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
Vibha Prasad

An Introduction to the SPIN Model Checker 20
Xiang Yin

KRONOS: A Verification Tool for Real-Time Systems 30
Na Zhang

Model-Based Development Tools

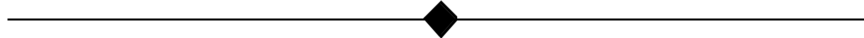
Model-Based Development Using Simulink 40
Ben Taitelbaum

Safety Critical Application Development Environment 45
Kendra N. Schmid

***Perfect Developer* Tool Suite** 53
Michael Spiegel

SCRTool and the SCR Specification Language 58
Patrick John Graydon

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.

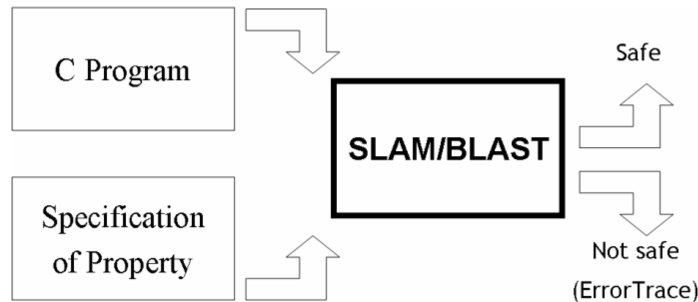


Fig. 1. Overview of SLAM and BLAST (from slides of [3])

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.

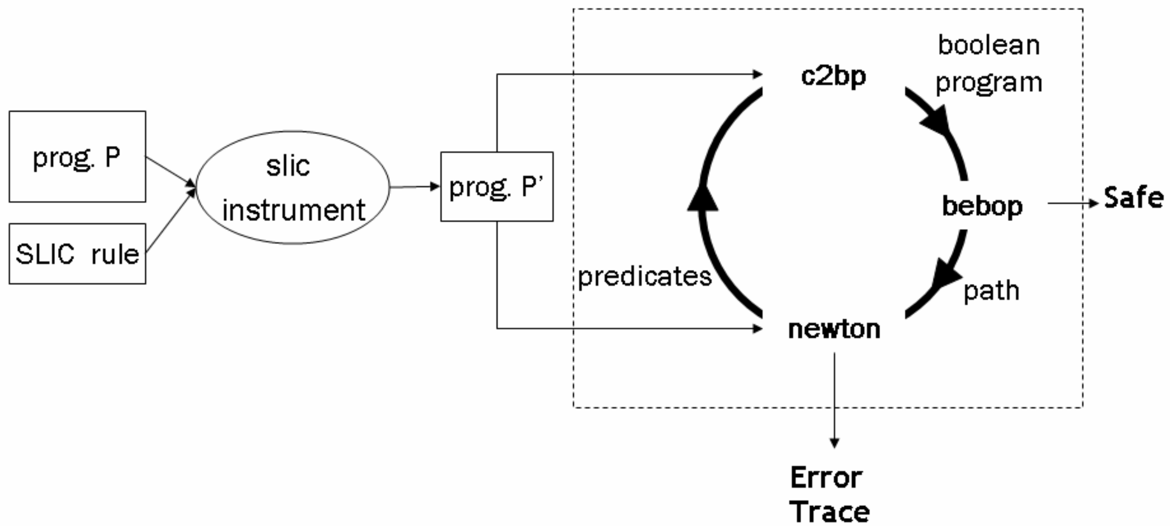


Fig. 2. Working of SLAM ([5])

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.

