed automaton does not support some data types such as boolean and integer variables. Therefore, KRONOS can only work on the high level system abstraction. A lot of details of the system are ignored. One tool that extends the timed automata with more general types of data variables is UPPAAL [7]. More information about this tool will be discussed in section VI.

V. EXTENSION OF THE KRONOS TOOL - TAXYS

KRONOS has been extended and combined with other verification tools such as TAXYS [7]. TAXYS is a tool for verifying real-time properties of embedded systems. The tool connects France Telecom's ESTEREL compiler SAXO-RT with VERIMAG's model-checker KRONOS.

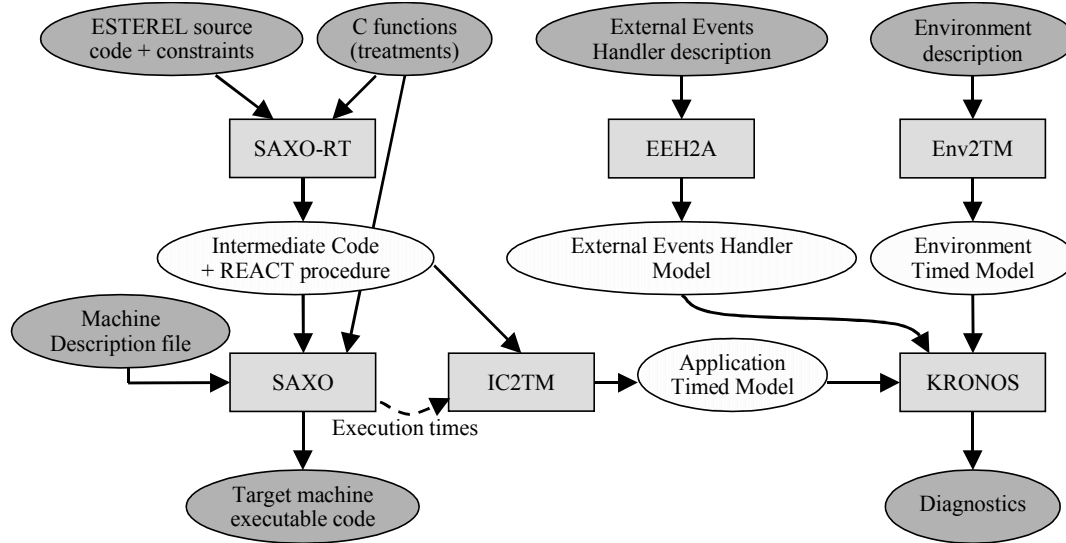


Figure 4: TAXYS general architecture

Figure 4 shows the general architecture of TAXYS. SAXO-RT, SAXO, Env2TM, EEH2A, IC2TM and KRONOS are integrated into the TAXYS environment. TAXYS takes as inputs the description of the application, of the EEH and of the environment, the constraints that have to be verified and the features of the hardware target architecture. From ESTEREL source code annotated with temporal constraints and from C functions, the SAXO-RT tool produces an intermediate code. This intermediate code is compiled by the SAXO tool in order to generate self-sequenced code directly executable by the target architecture. A timed model of the implementation of the application is then produced from the intermediate code by the IC2TM tool. The environment is also modeled as a timed automaton by the Env2TM tool. The EEH is modeled as an automaton by the EEH2A tool. The three automata are then composed, in order to obtain a global model, and analyzed by the KRONOS tool which generates diagnostics if the timing constraints are not satisfied.

VI. COMPARISON WITH OTHER MODEL CHECKERS

In this section, we compare the KRONOS tool with other popular model checkers such as SPIN [16], BLAST [17], and UPAAL [7].

Spin is a popular open-source software tool. It can be used for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.

BLAST is a software model checker for C programs. The goal of BLAST is to be able to check that software satisfies behavioral properties of the interfaces it uses. Blast uses counterexample-driven automatic abstraction refinement to construct an abstract model which is model checked for safety properties. The abstraction is constructed on-the-fly, and only to the required precision. The BLAST project is supported by the National Science Foundation

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.). The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark.

A. Input/Model

As shown in Figure 3, the input file to KRONOS is the textual description of the timed automaton of the corresponding component is the system. It is a high level abstraction of the system. Given a system modeled as a network of synchronizing timed automata, KRONOS can verify whether it satisfies the correctness criteria specified as formulas of the timed temporal logic called TCTL.

SPIN supports a high level language to specify systems descriptions, called PROMELA (a PROCESS META Language). PROMELA is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and borrowing the notation for I/O operations from Hoare's CSP language. SPIN (starting with version 4) provides direct support for the use of embedded C code as part of model specifications. This makes it possible to directly verify implementation level software specifications, using SPIN as a driver and as a logic engine to verify high level temporal properties. SPIN can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Büchi Automata, or more broadly as general omega-regular properties in the syntax of never claims.

In BLAST, the specification language is processed by a command-line tool that takes as input a specification and a list of C source files. A single instrumented C source file is created that combines the input sources and ensures that they satisfy the properties described in the specification.

UPPAAL is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks. The basis of the UPPAAL model is the notion of timed automata too. To provide the a more expressive model and to ease the modeling task, UPPAAL further extends timed automata with more general types of data variables such as boolean and integer variables. In this way, they are trying to develop a modeling/design language which is as close as possible to a high-level real-time programming language with various data types. This may create problems for decidability of model-checking. However, they always require the value domains of the data variables to be finite, in order to guarantee the termination of a verification procedure. By comparison, in the KRONOS tool, data types are not involved. Therefore, the KRONOS can only work on the high level system abstraction.

B. Properties that can be verified

KRONOS can check many interesting properties of a real-time system such as reachability, safety, non-zenoness and bounded response. UPPAAL can check as reachability, safety, deadlock and liveness properties. BLAST focuses on the safety properties of C code. SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. Both BLAST and SPIN can not deal with real-time properties.

C. User Interface

The user interface of KRONOS is simply command line. While on the other hand, UPPAAL provides a nice graphical interface. SPIN provides a friendly user interface, both in command line and graphical. Xspin is an optional but highly recommended, graphical user interface to SPIN. Blast comes with a rudimentary whose chief purpose is to make it easier to view counterexample traces. A good GUI is provided. For example, source and predicate files are loaded in using File in the main toolbar, or by entering the filenames in the appropriate text boxes and clicking the load button.

D. Application

As stated before, some examples of application domains where KRONOS has already been used are: real-time communication protocols such as CSMA/CD, FDDI protocols, Philips audio control protocol, timed asynchronous circuits, and hybrid systems.

SPIN has been applied to many inspiring applications such as flood control, call processing, and mission critical software. For example, logic verification of the call processing software for a commercial data and phone switch, the PathStar switch that was designed and built at Lucent Technologies. The application was based on model extraction from the full and unmodified ANSI-C code of the implementation, which was checked for compliance with a group of roughly 20 class-5 features formalized in linear temporal logic (e.g., call waiting, conference calling, etc.). A cluster of 16 CPUs was used to perform the verifications overnight, every day for a period of several months before the switch was marketed. Perhaps the largest application of software model checking to date.

BLAST has been applied to device driver verification. It is used to check the interface usage rules and lock-unlock property.

UPPAAL has been successfully applied to case studies ranging from communication protocols to multimedia applications. Examples include Bang & Olufsen audio/video protocol, TDMA Protocol Start-Up Mechanism, bounded retransmission protocol over a lossy channels, Lip synchronization algorithm and power-down controller in an audio/video component.

VII. CONCLUSION

We have briefly presented KRONOS, a tool for the verification of real-time systems. This tool is freely distributed through the web for academic non-profit use. The main purpose of the paper has been to illustrate the use of KRONOS with a very simple case study. This example allowed us to present some of the features provide by KRONOS, such as: backward analysis, forward analysis, and generation of counter examples. The verification of more complex systems should require the use of other features and options provided by KRONOS. Such features, whose explanation is out of the scope of this paper, include the optimization of the number of clocks used in the model [18], the use of on-the-fly [19] and partial-order techniques [20]. All these features have an important aspect in common: they try to reduce the size of the state-space explored during the verification. They have been implemented in order to improve both the amount of memory and time required by the verification algorithms.

REFERENCE

- [1] S. Yovine. Kronos: A verification tool for real-time systems. International Journal of Software Tools for Technology Transfer, Vol. 1, Issue 1/2, pages 123-133, October 1997. Springer-Verlag.
- [2] R. Alur and D.L. Dill. A theory of timed automata. Theoretical Computer Science, 126:183-235, 1994.
- [3] T. Henzinger, X. Nicollin, J. Sifakis, S. Yovine. Symbolic model checking for real-time systems. Information and Computation, 111(2):193--244, June 1994. Academic Press.
- [4] A. S. Tanenbaum. Computer Networks. Prentice-Hall, Englewood Cliffs, second edition, 1989.
- [5] Jean-Claude Fernandez and Hubert Garavel and Laurent Mounier and Anne Rasse and Carlos Rodríguez and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. Proceedings of the 14th International Conference on Software Engineering ICSE'14, Melbourne, Australia, 1992. 10
- [6] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, S. Yovine. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In Proceedings of "Conference on Decision and Control, CDC'01". Orlando, December, 2001. IEEE Control Systems Society.
- [7] K.G. Larsen, P. Petterson, Wang Yi. "UPPAAL in a nutshell", Journal of Software Tools for Technology Transfert, vol. 1-1/2, pp 134-152, 1997.
- [8] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. IEEE TSE Special Issue on Real-Time Systems, 18(9):794-- 804, September 1992.

- [9] S Bradley, D Kendall, W D Henderson, and A P Robson. Validation, verification and implementation of timed protocols using AORTA. In Piotr Dembinski and Marek Sredniawa, editors, Protocol Specification, Testing and Verification XV (PSTV '95), Warsaw, pages 193--208. IFIP, North Holland, June 1995.
 - [10] Daws, A. Olivero, and S. Yovine. Verifying ETLOTOS programs with KRONOS. In Proc. Int. Conf. on Formal Description Techniques VII (FORTE'94), pages 227--242, 1994.
 - [11] M. Jourdan, F. Maraninchi, and A. Oliveira. Verifying quantitative real-time properties of synchronous programs. In Proceedings of CAV 93, volume 697 of Lecture Notes in Computer Science, pages 347--358. Springer-Verlag, Berlin, 1993.
 - [12] Deshpande, A. Gollu and P. Varaiya. "SHIFT: A Formalism and a Programming Language for Dynamic Networks of Hybrid Automata". To appear in Hybrid Systems V. LNCS. 1997.
 - [13] Deshpande and S. Yovine. The DIADEM-KRONOS Connection: Bridging the Gap between Implementation and Verification of Hybrid Systems. 1997 Hybrid Systems V Workshop, Notre Dame, IN. workshop.
 - [14] Comparing Timed Condition/Event Systems and Timed Automata. Proc. of HART'97, Grenoble, LNCS 1201:81-86, Springer-Verlag, 1997.
 - [15] R. Milner. Communication and concurrency. Prentice Hall, 1989.
 - [16] Addison-Wesley, The Spin Model Checker: Primer and Reference Manual , ISBN 0-321-22862-6, 608 pgs, cloth-bound.
 - [17] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with Blast *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, LNCS 3442, pages 2-18, Springer-Verlag, 2005.
 - [18] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In Proc. 1996 IEEE Real-Time Systems Symposium, RTSS'96, Washington, DC, USA, December 1996. IEEE Computer Society Press.
 - [19] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model-checking for real-time systems. In Proc. 18th IEEE Real-Time Systems Symposium, RTSS'97, San Francisco, USA, December 1997. IEEE Computer Society Press.
 - [20] F. Pagani. Partial orders for real-time systems. In Proc. 4th Intl. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'96, Uppsala, Sweden, September 1996.
-

Model-Based Development Tools



Model-Based Development Using Simulink

Ben Taitelbaum

I. INTRODUCTION

MODEL-BASED development is a new paradigm in software engineering that is receiving a good amount of attention lately in both industry and academia. In this paper, we explore the most widely-used model-based development tool, Simulink. We start by introducing the model-based development approach, and then specifically the Simulink tool. We then discuss the uses and limitations of Simulink. When appropriate, we differentiate between specific limitations of this tool, and general limitations of model-based development.