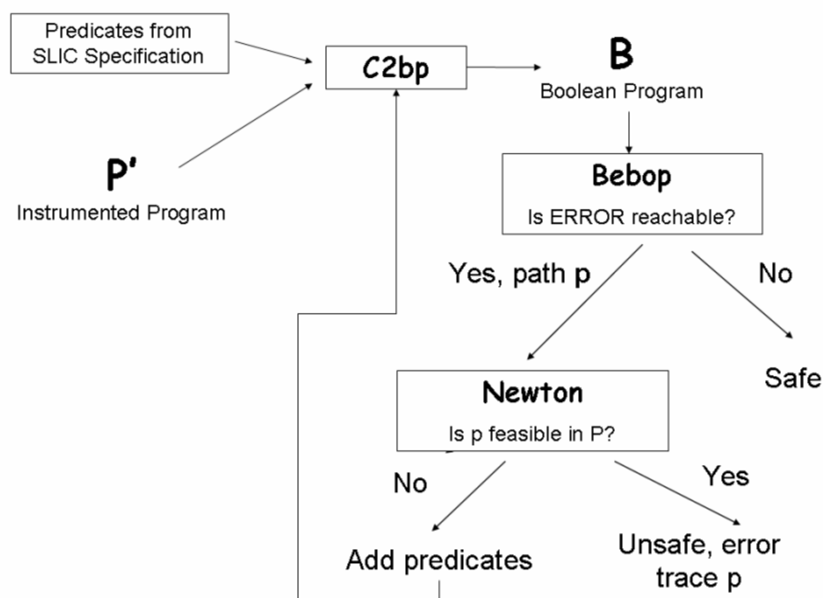


Fig. 3. Components of SLAM (from slides of [3])

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.

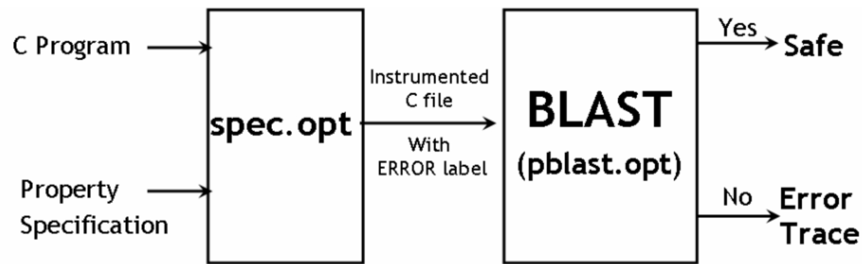


Fig. 4. Working of BLAST ([15])

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 checksneccessary 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 vampyre as theorem provers.

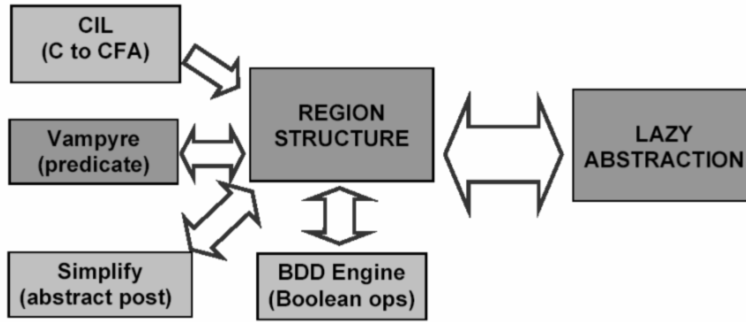


Fig. 6. The BLAST architecture (from slides of [16])

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.

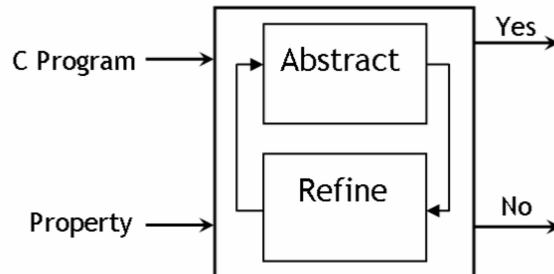


Fig. 7. Working of BLAST (from [15])

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.

```
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 :-(
```

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 - SecureCRT

File Edit View Options Transfer Script Tools Help

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 extend 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-Verilag 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.