A. Model-Based Development

Even the smallest HelloWorld¹ application is difficult for many to think about at the machine code level of strictly 0's and 1's. To handle this complexity, we introduce several layers of abstraction. First, the 0's and 1's can be strung together and replaced with their hexadecimal equivalents. This makes the program much more readable, and less prone to errors (it is very hard to see that a single bit has been entered incorrectly). The program can be further abstracted to the level of assembly instructions, where the programmer may now use mnemonics (strings instead of numbers) for instructions, as well as some other conveniences, like base-10 numbers and string literals, that will be converted into 0's and 1's by the assembler. The next level of abstraction is what is often called a high-level language, and is what most people consider programming. These high-level languages, like Algol, Fortran, C, C++, and Java, allow programmers to use a richer, more natural way of expressing their programs. While our HelloWorld program takes up about 6,000 bytes (or 48,000 0's and 1's) at the machine-code level, at the assembly level it is 400 bytes, a size reduction of 93%. In C, this program is only 80 bytes.

Each of these layers of abstraction was introduced to combat the problem of software becoming ever more complex. Many modern software systems are so complex, that they comprise more than a million lines of high-level code. At this point, it becomes difficult to reason about the system. Model-based software development abstracts the system into abstract, often graphical, representations of subsystems and their interactions in an effort to clarify how the system behaves. As the adage goes, *a picture is worth a thousand words*, so model-based development approaches strive to use pictures instead of words wherever possible.

B. Simulink

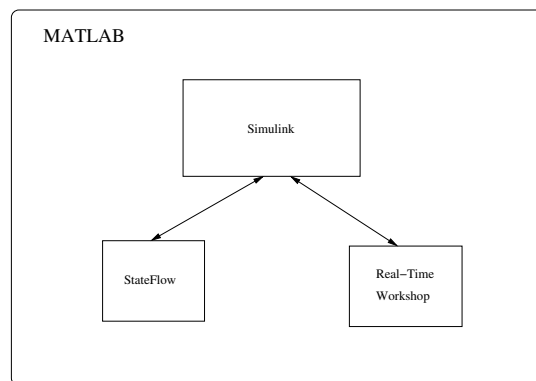
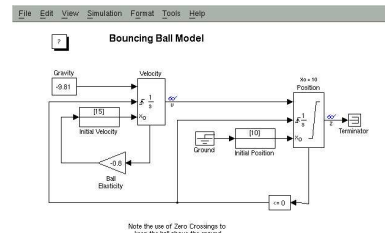


Fig. 1. The functionality of the MATLAB suite is greatly extended by additional toolkits which can interact. The Simulink toolkit controls modeling and simulation, while the StateFlow toolkit adds events and state charts, which can be used in the simulation. The Real-Time Workshop allows for automated code generation from Simulink models.

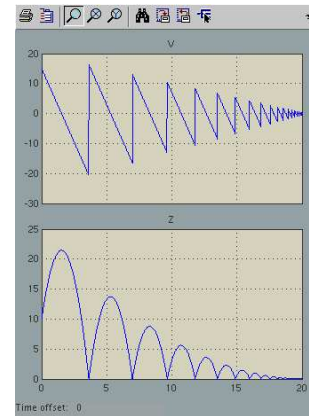
¹the first computer application most programmers learn to write simply prints "Hello, World!" to the screen

Simulink is a model-based development toolkit that can be added on to the popular MATLAB mathematics suite from MathWorks. Because it is not a standalone application, it builds on a rich, well-tested base, and has access to myriad mathematical tools, like graphs and equation solvers.

Simulink is built around the concept of simulating a system, even if all the components of the system are not finished. This allows early feasibility and sanity testing of system components, as well as rapid prototyping. Simulink diagrams have arbitrary levels of zoom, since double-clicking on a subsystem will bring up that component's diagram (which may contain its own subsystems). In figure 2, we see a typical Simulink model (from the Bouncing Ball demo that comes with the toolkit) and the output of its simulation.



(a) A Simulink model depicting a bouncing ball.



(b) The output of this simulation, showing the position vs. time graph (bottom) and the velocity vs. time graph (top)

Fig. 2. An example of a Simulink model with graphical simulation output.

In addition to Simulink, two other MATLAB toolkits can give additional functionality that is often required for complex systems [2], [4], [8]. The StateFlow toolkit adds state charts to the simulation, which can be added just as any other subsystem. Events in the Simulink model may trigger state changes in the StateFlow chart, and state changes in StateFlow may change values in Simulink. This interoperability is a strong argument in favor of using MATLAB as a base tool.

The other relevant toolkit is Real-Time Workshop, which can take Simulink (and StateFlow) models and automatically generate C code. We will discuss the possible problems with this automatic generation in section III-B.

Because of MATLAB's rich graphics libraries, the simulations can produce extremely informative graphs and visualizations. The simulations can also include interactive components, where values are changed and switches are toggled.

Fourteen libraries of components come with Simulink, and include mathematical operators, logical operators, and more advanced libraries like signal routing and model verification. These libraries are shown in figure 3. It is also very easy to create new libraries with user-defined components, providing an extensible tool set. These user-defined components can either be superblocks created out of other components, or special S-functions which can be coded in C or MATLAB. It is necessary to specify how the target language compiler (TLC) should handle these S-functions when using Real-Time workshop's automated code generation. Some tools can build the TLC scripts automatically, but in some circumstances it may be necessary to hand-code these.

II. CURRENT USE

Simulink is most often used in feedback and control systems for embedded devices. Embedded device manufacturers realized long ago the benefit of modeling a device before blindly building it. Simulink gives this modeling capability with the added benefit of actually simulating the device before building it. The Model Verification library provides some assertions that can be added to the model to aid in testing as well.

Unlike embedded device engineers, desktop software engineers often start building a system straight from the requirements. Even if they have formal specifications to start from, they often cannot get realistic simulations until

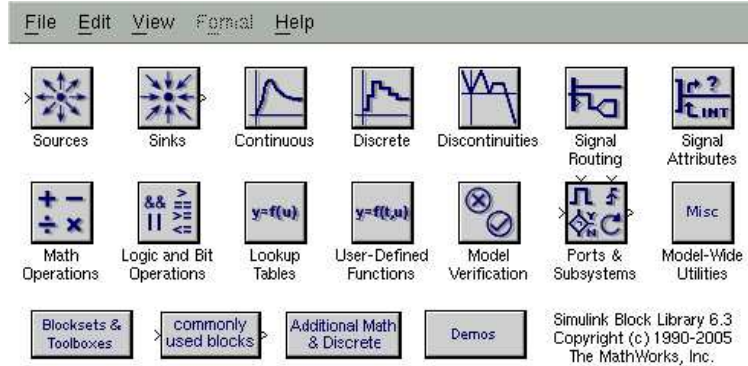


Fig. 3. 14 libraries come with Simulink, but more can be defined.

nearly all of the components have been built. At this stage, it is extremely expensive to learn that some of the system properties have not been satisfied. Still, these software engineers tend to stay with their code-debug paradigm. As many companies are finding out, with current complex desktop software, this methodology costs too much money and time. Because of this, I believe we will start to see more and more projects using model-based development tools, as well as these tools catering more towards the software engineer.

Currently, Simulink is used in many industries ranging from aeronautics [2], [4] to signal processing [6].

III. LIMITATIONS

Since Simulink is a Turing Complete modeling environment, in theory it can describe any system that traditional code can, so it is not limited in this way. The limitations deal more with how the tool functions, and are more social factors than engineering ones.

A. Costs

While many projects have claimed a cost reduction from using Simulink [8], there is always an overhead associated with switching tools. While the actual cost of purchasing the software may be prohibitive for a small startup company, it is nothing to serious companies. The cost of training employees, however, is a factor to consider. As Simulink enters more into the mainstream and is taught by universities, this training cost will not be a factor, since companies can simply hire engineers and managers with the proper backgrounds.

B. Automatic Code Generation

When high-level languages came about, people were worried that the compilers would introduce errors in their assembly code generation. This is the same worry that people now have with automatic code generation from models [1]. Over time, the fear of compilers subsided, and I believe this will be the case with model based development tools as well. What is really needed, however, is a rigorous formal analysis of the code generation, to prove that the generated program is semantically identical to that which has been modeled. This is especially true if testing takes place at the model level.

Another difficulty with automatically generated code is that it is not nearly as optimized as manually generated code. This was also the case with compiled code versus hand-written assembly code, but just as in that case, eventually the benefits of error-free code and reduced implementation time will overcome the optimizations.

C. Precise Semantics

While many Simulink components have widely-understood semantics in the engineering domain, there are no explicit formal semantics associated with these components. For example, the documentation for the `sqrt` function states that, “`B = sqrt(X)` returns the square root of each element of the array `X`. For the elements of `X` that are negative or complex, `sqrt(X)` produces complex results” [3]. While this may be precise enough for most engineers to understand, it is not precise enough for automated theorem provers, or formal analysis of the operation. What

is needed is a formal statement saying that `sqrt` returns a complex number that when squared is equal to the operand. Furthermore, we need to be able to reason from the formal semantics about how precise a result we can expect (depending on the target system).

D. Theorem Proving

While Simulink provides simulation-time assertions, it does not have the capabilities to prove that these assertions always hold. Adding this functionality would greatly increase the applicability of this tool to safety-critical software; however, before this can be added, the tool needs precise semantics (as mentioned in section III-C) and a way of proving that the assertions will still hold after code has been generated. This would also prevent engineers from hand-optimizing or modifying the generated code.

E. Right Tool for the Job

A common engineering aphorism is that the right tool should always be used for the job. Thus, if we have a component whose job is to parse certain regular expressions, many would argue that `perl` would be the correct program for this part of the system. In fact, writing a correct and robust regular expression parser is no trivial task, and highly prone to errors. It would not make sense to divert effort from the main system to build this component in Simulink. A possible workaround for this particular problem would be to write a Simulink component that makes calls to a native regular expression library. This would allow fast prototyping of the system, but would prevent automatic code generation (unless the same library exists for the target platform) and would also prevent any kind of formal reasoning about the correctness of this component.

If Simulink becomes friendlier to typical software engineers, as predicted in section II, then we may very well see operations such as regular expression parsing as Simulink libraries, with precise formal semantics.

IV. COMPARISON

There are currently many other model-based development tools on the market, such as SCR, SCADE, UML2, and PerfectDeveloper, Simulink and SCADE. SCADE is most similar to Simulink, but there are a few features that separate these two applications. First, while Simulink is a cross-platform tool with its own look and feel, SCADE is a Windows application that presents the familiar Multiple Document Interface (MDI) look and feel, which is comfortable for most Windows users. Simulink is more powerful in terms of what can be modeled, because it has more built-in blocks, is easier to define new blocks, and can provide more data types. This last point is worth a closer look; while SCADE provides `boolean`, `float`, and `int` data types for block input and output, Simulink provides both single and double-precision floating point, `int`'s of varying byte-sizes, and most importantly, vectorized inputs and outputs. With StateFlow, it is also possible to model events and state charts, a feature lacking from SCADE. Both tools provide automated code generation, but a comparison of the generated code has not been done. Lastly, an extremely important feature of SCADE that is lacking from Simulink is the ability to perform formal reasoning about assertions. For this reason, SCADE is often used in conjunction with Simulink, where a system is modeled in Simulink and then converted into SCADE where assertions can be proved.

In fact, SCR and PerfectDeveloper also provide formal proof techniques to guarantee assertions. It is possibly because of Simulink's powerful modeling capabilities, along with the absence of formal methods from DO-178B, that MathWorks has not added this functionality. SCR is very limited in what it can model, because it does not model memory or continuous variables. SCR models are very similar to StateFlow charts in the sense that they contain multiple even and condition tables. StateFlow charts are more readable, however, because they have a more advanced interface, and mix between graphical representations and textual table-based representations, whereas SCR always uses tables.

PerfectDeveloper is a text-based tool, very much like SPARK Ada (although not as mature). In this model, the code, along with pre-conditions, post-conditions, and assertions, serves as the model. It does not provide the simulation tools that SCADE and Simulink have, where values can be inspected and visualized, but rather relies more heavily on formally proving assertions. In this sense, neither SCR nor PerfectDeveloper allow for rapid prototyping like SCADE and Simulink do, for a system must be provably correct before it is run.

None of the model-based development tools listed above (possibly with the exception of UML2) is suited for gathering non-functional requirements, like timing or resource-load requirements. While a Simulink simulation will often give a good approximation for a system's timing, and even provides synchronization blocks, non-functional requirements can only be tested once the system has been compiled into code for the embedded system (which Simulink will automate).

V. CONCLUSION

Simulink is a powerful model-based development tool that can decrease both the cost and bugs of a system. Teaching Simulink to undergraduates would be extremely beneficial to them for many reasons. First, it would introduce them to the next major paradigm in software development. Next, it is currently being used and investigated by many leading companies, and would give students a competitive edge in the job market. Lastly, tools like Simulink help students visualize how a system is made up of subsystems and what good design really looks like. As a specification tool, Simulink often conveys more information than a written natural language document, because it forces certain features to be considered that are often forgotten in a typical undergraduate specification exercise. While this tool is not yet perfect for all types of development, the domain of its applicability is ever growing, and therefore can be introduced early on in any engineer's education.

REFERENCES

- [1] Paul Bernard. Software development principles applied to graphical model development. <http://www.mathworks.com/mason/tag/proxy.html?dataid=6804&fileid=28446>.
- [2] MathWorks. Mathworks tools help land unpiloted boeing spacecraft. http://www.mathworks.com/company/user_stories/userstory2329.html?by=company.
- [3] MathWorks. Matlab user's manual. <http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.html>.
- [4] MathWorks. NASA uses stateflow and stateflow coder to generate fault-protection code for deep space 1. http://www.mathworks.com/company/user_stories/userstory2332.html?by=company.
- [5] MathWorks. Real-time workshop manual. <http://www.mathworks.com/access/helpdesk/help/toolbox/rtw/>.
- [6] MathWorks. Sandia implements a high-performance radar receiver using mathworks and xilinx dsp design tools. <http://www.mathworks.com/products/simulink/userstories.html?file=4517&title=Sandia%20Implements%20a%20High-Performance%20Radar%20Receiver%20Using%20MathWorks%20and%20Xilinx%20DSP%20Design%20Tools>.
- [7] MathWorks. Simulink user's manual. <http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/>.
- [8] MathWorks. User stories. http://www.mathworks.com/company/user_stories/index.html.
- [9] Martin Törngren and Ola Larses. Characterization of model based development of embedded control systems from a mechatronic perspective. Technical report, Mechatronics Lab, Department of Machine Design. Royal Institute of Technology, KTH, 2004.

Safety Critical Application Development Environment

Kendra N. Schmid

I. INTRODUCTION

There is an increasing amount of reliance placed on computers in aircraft and automobiles. Failures of these systems could cause catastrophic loss of life, thus the computer systems in these applications must be considered safety-critical components. The hardware safety component is often addressed through redundancy. However, where the software that sits on top of the hardware needs to be as safe and reliable, this requirement cannot be achieved through redundancy [12].

This report looks at one tool that aims to create safe code efficiently: the Safety Critical Application Development Environment (SCADE), produced by Esterel Technologies [8]. SCADE is a model-based development tool that comes in a suite that includes the block diagram editor, Safe State Machines Editor, a Simulink Gateway, Simulator, C and Ada code generators, and Design Verifier. The remainder of this paper is divided into sections presenting: background information; the capabilities of the tool; the problems to which SCADE is best suited; the problems to which SCADE is not ideal; and a comparison of SCADE with Simulink.

II. BACKGROUND

SCADE is classified as a model-based development tool. The term “Model-Based Development” (MBD) is not strictly defined, but the general idea is that instead of moving straight from specification to code, an intermediate representation, the model, is created.

This concept of building correct code from a model representing a system has been called “Correctness-by-construction” [3, 7]. Correct-by-construction claims that an implementation accurately represents a model through the use of automatic code generators. Unlike traditional software development processes in which humans develop implementations by hand, automatic translation introduces no errors (if the translator is perfect). Practically, no translator is or will be perfect; the amount of reliance that can justifiably be placed on automatic translators is an open question [1].

Another advantage of having a model is that it facilitates validation and verification, activities that are difficult in a typical software development processes. SCADE, for instance, provides model checking and proof objective analysis capabilities to aid verification and simulation of a model to aid in validation. Furthermore, MBD tools provide graphical input languages that are designed to be familiar to domain experts. This familiarity reduces the potential for miscommunication between domain experts and software engineers because domain experts can be directly involved with the design of the system. SCADE allows control engineers to specify designs very naturally using block diagrams.

III. CAPABILITIES

SCADE is a suite of several components that, together, facilitate the development of safety-critical applications. This section describes those components and how they fit into the development process. The components and their relationships are summarized in Figure 1.

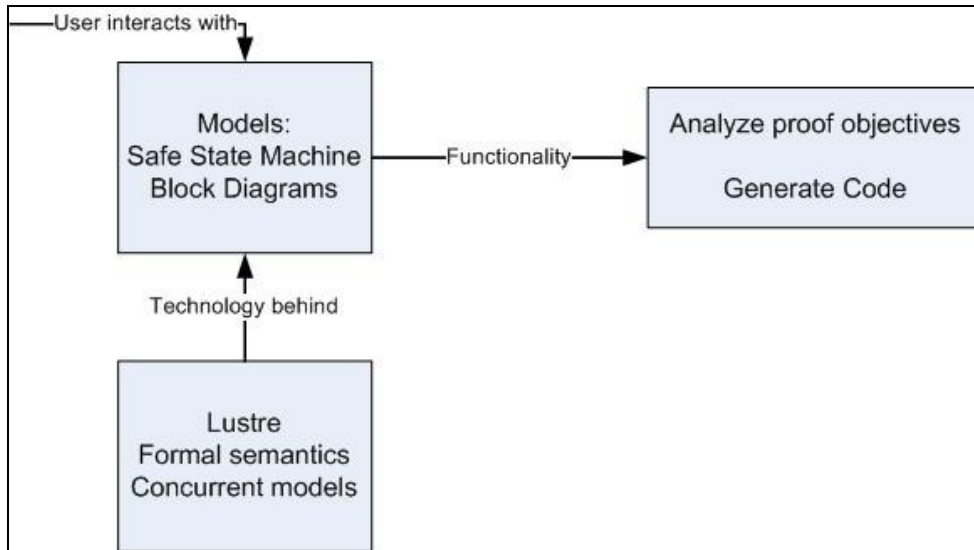


Figure 1: SCADE components

A. Building the Model

SCADE provides two primary ways to build a model. The first is through block diagrams, which are used for modeling continuous control. *Continuous control* systems continuously sample an environment and react to it [3]. The second is through Safe State Machines (SSM), which are essentially hierarchical state machines (these are similar to Statecharts in other systems) [3]. SSMs are used for *discrete control*: reactions to specific external events such as interrupts or user interaction [3]. These two model types can be used in combination.

1) Block Diagrams

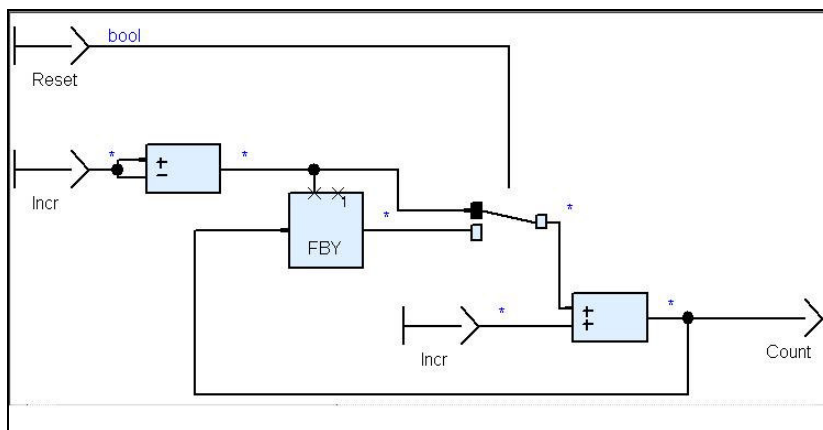


Figure 2: Block Diagram

Block diagrams are models that represent the system in terms of blocks representing components and the interactions between components through connectors, as seen in figure 2. Blocks are hierarchical, and the SCADE tool allows users to inspect the internal structure of blocks used in a diagram such as figure 2. The importance of block hierarchy is that a component can be developed independently of the rest of the model.

The SCADE block diagram editor has a number of features that make creating block diagrams easier. First, a collection of tool bars gives quick access to commonly used components such as addition, multiplication, input, and output. Less commonly used components can be found in a set of libraries that are included with the tool. In addition, one can create new libraries of components that would be useful for sharing common parts within an organization. Second, the editor has a quick check for construction errors. It both alerts the designer if any inputs or outputs are not used based on the declared interface and type checks all inputs and outputs of blocks used in the diagram.

The main domain in which block diagrams are used is control systems. Control systems do not wait for specific events to occur (such as interrupts or exceptions); rather they pull their data at regular time intervals and perform functions on that data. The block models are able to feed calculated results back into the system, a key feature of reactive systems, which sample their environment and recycle changes back into that environment. Continuous control systems can handle discrete values as well as the flow of data that is sampled. The pace at which the environment changes depends on both external events and the reactions that the system makes. Thus, if the feedback from the system is delayed, it does not mean that environmental changes are delayed as well [11, 13]. This is in contrast to interactive or discrete systems in which a process can force the changes to the environment to be delayed.

2) *Safe State Machines*

The other type of model supported by SCADE is the Safe State Machine (SSM). SSMs are built from finite state machines (FSM), in which nothing occurs until an event happens. Events trigger state changes and possible actions [2, 3].

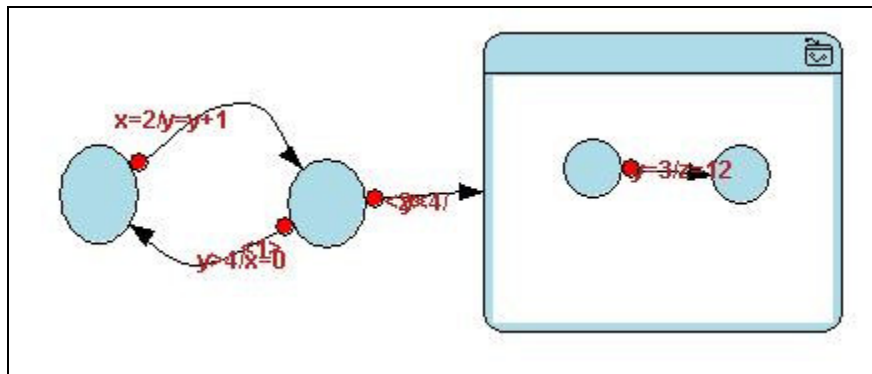


Figure 3: Hierarchical Safe State Machine

SSMs are different from simple FSMs because SSMs do not allow “unsafe actions” such as back-tracking transitions or crossing macrostate boundaries [3]. Macro state boundaries are much like hierarchies in block diagrams: each state can be made up of a smaller state machine. These smaller state machines are not allowed to transition out of their component unless they reach one of the transition locations. Just like block diagrams, SSMs can handle multiple machines running concurrently. In fact, the SSMs are quite often modeled as concurrent finite state machines [2].

The SCADE SSM editor assists in the creation of the SSM model. When creating new transitions between nodes, the editor provides a dialogue box for inputting the event and the effect as well as labels. In addition, the dialogue box has convenient help for the syntax of the transitions, including information about guard conditions.

The SSMs are used to model the discrete portion of the control system interactions. A SSM can be placed inside a block diagram component to represent the discrete event handling.