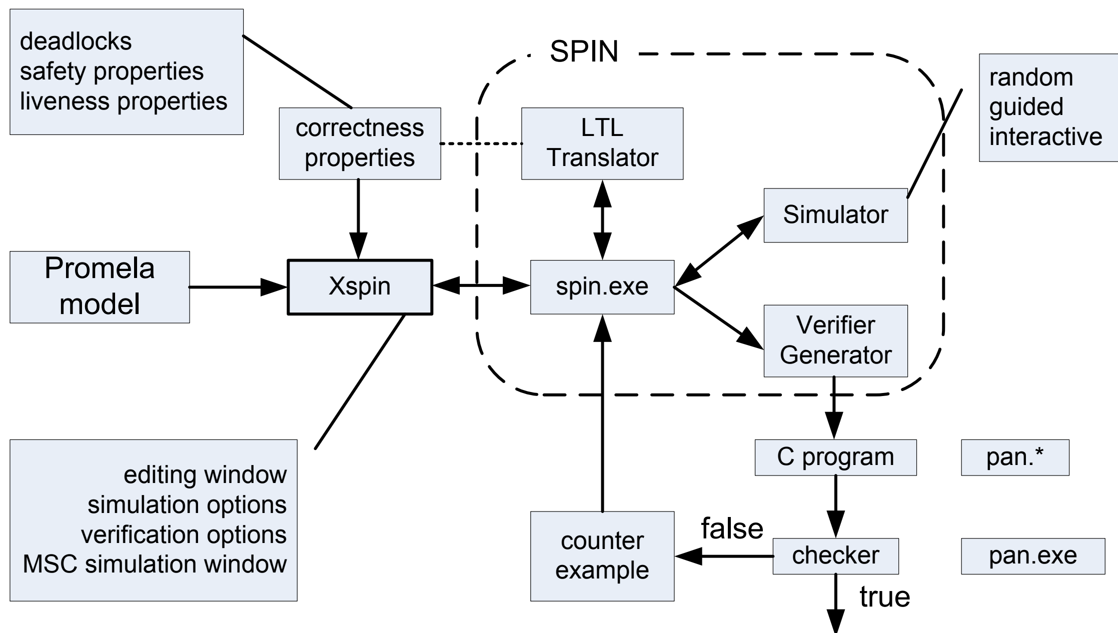


Fig. 1. SPIN architecture [3][4]

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 $\Box(req \rightarrow \Diamond ack)$ 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 extracts 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 bread-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.

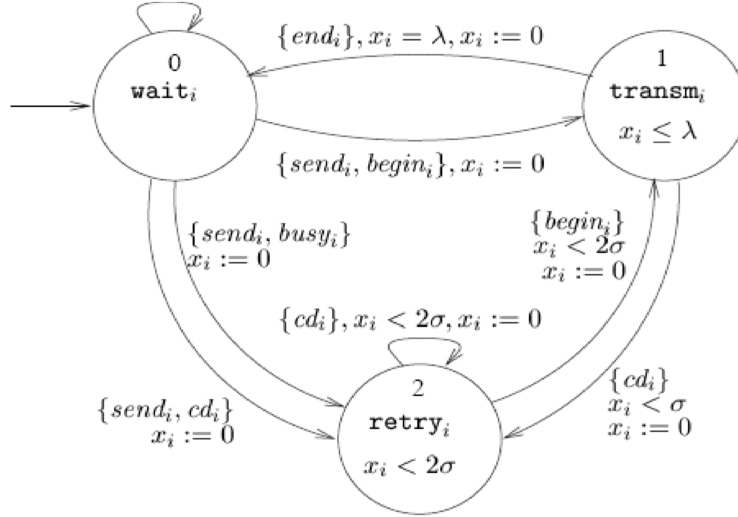


Figure 1: timed automaton for sender*i* in CDMA/CD protocol

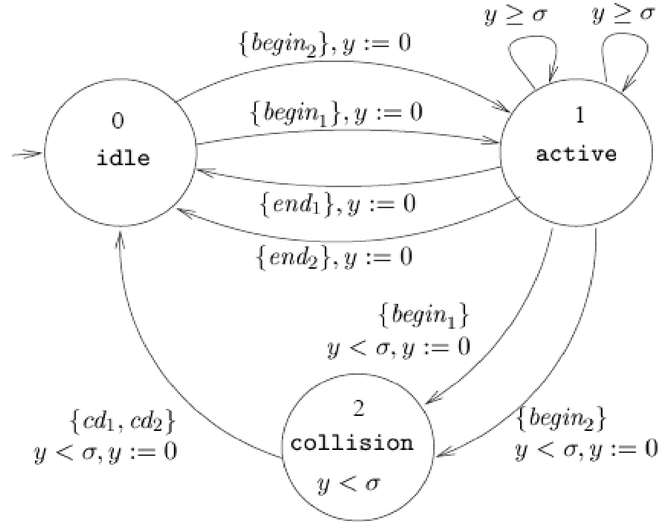


Figure 2: timed automaton for bus in CSMA/CD protocol

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 reach-ability 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:
  $init \Rightarrow \forall \Box\, safe$
  where the symbol "$\forall \Box$" 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:

$$init \Rightarrow \forall \Box \exists \Diamond_{=1} \; true$$

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.