B. Code Generation

SCADE can generate code in Ada, SPARK Ada, Qualifiable C (v3.1 or v4.2), and standard C. The implementation is synthesized by an automatic code generator that translates the SCADE model into code. The only user interaction that is required is to choose a target language and the root node. The root node is the block diagram that is considered to be the top level and code will be generated for this and all subcomponents of it.

To make ‘safe’ code, the generator follows certain rules. First, the generator makes no extraneous code [10]. This is accomplished by only generating code that directly corresponds with the present model. Thus if a model is changed and recompiled, any old code will be replaced. Second, it does not insert any dead code (code that will never be reached). Both of these keep the code base smaller and compliant with the DO-178B level A requirements [9]. Third, the code generator does not create code that uses pointer arithmetic [10]. This reduces the chance that memory will be overwritten due to arithmetic leading the pointer out of its desired bounds. Finally, the code generated also uses static memory allocation, which means all the memory is allocated at compile time and static analysis can determine whether an architecture has sufficient memory to support the application [10].

No manual modification should be done to the code without acknowledging that any changes will not be reflected in the model or any assertions about the model. Even though it should not be touched, the code generated is quite readable, with numerous comments and function names reflecting the original model.

Another way to increase code safety is the use of a precisely defined language. Using a language such as SPARK Ada that restricts the available features of Ada to those with precise definitions means that all code written using Spark Ada will have only a single interpretation. This creates applications that will lead to the same functionality given different compilations. In addition, using a limited subset with strict definitions allows for rigorous checking of the semantics [1].

The rest of this section discusses how the code is generated from the model.

1) Formal Semantics

The formal typed semantics of the models allow precise translation into code. The types offered for the models line up with types in code, such as Boolean, int, and string. SCADE requires all blocks and inputs to be assigned to a type, which means the code generator does not have to make any arbitrary decisions about the type of a component. In addition, the type checker can determine if there are inappropriate interactions of variable types.

Formal semantics also help the safety of a system because each block provided by the libraries is precisely defined [3]. Investigating the semantics of a block down to its logical component is possible in almost all cases. A few components that deal with mathematical operations on reals, such as square root, are not able to be investigated and will ultimately depend on the compiler library used for compilation of the final code.

2) Lustre

Lustre is used for intermediate representation for supporting cycle computations. In SCADE, both block diagrams and Safe State Machines have a notion of a cycle. In block diagrams, sampling starts again as soon as calculations and reactions from the previous cycle are complete. For SSMs, the cycle starts based on the occurrence of an event. This introduces a paradigm of a stream of data that is accessed whenever a cycle is ready for it. SCADE uses Lustre to support this cycle-based approach.

Lustre is a language that has been developed for synchronous data flow. Data flow is important for SCADE and other uses because it is able to represent the continuous flow of data better than languages and data types that work with discrete values. A user does not need to work directly with Lustre; the

Lustre code is generated automatically from the model.

Lustre is compatible with safety code because it uses clock calculus to give a defined meaning to the data flows [4, 5]. This allows SCADE to check for data consistency as well as time consistency within a program. SCADE can check for properties such as that all flow variables of a calculation must be alive at the time of the calculation [4].

3) *Concurrency in models*

SCADE supports concurrency by allowing all components (blocks and SSMs) to do calculations simultaneously. This is important because control systems are unlikely to ever exist in isolation, which means there are likely many subsystems that will be running concurrently in a system built using SCADE. Concurrency introduces many problems if the commands in a program interleave in arbitrary ways. By allowing arbitrary interleaving, an application could end up exhibiting non-determinism and testing might not be able to find all of the possible incorrect interactions.

SCADE's solution is to make the final interleaving deterministic by doing cycle fusion [3]. Cycle fusion orders the events using messages that may be sent from one cycle to another. From this deterministic interleaving, deterministic code can be generated.

C. *Analysis*

1) *Proof analysis*

SCADE incorporates a design verifier that can perform basic model checking. Model validation and verification is useful to ensure that unsafe properties have not been introduced into the system. This can be done at many levels in SCADE, from simple syntax checking to the more complex assertion statements [6, 11].

The verification portion checks for user-defined properties through proof objective analysis, which is essentially assertion checking. Assertions are added through observer nodes, which are built in the same way as a normal block diagram node. A node checks the validity of a particular safety property within a specified amount of time (up to five ticks or true at all times) [11]. The observer node can be run in parallel with the main program for testing purposes alone, or it can remain present in the final version as well. If the assertion does not hold, the verifier produces counter examples with links to the failed section.

2) *Simulation*

The SCADE tool can also aid validation through the use of model simulation. SCADE first generates code, then allows the user to manually or automatically step through the cycles. This stepping through cycles can be done at 1, 5, 10, 15, 20, or 25 steps at a time in manual mode. Any variable used in any layer of a block can be watched in textual or graphical form. Textual forms are presented in a list, whereas the graphical representation maps the time versus value.

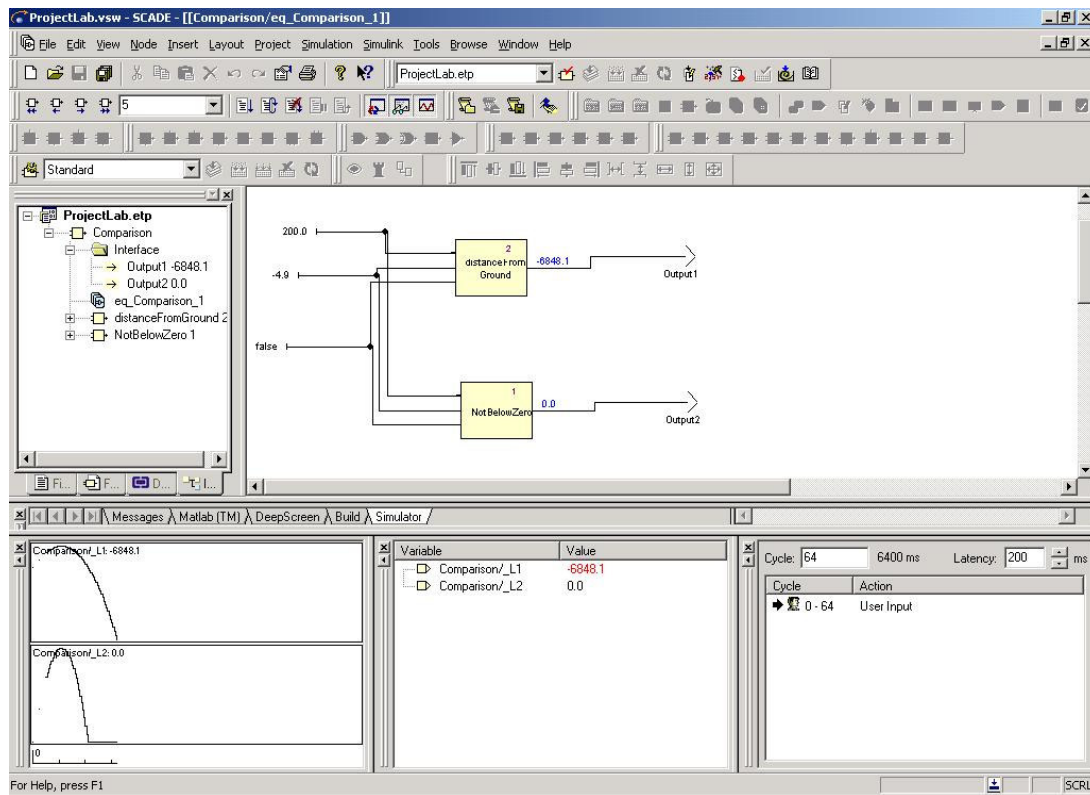


Figure 4: Full screen of a simulation running

D. Other capabilities

In addition to the components that make up the suite to create independent projects, SCADe can also work with DOORS and import Simulink models.

1) Requirements tracing

DOORS is a requirements-tracking system that tracks information about requirements, including the originator, various changes to the requirement, and dependency trees. SCADe is able to create a link with the DOORS database and that can link the requirements stored in DOORS to the models in SCADe. This provides full traceability from the requirement to the code which is important to DO-178B Level A to trace back to the high level requirements [9].

2) Simulink compatibility

SCADe can also import models from Simulink which is another popular model based development tool. Older versions of Simulink did not have code generation or the requirement of strongly typed components [4]. Thus, to increase the formalism it was desirable to transport the already created Simulink model into SCADe to do the final steps of simulation and code generation. Simulink is now able to do some of these functions on its own, but for groups that still feel more comfortable designing and modeling in Simulink, they can still get the added benefit of moving toward the DO-178B compliance with SCADe.

IV. ASSESSMENT OF SCADE

A. *Problems to which SCADE is well suited*

The SCADE suite was designed with safety-critical systems in mind. The industries that SCADE has been used in include avionics, defense, railway, and automotive [8]. SCADE caters to these groups by incorporating features to assist in the certification process. For the DO-178B standard, SCADE includes the generation of code that is traceable to the requirements.

The capabilities that SCADE offers are consistent with what would be required for designing avionics and automotive systems. In avionics, control systems need to constantly check values of their environment such as the altitude, wind speed, or heading. Thus, SCADE's ability to model reactive systems is important.

Another category of problems SCADE would be well suited to is the design and verification of circuits. Circuits also use the continuous control paradigm in which signals are fed back into the system [S_23]. Through the use of design verification, safety properties such as deadlocking could be quickly checked for these systems.

B. *Problems to which SCADE is not suited*

SCADE is not suited to model many types of systems such as multi-threaded or objected-oriented environments. SCADE is only able to generate variations of C or Ada code, both of which are not primarily object orientation. While avoiding object oriented may provide more streamlined code, it can make it more difficult to be understandable to the reader of the code.

SCADE would not be ideal for environments desiring the non-deterministic behavior of concurrent cycles. While predictability of code order is good, it is likely to be less efficient than to allow threads to interweave arbitrarily. Thus for configurations such as distributed systems, generating implementations using SCADE semantics would not be advisable since it would slow the overall performance.

In addition, SCADE is not suited for systems that need to have highly efficient code. While the conversion from block diagram to code does remove the artificial hierarchy structure which will optimize the code some, for the most part performance is not considered. Thus, the code must still be checked for the correct performance requirements.

Using SCADE's automatic code generation is also not ideal for creating device drivers [1]. Device drivers have specific interfaces that must be matched, but the interfaces cannot be effectively implemented through the SCADE models. In addition, the generated code cannot be modified without losing the trustworthiness of the safety assertions. Therefore, any verification advantage gained by using SCADE is reduced by the necessity to change the code to meet the interfaces.

V. COMPARISON TO SIMULINK

Simulink is also a model-based development tool that has many similarities to SCADE. Both tools are primarily used for controls systems and model those systems through block diagrams. Both tools allow for validation through simulation. Also, the newer version of Simulink now offers automatic code generation from diagrams.

The SCADE suite has some advantages over Simulink. First, SCADE has support for proof objective analysis, which is important for verification. Second, SCADE can generate Ada and qualifiable versions of C where as Simulink only offers support for C. SCADE also forces all components to have types, where in Simulink typing is an option but is not strictly required. Finally, SCADE has support for importing Simulink models into its own environment.

Simulink has one main advantage over SCADE: the strong mathematical basis. Simulink is built on top of MATLAB, and thus has integrated support for a number of types of mathematical analyses.

VI. CONCLUSION

SCADE is a model based development tool that can be used to model systems from which proof objective analysis and code generation can be done. The systems are modeled using block diagrams or safe state machines. These allow the modeling of continuous or discrete control systems. In addition, SCADE focuses on safety critical systems by offering features related to the certification process such as code that is compliant with DO-178B standards. The tool is well suited for control systems, but is not well suited for real time based systems.

REFERENCES

- [1] P. Amey, B. Dion, "Combining Model-Driven Design With Diverse Formal Verification", *Embedded Real Time Software*. 2006.
 - [2] C. Andre, "Semantics of SSM (Safe State Machine)," Esterel Technologies, 2003.
 - [3] G. Berry, "The Effectiveness of Synchronous Languages for the Development of Safety-Critical Systems," Esterel Technologies. 2003.
 - [4] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications". In *ACM-SIGPLAN Languages, Compilers, and Tools for Embedded Systems*, 2003.
 - [5] J. Colaço, A. Girault, G. Hamon, and M. Pouzet, "Towards a higher-order synchronous data-flow language". In *Proceedings of the 4th ACM international Conference on Embedded Software*. 2004. ACM Press, New York, NY, 230-239.
 - [6] S. Dajani-Brown, D. Cofer, A. Bouali, "Formal Verification of an Avionics Sensor Voter using SCADE". In *proceedings of International Conference on Formal Techniques in Real-time and Fault-tolerant Systems*, 2004.
 - [7] B. Dion, "SAE Paper # 4AE-129: Correct-by-Construction Methods for the Development of Safety-Critical Applications". In *SAE World Congress*, 2004.
 - [8] Esterel Technologies. <http://www.esterel-technologies.com/>
 - [9] T. K. Ferrell, U. D. Ferrell, "RTCA DO-178B/EUROCAE ED-12B," *The Avionics Handbook*. CRC Press, 2000.
 - [10] J. Gartner, "Efficient Development of Embedded Automotive Software with IEC 61508 Objectives using SCADE Drive," In *Embedded World*, 2006.
 - [11] N. Halbwachs, "Synchronous Programming of Reactive Systems," Kluwer Academic Publishers, 1993.
 - [12] J. Knight, and N. Leveson, "A Large Scale Experiment In N-Version Programming," *Digest of Papers FTCS-15: Fifteenth International Symposium on Fault-Tolerant Computing*, June 1985, Ann Arbor, MI. pp. 135-139.
 - [13] O. Labbani, J.-L. Dekeyser, and P. Boulet, "Mode-automata based methodology for SCADE," In *Hybrid Systems: Computation and Control: 8th International Workshop*, LNCS 3414, pages 386–401. Springer-Verlag, 2005.
-

Perfect Developer tool suite

Michael Spiegel

I. INTRODUCTION

The tool studied in this report is *Perfect Developer*, developed by Escher Technologies. The goal of *Perfect Developer* is to provide a programming environment that combines object-oriented language features with verification guarantees using an automated theorem-prover. This tool was created to address safety-critical software projects that can benefit from an object-oriented approach yet require formal method techniques for a safety analysis.

Perfect Developer is unique for its successful fusion of object-oriented design with static analysis and theorem proving to facilitate design-by-contract. Object-oriented programming has become the dominant paradigm for building large-scale software projects. But the same abstractions that are so popular with software developers are anathema to safety-critical designers. Formal program verification has not gained much acceptance with popular object-oriented languages. The *Perfect Developer* language is successful because it was explicitly designed for static safety analysis. Dynamic method binding is allowed only because an overriding method weakens the precondition or strengthens the postcondition of the overridden method [1]. Inheritance is permitted but the derived class must preserve all the invariants of the base class. The language is strongly typed to ensure that explicit type conversions always succeed. As a consequence of these design decisions, runtime exceptions are not possible in the *Perfect Developer* language. Instead the combination of defensive programming and static contract analysis ensure that exceptional behavior does not occur.

Escher Technologies is a UK-based company founded in 1995. The first version of *Perfect Developer* was released in 2001. *Perfect Developer* version 2.0 was released in 2002 and version 3.0 was produced in 2005. The educational version of the toolset is available free of charge to academic institutions.

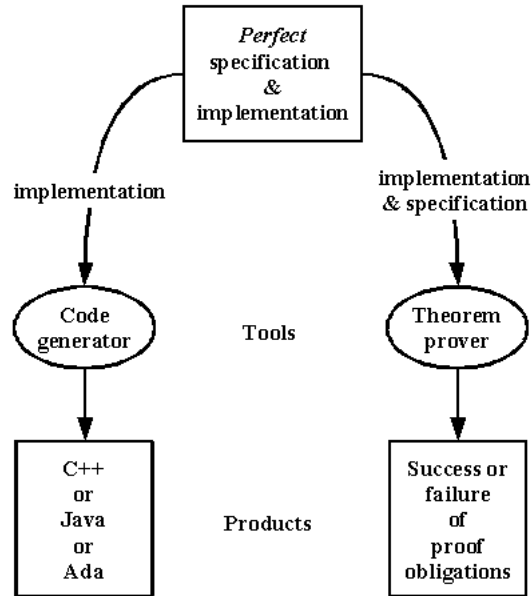
II. TOOL COMPONENTS

The *Perfect Developer* tool is based around the programming language *Perfect*. *Perfect* is both a specification and implementation language not unlike the B-Method [2]. A *Perfect* program is a combination of a specification and implementation. The specification describes the behavior of the program without specifying how the behavior is achieved. The specifications consist primarily of a pair of precondition and postcondition statements on each method of a class. The implementation is a refinement of the specification: it is the algorithm that will be executed to achieve the program specification. A typical program will have interleaved implementation and specification sections; however, it is trivial to identify one section from the other. *Perfect Developer* uses static analysis to enforce design-by-contract. Therefore, the code generator uses only the implementation sections. The program contract is not used during code generation.

Perfect Developer as a tool suite consists of a compiler and an automated theorem prover (Figure 1). The compiler will translate a program from the *Perfect* language into either C++, Java, or Ada. The goal of the *Perfect* compiler is to produce high-level language code that can then be further compiled into a machine or virtual-machine executable format.

The goal of the automated theorem prover is to prove a set of verification conditions, also known as proof obligations. If a program's verification conditions are all proven true, then the program correctly implements its specification. A verification condition is a statement about the program that has been generated from the program's specification and implementation.

Fig. 1. *Perfect Developer* tool suite



A. Proof Obligations

Proof obligations are generated using the design by contract paradigm. In design by contract, a set of preconditions and postconditions together specify a contract between a subroutine and the rest of the program. A subroutine has a set of preconditions that must be satisfied before it can be invoked. And has a set of postconditions that it must satisfy when it has completed execution. If the program agrees to honor its obligations in the precondition, then the subroutine will agree to honor its obligations in the postcondition.

The design by contract paradigm does not dictate the enforcement mechanisms for contracts. Languages such as Eiffel [3] and iContract [4] use runtime checking to ensure that contracts are satisfied. *Perfect Developer* has a static analysis tool that extracts a series of verification conditions from a *Perfect* program. These verification conditions are generated from the preconditions and postconditions in a *Perfect* specification. The static analysis offers a strong guarantee on its proof obligations. With static analysis we know that the proven verification conditions will hold for all possible runtime conditions. Runtime analysis cannot provide this guarantee. Additionally, the execution overhead of runtime analysis is not present in static analysis.

Perfect is designed as an object-oriented language which means it has support for encapsulation, abstraction, inheritance, polymorphism, and dynamic binding. What separates *Perfect* from other object-oriented languages is that these design features are incorporated into the verification conditions for the automated theorem prover. The automated theorem prover generates verification conditions that check for the following conditions:

- "Every method precondition is satisfied at each point of call;
- Every constructor and procedure satisfies its postcondition and postassertions;
- Every function delivers its declared result value;
- When one method overrides another and declares a new contract, the new contract respects the old;
- Class invariants are established by all constructors and preserved by all methods;
- Loop invariants are established and preserved;
- Loops terminate after a finite number of iterations;

- Assertions embedded within an implementation are satisfied;
- Behavioural properties specified by the user are satisfied;
- Explicit type conversions always succeed.” [5]

B. The Language

The *Perfect* language is large by necessity. It is designed as both an implementation and specification language and as an object-oriented language. The language has no less than 99 keywords and 40 operators (listed in the Appendix). Some of these keywords are recognizable from their counterparts in mainstream high-level languages such as C++, Java, or Ada. We encourage the reader to browse through the Appendix and estimate for themselves the learning curve for these language constructs. Even an experienced programmer will need an adjustment period to learn effective programming in *Perfect*.

Perfect has been designed to facilitate proving verification conditions. It contains several design features that are atypical of most programmers’ experiences. The assignment operator has been replaced with postcondition statements. The general form of a postcondition consists of a list of expressions to be changed and a list of predicates to be satisfied. The code generator will attempt to generate intermediate code that performs the necessary assignment operations. These postconditions are subsequently used to discharge verification conditions.

The conditional branching forms of code flow have been replaced with a loop statement that includes (a) a set of local variable declarations, (b) a list of variables which are permitted to change, (c) a predicate list of loop invariants, (d) a list of expressions that are guaranteed to decrease on every iteration but never become negative, and (e) the loop body. The loop invariants are predicates that must be true at the start of the loop and remain true throughout its execution. The loop invariants and decrease variants are necessary to prove loop termination and satisfiability of the loop postconditions.

The learning curve for the *Perfect* language would be much more reasonable if the accompanying documentation was in a better state. The *Perfect Developer Language Reference Manual* [6] is the best source of information on the language. The material is presented in a clear and concise manner. But the reference manual is sorely lacking an index of terms. The only available method of finding the purpose of a particular keyword is to search through every section of the manual. A great investment has been made to produce the *Perfect Developer* tool suite. This investment will not be effectively used in industry and academia until the necessary changes are made to the product documentation. The lack of an index in the language reference manual is the major obstacle to the usability of *Perfect Developer*.

Perfect Developer translates the implementation sections of *Perfect* programs into C++, Java, or Ada. These high-level languages are then either compiled or interpreted by a virtual machine to execute the program. The readability (for a human) of the generated code is poor when compared to the original sections from the *Perfect* code. We will use several examples of obfuscation from the autogenerated C++ code, but similar arguments can be made for the other languages. Primitive datatypes are all aliased by means of type definitions. While this is a great method for assisting portability when compiling on separate machines, the typedef aliases are much harder to read in the code. It is preferable to specify during *Perfect* translation the datatype translation, and then eliminate the extra level of indirection. Variable names are poorly translated to avoid namespace collisions:

```

if (:: _oLess (members [_vLet_p_38_21], x))
{
    i = :: _oSucc (_vLet_p_38_21);
}
else
{
    k = _vLet_p_38_21;
}

```

Finally there are a few lines of code that are simply indecipherable:

```

static const _eHndl < _eModuleDescriptorData > _amoduleData (
    new _eModuleDescriptorData (_eSeq <_eHeapTableEntry > (),
        _eSeq < _eTypeTableEntry > (_eTypeTableEntry (
            _mString ("Table"),
            _mInteger (1),
            _eSeq < _eSeparatorId :: _eEnum > ())),
        _eSeq < _eInstantiationTableEntry > ());
static const _eMDH _aobjLoaderNode (_amoduleData, NULL,
    NULL, NULL);

```

We recommend that namespace collisions be avoided using the standard namespace encapsulation techniques that are present in all three high-level languages. Additionally an effort should be made to transform the *Perfect* translator into a faithful translation of the programming style used in *Perfect* implementations. The rationale for these improvements goes beyond aesthetics. The target market for *Perfect Developer* is safety-critical software that needs to meet software standards such as DO-178B. One component of DO-178B is a set of a software code standards (SCS). It can be argued that an SCS is unnecessary for the high-level languages if an SCS is provided for programming in the *Perfect* language. But a stronger argument for the purposes of software traceability would be a set of conforming standards for both the *Perfect* language and the autogenerated high-level language.

III. TOOL WEAKNESSES

Perfect Developer does not have any mechanisms for enforcing real-time constraints. But on the other hand, neither did any of the other tools studied in this course offer mechanisms for enforcing real-time constraints. Crocker claims that worst case execution timing can be studied in *Perfect Developer* using temporal contracts: “Provided my precondition P is satisfied on entry, I promise to return within time T in a state satisfying my postcondition R .”[5] However there is no support to specify temporal quantities in the current version of *Perfect Developer*. Additionally such discussions of worst-case execution time are very impractical on modern computing architectures with a multitiered memory hierarchy, pipelined execution instruction, and operating system scheduling constraints. Typical execution behavior is much better than absolute worst-case execution time.

Perfect Developer supports bounded memory execution by providing memory management libraries. These libraries require an upper limit on the total number of objects that can be stored at any time. This places a memory bound on the size of the heap. Since the *Perfect* language supports recursion, it must also ensure bounded memory execution on the call stack. Since all recursive calls have verification conditions that ensure termination, we can guarantee an upper bound on the size of the stack. However there is no explicit mechanism for calculating the upper bound of memory used by the program heap and stack.

Perfect has no threading model any other form of support for concurrent programming. Consequently, there is no support for shared memory, message passing, or synchronization primitives. If a threading model were added to the *Perfect* language, the proof checker would need to be augmented in order to prove temporal logic conditions. These temporal logic conditions would look very much like the conditions that are checked by a model checking tools such as SPIN [7]. With the current available technologies, the recommended solution is to design each process independently using *Perfect Developer*. Check the verification conditions per process using the *Perfect Developer* theorem prover, and then combine the various processes at the final step using a model checker. The drawback to this solution is that it does not allow thread-level granularity of concurrent programming.

IV. CONCLUSIONS

Perfect Developer provides a programming environment that combines object-oriented language features with verification guarantees using an automated theorem-prover. This tool was created to address safety-critical software projects that can benefit from an object-oriented approach yet require formal method techniques for a safety analysis. The *Perfect Developer* language has support for encapsulation, abstraction, inheritance, polymorphism, and dynamic binding. What separates *Perfect* from other object-oriented languages is that these design features are incorporated into the verification conditions for the automated theorem prover. The major obstacle to learning *Perfect Developer* is the missing index in the language reference manual. Additionally we recommend improving the high-level language compiler to meet traceability requirements for software standards such as DO-178B.

REFERENCES

- [1] B. Liskov and J. Wing, "A behavioral notion of subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, pp. 1811–1841, November 1994.
- [2] S. Schneider, *The B-method: an Introduction*. Palgrave Macmillan, 2001.
- [3] B. Meyer, *Eiffel: The Language*. Prentice-Hall, 1991.
- [4] R. Kramer, "iContract: The Java(tm) Design by Contract(tm) tool," in *Proceedings of the Technology of Object-Oriented Languages and Systems*, p. 295, IEEE Computer Society, 1998.
- [5] D. Crocker, "Safe object-oriented software: The verified design-by-contract paradigm," in *Proceedings of the Twelfth Safety-Critical Systems Symposium* (F. Redmill and T. Anderson, eds.), (London), Springer-Verlag, 2004.
- [6] Escher Technologies, *The Perfect Developer Language Reference Manual Version 3.0*, 2004. Available at http://www.eschertech.com/product_documentation/Language%20Reference/language_reference.pdf.
- [7] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

APPENDIX: *Perfect* LANGUAGE

Operators

`->, <-, <->, ::, ^=, ..., ~, ==>, <==, <==>, <=, <<=, >=, >>=, <<, >>, <, >, ++, --, +, -, *, **, %, ##, *, %, /, #, .., ^, &, =, |, ~~, [], ., :-, ||`

Keywords

`abstract, absurd, any, assert, associative, axiom, bag, begin, bool, build, byte, change, char, class, commutative, confined, const, decrease, deferred, define, done, early, end, enum, exists, extern, false, final, for, forall, function, ghost, goto, has, heap, highest, identifier, identity, if, import, idempotent, inherits, int, interface, internal, invariant, it, keep, let, like, limited, loop, lowest, map, name, nonmember, null, of, on, opaque, operator, par, pass, post, pragma, pre, proof, property, public, rank, real, redefine, ref, repeated, require, result, satisfy, schema, selector, self, seq, set, storable, super, tag, that, then, those, total, trace, true, until, value, var, via, void, when, within, yield.`

SCRtool and the SCR Specification Language

Patrick John Graydon

I. INTRODUCTION

SCRtool is a prototype tool intended to allow software engineers to create concise, unambiguous requirements specifications for real-time embedded systems [2]. The product of research by Constance Heitmeyer and others at the Naval Research Laboratory (NRL) [2], the tool is designed around the tabular SCR specification notation developed by Katherine Heninger, David Parnas, and others at NRL as part of the Software Cost Reduction (SCR) requirements method [5].

The tool contains four basic components: a specification editor, a consistency checker, a dependency graph browser, and a simulator. We will describe these components in the context of how they could be used to generate and validate a specification for an example embedded control system.

II. SPECIFYING AN EXAMPLE EMBEDDED CONTROLLER

As an example application to guide our discussion of the SCRtool, consider the simple microwave oven depicted in fig. 1. It has a chassis, a microwave emitter for cooking food, an interior light for illuminating the oven cavity, a door switch in order to sense when the door is open, a timer dial to allow the user to select a cooking duration, and a start button that causes the unit to begin cooking. We must create a specification for an embedded computer-based controller that will read input from the door sensor, timer dial, and start switch and control the microwave emitter, light, and timer motor.

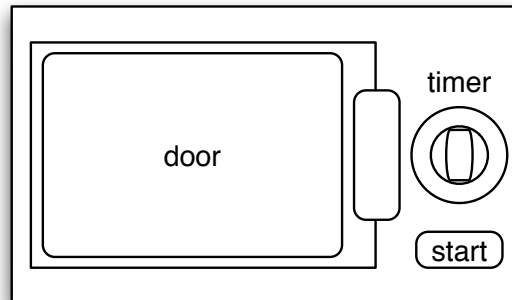


Fig. 1. A simple microwave oven

A. Monitored Variables

In the SCR specification model, the software observes *monitored variables* in the environment through input devices, computes the value of *controlled variables*, and then outputs these values to the environment through output devices. These variables are abstract; they represent the meaning of what an input device reports or what an output device is asked to do without being tied to the specific bit patterns received from or sent to these devices. This abstraction is intended to allow the software to be more easily adapted for use with input or output devices that are similar in function to, but do not encode values in exactly the same manner as, the devices it was first used with.

Before we can define variables for our microwave oven controller, we must first define types for them. Custom types in SCRtool may be either numerated types, as shown in fig. 2, or integer or floating-point types with defined ranges. The `DoorSensorStates` type, for example, is an enumerated type

representing the values that the oven's door sensor can indicate. By defining a custom type that contains the values `Open` and `Closed`, we give ourselves a way to refer to the door sensor states abstractly and avoid littering the remainder of the specification with references to the bit patterns that the controller will actually receive.

To view, add, edit, or remove custom types, we use SCRtool's *type dictionary*. This dictionary, shown in fig. 2, displays the custom types in a specification in tabular notation. Creating a new type is as simple as selecting the add row command from a context menu and using mouse selection and keyboard input to populate the cells in the row.



Fig. 2. SCRtool's Type Editor

After defining types, we define monitored variables for the oven's door, timer, and start button, and controlled variables for the microwave emitter, interior light, and timer motor. In order to represent the state of the door, for example, we define the monitored variable `Door` of type `DoorSensorStates` as shown in fig. 3. Just as we used the type dictionary to manage types, we use the tabular *variable dictionary* to manage variables. Adding new variables is again just as simple as adding a new row and filling in its cells. For each variable we give an initial condition. We will eventually give a table describing how the values of controlled variables ought to be derived, but before we do, we must create a model of the oven's modes as described in the next section.

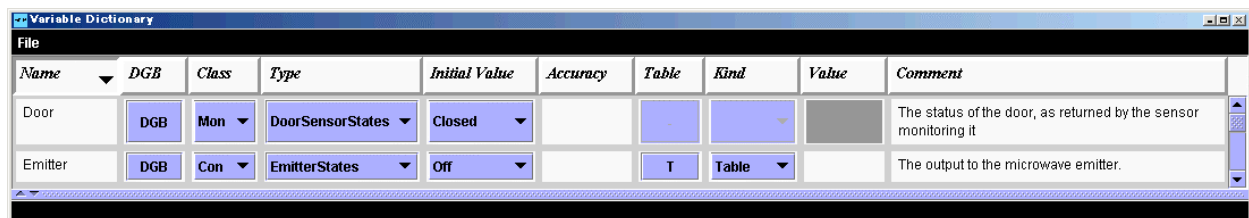


Fig. 3. SCRtool's Variable Editor

In addition to monitored and controlled variables, the SCR specification notation supports *terms* and *constants*. Terms are intermediate variables that represent functions over the monitored variables but do not directly control the actuators. They are used in place of common expressions in the definition of the values of controlled variables, more formally serving one of the needs for which text macros were used in the original SCR tabular notation [5]. Constants replace common literal values, helping to eliminate magic numbers from the specification.

B. Modes

While we could probably specify the value of the outputs of this simple system in terms of logical predicates over the monitored variables quite easily, in many specifications we would find if we attempted this that many of our formulae had common sub-expressions that picked out specific modes of operation. The SCR tabular notation makes these modes, and their effect on the output variables, more explicit through the description of *mode classes* and the *modes* that form their values. In the case of our microwave oven, we identify one mode class, `OvenMode`, with two modes: `Ready` and `Cooking`. In `Ready` mode the oven is standing by, waiting for the user to insert food, select a cooking time on the

timer dial, and press the start button. In `Cooking` mode, the oven cooks the food for the specified length of time, pausing each time the user opens the door to sample the food's temperature, stir the food, etc. SCRtool's specification editor allows us to declare `OvenMode` in a convenient table editor as shown in fig. 4.

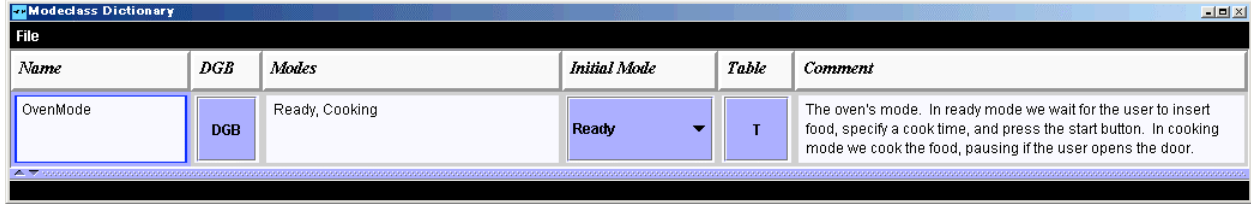


Fig. 4. SCRtool's mode class editor

Transitions between modes in SCR are defined in a *mode class table* as shown in fig. 5. Each row in the table specifies a transition from a source mode or modes to a destination mode in response to an *event*. Events in SCR are written in the form $@T(X)$ or $@F(X)$, where X is a predicate over the monitored variables. $@T(X)$ means “the moment X becomes true”, and $@F(X)$ means “the moment X becomes false”. Either of these may be followed by a guard condition in the form *WHEN* Y , where Y is a predicate over the monitored variables that must be true if the event is to cause a state transition. In fig. 5, the event expression we give for the transition from `Ready` to `Cooking` specifies that the transition will be made when the door is already closed, the timer has already been set to something other than zero, and the start switch is depressed.

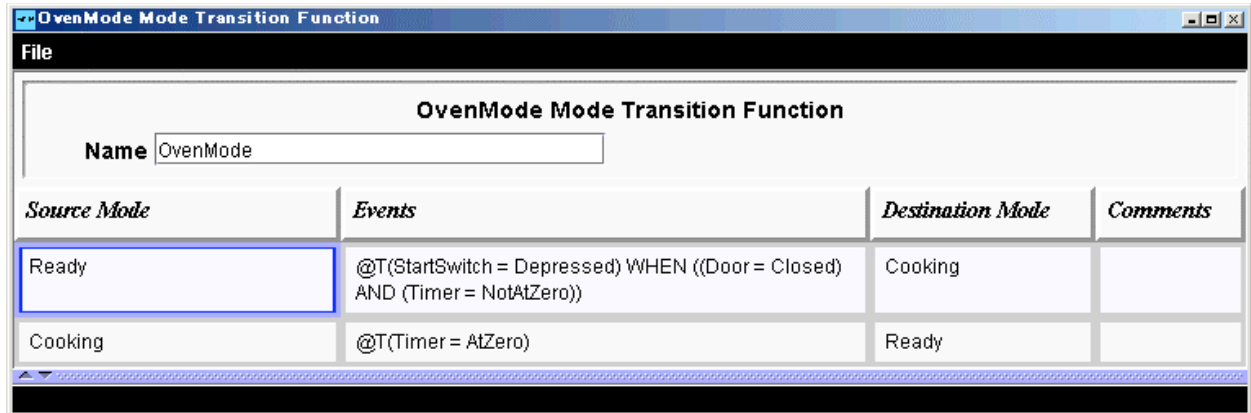


Fig. 5. SCRtool's mode transition table editor

C. Function Tables

With the monitored variables, controlled variables, and modes defined, we can define function tables that specify the values the controlled variables should be given by the controller. In an SCR specification, a *condition table*, *event table*, or *hybrid table* may govern a controlled variable. Fig. 6 shows the condition table defining how the output to the oven's microwave emitter is derived. Each row in a condition table represents a mode or modes, and each column represents a value or group of values. Each cell represents the condition under which, when in one of the modes represented by the cell's row, the governed variable should take on the value represented by the cell's column. In fig. 6, for example, the cell in the `Cooking` mode row and the emitter is `On` column contains the expression “`Door = Closed`”, meaning that when the `OvenMode` is `Cooking`, the microwave emitter should be on when the oven door is closed.

Columns need not necessarily represent constant values. When a condition table is used to define an integer or floating point variable, the value expression defining the column may be an arithmetic expression. A controller aiming to keep the level of fluid in a tank constant, for example, may use such a construct to set the fill rate proportional to the amount of fluid that needs to be added.

Emitter Condition Function
Defines a Controlled Variable

Name: Mode Class:

Modes	Conditions	Comments
Cooking	Door = Closed	Door = Open
Ready	FALSE	TRUE
Emitter =	On	Off

Fig. 6. SCRtool's condition table editor

Event tables are similar to condition tables, except that the cells contain the events upon which the variable should be set to the indicated value, rather than the conditions under which it takes the indicated value.

D. Assertions

Our microwave oven's cabinet is designed to contain harmful radiation while the oven is operating so as to protect the user. We expect users to attempt to open the door while cooking is in progress. In some cases this may be inadvertent, but we expect the user to periodically inspect the temperature of the food, stir liquids, turn solids over, and so on. In order to avoid exposing the user to microwave radiation, we must never allow the microwave emitter to be on while the oven door is open. We could accomplish this with hardware devices, such as an electromagnetic lock on the door or a hardware interlock that blocks electrical current to the microwave emitter while the door is open, but for this example we will make the controller responsible for the safety of the operator. It is critical that our controller logic satisfy the property that it never commands the emitter to be on while the door is open. The SCR notation allows us to write this property as an *assertion* as shown in fig. 7. Expressing this constraint as an assertion has two purposes: it formally records this design requirement, and it enables the use of formal verification tools to ensure that the specification guarantees that it will be satisfied.

Name	DGB	Source	Include?	Prove?	Proof Result	Expression	Comment
UserNotCooked	DGB	User	yes	yes	Proven	(Door = Closed) Or (Emitter = Off)	
LightOnWhenDoorOpen	DGB	User	yes	yes	Proven	(Door = Open) ==> (Light = On)	

Fig. 7. SCRtool's assertion editor

E. The Dependency Graph Browser

Understanding the components of a system as small as our microwave oven controller, and the dependencies between them, poses little difficulty. As specifications grow in size, however, understanding the relationships between specification components becomes more difficult [2]. To address this difficulty, SCRtool includes the dependency browser shown in fig. 8.

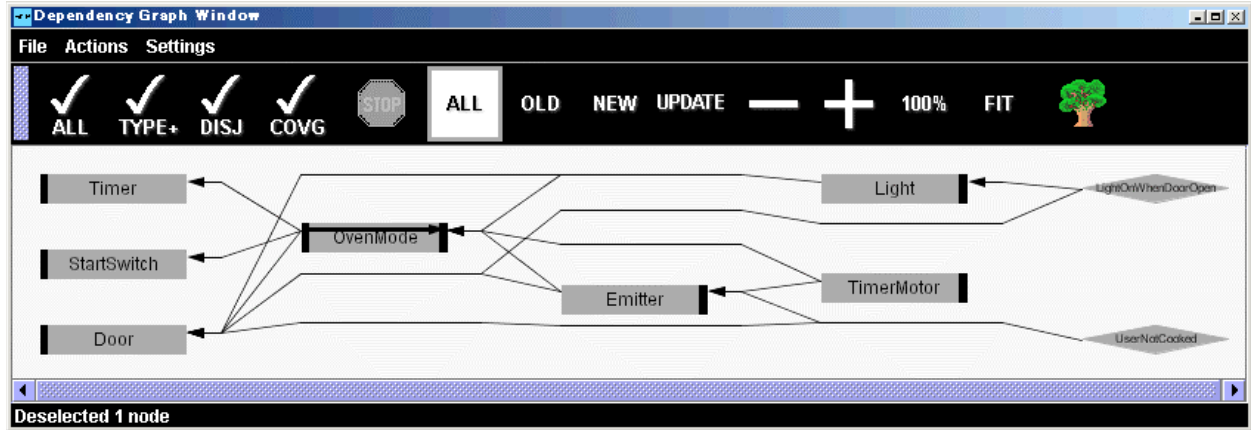


Fig. 8. SCRtool's dependency graph browser

In order to help the engineer cope with complex specifications, the dependency graph browser includes a feature for separating a selected sub-graph into its own specification file.

F. The main window

SCRtool's main window is organized as a tree view as shown in fig. 9. Expanding the leaves of the tree allows the specification engineer to browse the specification hierarchically. Double-clicking on portions of the specification causes the SCRtool to open dictionary windows for viewing, adding to, editing, or deleting parts of the specification.

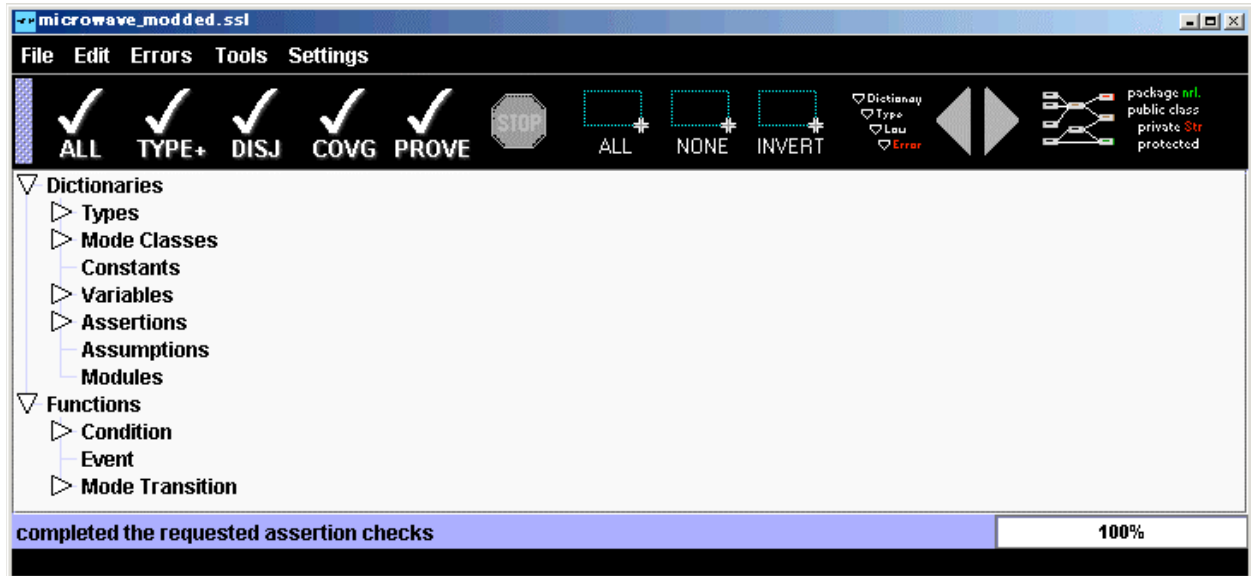


Fig. 9. The main SCRtool window

III. CHECKING A SPECIFICATION

Now that we have written a specification for our microwave oven controller, we would like to check it for specification errors. The first step is to run a suite of simple consistency checks. Running these checks is literally a push-button affair; we direct SCRtool to perform *type+*, *disjointness*, and *coverage* checks by clicking on the “check all” button in the toolbar of the main SCRtool window (shown in fig. 9).

A. Type+ Checking

Type+ checking will catch several kinds of specification errors. It includes simple syntax checking, which will help us find typos such as `@(Foo=Bar)` when `@T(Foo=Bar)` was intended. It also includes checking similar to a compiler’s semantic checking that will catch references to non-existent variables or constants, or attempts to equate to a variable a value that is not a member of that variable’s type. During type checking the tool set will automatically verify that the initial conditions for the specification’s variables are consistent with the function definitions. The tool also checks for circularities in variable definitions, which are illegal. Finally, the tool will verify that all modes in all mode classes are reachable from the initial modes. If an error is found, the tools will highlight the offending portion of the specification in red and describe the problem, as shown in fig. 10.

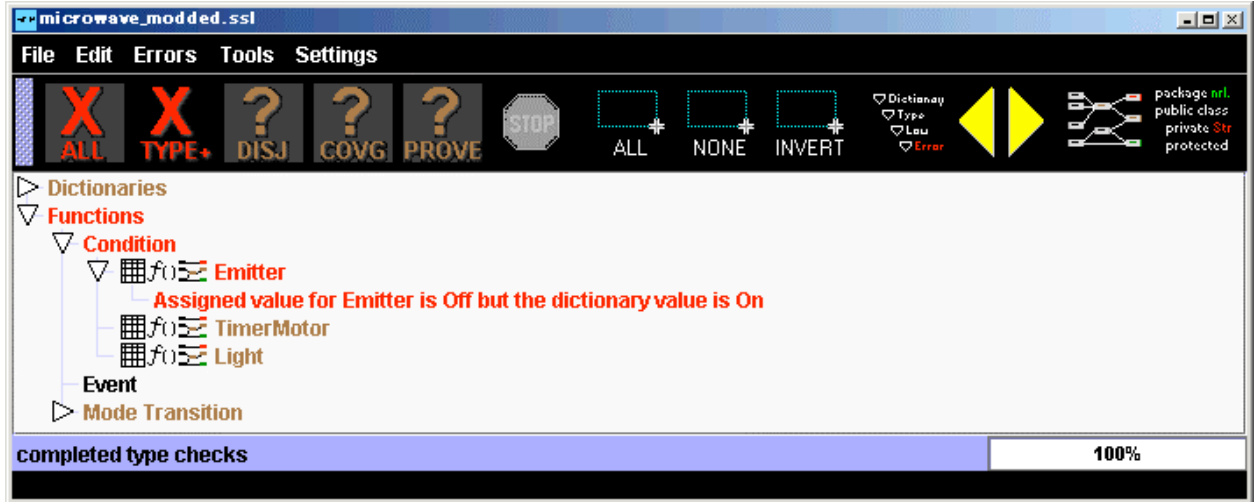


Fig. 10. SCRtool reporting a type checking error

B. Disjointness Checking

Disjointness checking verifies that the conditions in each row of each condition table and the events in each row of each event table or mode transition table are disjoint. If there were a combination of monitored variable values that caused multiple conditions or events in a row to be applicable, the variable controlled by the table would be specified to simultaneously be equal to at least two different values. With simple conditions, such as the one shown in fig. 6, it is easy to determine by inspection that the conditions are disjoint. With more complex conditions, however, this becomes more difficult. By offering a mechanical check for disjointness the SCRtool gives us increased confidence that our specification is not self-contradictory and reduces the need for expensive manual checking. As with type checking, consistency checking is fully automatic.

C. Coverage Checking

Coverage checking verifies that the conjunction of the conditions in each row of a condition table is true. If there were some combination of monitored variable values that caused no condition on a row to be true, the value of the variable being defined by the table would be undefined. While non-determinism

isn't necessarily erroneous, the SCRtool authors contend that deterministic specifications are far easier to analyze and debug than nondeterministic ones [3], and correspondingly require specifications created with their tool to be deterministic. Again, as with type and consistency checking, coverage checking is fully automatic.

D. Proving Assertions

Once we have verified that our specification is free of type, disjointness, and coverage errors, we can use SCRtool's proof mechanism in order to prove that our assertions hold. Here too the process is fully automatic by design: the tool's creators explicitly aimed to minimize both the effort and expertise needed to use the tool [2]. SCRtool's prover was able to prove both assertions shown in fig. 7 in a few seconds on a commodity Pentium 4 desktop machine running the Windows XP operating system. The prover does not always work this quickly; proving a different assertion on the same machine took several hours.

In some cases proving an assertion will require knowledge about relationships between monitored and controlled variables that are enforced by mechanisms in the environment, rather than by the software. In the case of our microwave oven, for example, if the timer motor is on then the timer will eventually reach zero. When it is necessary to document how our manipulation of the controlled variables will affect the values taken on by the monitored variables, we can create an *assumption*. In SCRtool, assumptions are created and edited in much the same manner as assertions. Assumptions are treated as facts by the prover, which takes them into consideration when attempting to prove assertions.

In other cases, the prover will need help to prove the assertions in a specification even though all of the information needed to prove them is present. To help it out, the tool set includes a facility for automatically generating and proving invariants that the prover can take into account when attempting to prove other properties. Once again this is fully automatic; the user selects a menu item, chooses the invariant generation strategy and the parts of the specification to examine, and clicks a button to activate the generator.

In some cases, adding assumptions and generating invariants may not be enough to allow the tool to prove a property that is, in fact, true. A sample specification that we were given by the authors of the tool, for example, contains an assertion that is true but cannot be proved by the tool. In such cases, specification developers will either need to prove the property manually or use a different tool. The generated invariants will likely be of help in doing this.

IV. SIMULATION

In order to help engineers to validate their specifications or explore the meaning of an error discovered during verification, SCRtool includes a facility for simulating specifications. The simulator, shown in fig. 11, includes windows that show the monitored and controlled variables, terms, and mode classes. An engineer drives the simulation by entering new values for the monitored variables. As each change is made, the simulator computes the response commanded by the specification and shows any resulting changes to controlled variables. Fig. 11 shows a simulation of a user cooking food that is already in the microwave: the timer dial is first turned to the desired cook time, resulting in the timer reporting that it is *NotAtZero*; the user presses the start button, causing it to become *Depressed* and then *Released*; and finally the timer counts down to zero, causing it to report being *AtZero*, thus ending the cooking operation. If any of the monitored variable changes input by the engineer contradicted one of the assumptions in the model, the simulator would have displayed a warning of the violation in red in the log window. Likewise, if the simulated controller had violated an assertion, the simulator would have detected and reported the violation.

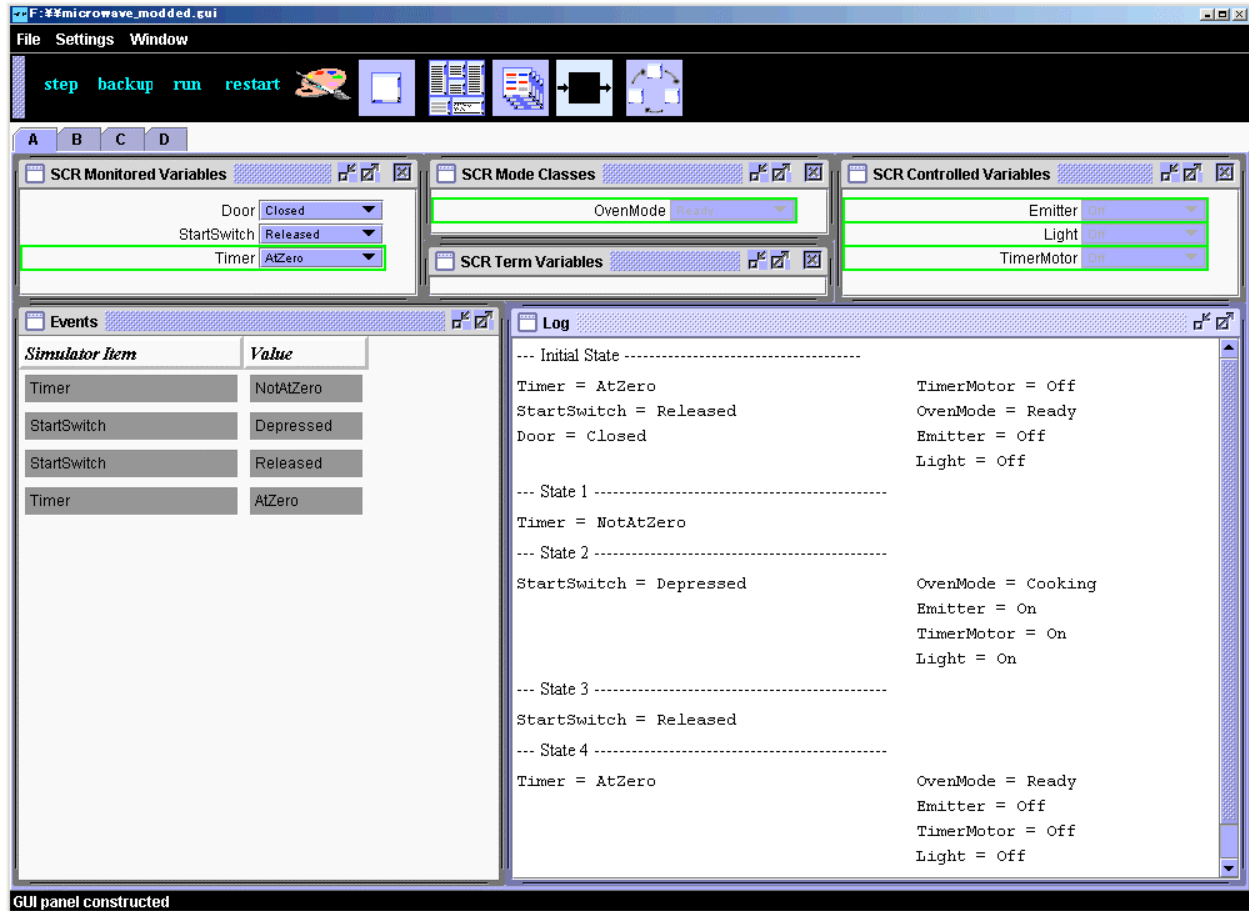


Fig. 11. SCRtool's simulator

V. DISCUSSION

SCRtool and the underlying tabular SCR specification notation have been used to develop software for a variety of real world systems, including the flight program of the A-7 aircraft, the shutdown system for the Darlington nuclear power plant, and the flight program for the Lockheed C-130J [2]. These systems are examples of the kinds of software programs SCRtool and the SCR specification notation are intended for: embedded control systems. While the tools offer a clear notation, a precise formal semantics [3], the ability to prove properties of interest, and the ability to simulate the system for validation and debugging purposes, neither the tool nor the language is well-suited to all kinds of software.

A. Complex user interfaces

The SCR notation is not particularly convenient for specifying complex user interfaces. Simple manual switch, dial, indicator lamp, and gauge components, which formed much of the A-7's user interface, present no difficulty, as these components can be easily mapped to monitored and controlled variables of the supported types. More complicated controls present more difficulty. The A-7, for example, includes a 13-character 7-segment LED display that serves to display several kinds of information to the pilot. The team that wrote the A-7 specification in SCR notation opted to model this display as 48 separate controlled variables, each representing the value that would be displayed on the panel if the panel was in the mode that caused it to be displayed [5].

This limitation does not seem to be specific to the SCR notation or the SCRtool. Model-based development tools such as Simulink and SCADE also have this limitation, as user interfaces must be

represented in these tools as input and output signals that are analogous to SCR's monitored and controlled variables.

B. Complex system state

The prototype tool we examined provides no support for string, list, array, stack, queue, map, or other non-scalar data types. Because controlled variables and modes are the only ways to represent the controller's state in SCR notation, the absence of these types means that there is no practical way to model collections of things in the controller's state. While many environmental values of interest to an embedded controller can be modeled using the simple Boolean, enumerated, integer, and floating-point types provided, the lack of a mechanism for modeling a state that includes a collection of any kind will make it impractical to use this version of the SCRtool to describe some kinds of software systems. A simple program to track a library of books to be lent out, for example, could not be easily specified.

Again, this limitation seems to also apply to tools such as Simulink and SCADE. These tools also lack the ability to represent collections of values. However, while future versions of SCRtool may include support for collections, the nature of Simulink and SCADE models seems to preclude them. Simulink and SCADE model the paths along which information flows from operation to operation as if these paths were wires and the operations electronic components; it is difficult to imagine how a collection of values would be represented within the framework of that metaphor.

C. Target audience

SCRtool and the SCR specification language are intended for use by requirements analysts [4], and SCRtool was deliberately developed to require as little skill in discrete mathematics and theorem proving as possible. In the view of the tool's creators, requirements analysts should create the specification on behalf of their customers by formalizing and describing the customer's expressed needs [4].

This view is quite different from the one seemingly held by proponents of model-based development tools such as Simulink and SCADE. These tools are intended for use by control engineers, who specify the system they want in a familiar graphical notation and then use the tools to automatically generate finished source code. By modeling the requirements of a computer-based system in a manner that is easy for control engineers to understand, these tools eliminate the need for a human software requirements analyst to translate a description of the needed functionality into a more software-centric language.

D. Code generation

Unlike both Simulink and SCADE, the current-generation SCRtool lacks the capacity to automatically generate finished source code. While it does have a source code generator, that generator produces Java code that is suitable for use in simulation, but almost certainly unsuitable as production code implementing the specified system. While Simulink and SCADE are touted for eliminating the need to manually write source code, a development team using SCRtool would need to write source code implementing the specification they develop.

VI. CONCLUSION

SCRtool offers tools to create, edit, analyze, and simulate specifications written in the formal SCR specification notation. The system model embodied in the language and tools describes software that is embedded in an environment that it senses through monitored variables and manipulates via controlled variables. The tool set includes a fully-automatic prover that can be used to demonstrate that assertions hold over the specification and a simulator that can be used both to explore specification errors and to allow the specification developers to assure themselves that the specification describes the desired controller.

While the tabular SCR specification language is clear and concise, the current-generation SCRtool is not suited to all types of software development. The tool's type system for monitored and controlled

variables in particular is well suited to the target embedded control system market, but may be ill-suited for applications that will need to maintain complex data structures.

Unlike such model-based development tools as Simulink and SCADE, SCRtool is designed for use by software requirements analysts. Where Simulink and SCADE are intended for use by control engineers, the development model surrounding SCRtool treats these system-level engineers as customers and requires the requirements analyst to translate the requirements imposed on the software by the system into the SCR tabular specification notation. Where Simulink and SCADE offer the developer the ability to automatically generate source code that can be used as the implementation of the system, human programmers are needed to generate production-quality source code implementing the functionality captured in a specification generated using the SCRtool.

ACKNOWLEDGMENT

I thank Ralph Jeffords for his help in acquiring and understanding SCRtool and its proof mechanism.

REFERENCES

- [1] C. Heitmeyer. "Managing Complexity in Software Development with Formally Based Tools." *Electronic Notes in Theoretical Computer Science*, vol. 108, 2004.
- [2] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. "Tools for constructing requirements specifications: The SCR toolset at the age of ten." *International Journal of Computer Science & Engineering*, vol. 20, no. 1, January 2005.
- [3] C. Heitmeyer, R. Jeffords, and B. Labaw. "Automated Consistency Checking of Requirements Specifications." *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, July 1996.
- [4] C. Heitmeyer. "Point/Counterpoint." *IEEE Software*, vol. 22, no. 1, Jan.-Feb. 2005.
- [5] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Application." *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, January 1980.