Verification of Software via Integration of Design and Implementation

Andrew S. Miner Samik Basu

Department of Computer Science,
Iowa State University, Ames, IA 50010
Email: {asminer,sbasu}@cs.iastate.edu

Abstract

Model checking is usually applied at the design phase to verify that preliminary high–level design specifications conform to their requirements. Source code analysis, on the other hand, is used to check for correctness of implementation once it is realized from the design specifications. However, the current practice of validating a design and its implementation in isolation makes it necessary to employ rigorous testing analysis to empirically ensure that the implementation satisfies the design specification. This article describes a formal framework that allows design models to contain embedded partial implementations as components; these models are then formally analyzed to ensure that global requirements are satisfied. This framework can be utilized to incrementally develop and ensure correctness of the design and the corresponding implementation. Realization of this framework requires consolidation and expansion of traditional formal verification techniques by integration of model checking, program analysis and constraint solving.

1 Introduction

Formal verification and validation methods provide a high degree of confidence of the correctness and reliability of software systems. At the design phase of the software life cycle, automated techniques, e.g. model checking [14, 37], can be applied to verify that the preliminary high–level design specification conforms to the requirements. The desired requirements and the design specification are represented, respectively, in temporal logic and in high–level modeling languages; their common aspect is that the semantics of both are expressible as finite state automata (regular languages). Finally, model checking, essentially defining a satisfaction relation, involves automata intersection and verifying that it accepts the empty language.

Once a correct design is obtained, the next phase is to develop the corresponding implementation. However, due to the presence of rich classes of data (e.g. infinite domain integers) and control structures (e.g. recursion), model checking source code behavior, in general, is undecidable [39]. Over the last decade, model checking techniques have been fine tuned [15, 17, 29, 33] to reason about infinite state systems. Recent breakthroughs in model checking source code [1, 20, 22] rely either on smart abstraction techniques or on application of constraint solvers to handle infinite domain data present in the implementation.

Formal verification of design is a useful step as it increases confidence that the implementation will perform correctly. Similarly, verification of the implementation (if possible) increases confidence. However, these separate verifications do not ensure that the implementation is a correct representation of the design as the design requirements may not be imposed directly on implementation and vice versa. This inherent gap between ensuring correctness of design and ensuring correctness of implementation is further complicated in practice since designs tend to change over time, even

1-4244-0054-6/06/$20.00 ©2006 IEEE

after implementation of the design has begun. Furthermore, this type of separate verification does not allow for verification of a component implementation against a global high–level design model, as might be desired before the entire system has been implemented. As software systems become increasingly multiple–process and multiple–thread, designs and implementations become more complex and the problem only worsens.

In this paper, we present a roadmap where high–level designs with multiple components are verified against implementations of some (not all) of its components. We proceed with the discussion of the prior research as it relates to current objective (Section 2). Section 3 presents the overview of our approach. The central theme is to verify local properties of implementation of a component and identify a set of conditions that must be imposed on other components (i.e., the environment) for its correct functionality. These conditions are referred to as the *obligation of the environment*. The global design model is then verified against the requirements and its obligations imposed by the component implementations. Sections 4 and 5 discuss the current work and future avenues of research, respectively, towards this objective.

2 Background and Related Work

2.1 Model Checking a High–level Model

Model checking [14] is the process of verifying that a finite state machine satisfies a desired property, which is usually specified in terms of some temporal logic, e.g. computational tree logic (CTL) [5] or linear temporal logic (LTL). The finite state machines that describe the dynamics of system behavior are not specified by hand; instead, a high–level modeling formalism is used, and the necessary structures to perform the model checking are constructed automatically as needed. In particular, the underlying finite state machine can be constructed from the high–level model automatically if necessary. Theoretically, any high–level description that has a formal set of rules for constructing the underlying finite state machine will work, including source code (with restrictions). To aid in discussion, in Section 2.1.1 we briefly review one such type of high–level formalism, namely *Petri nets*, which are one of the well–accepted modeling formalisms that satisfy the above requirement. High–level models often suffer from the *state explosion* problem: while a realistic system may be described compactly using a high–level formalism, its underlying finite state machine can be extremely large. This leads to practical difficulties for model checking. Researchers have attacked this problem in various ways; some techniques that have received widespread acceptance are described in Section 2.1.2.

2.1.1 High–level Models of System Design

Petri nets. Petri nets [36] are a graphical design tool with simple yet powerful underlying rules; these properties have contributed greatly to their success. Formally, a Petri net is a directed, bipartite graph consisting of *place* nodes and *transition* nodes. Usually, places are drawn as circles and transitions are drawn as bars. Each place may contain a non-negative number of *tokens*. The state of the Petri net, known as its *marking*, is given by the numbers of tokens in each place. Place p is called an input place to a transition t if there is an arc from p to t, and is an output place if there is an arc from t to p. Changes of state are dictated by the transitions: a transition is *enabled* if all of its input places contain tokens, and an enabled transition may *fire* by removing tokens from its input places and adding tokens to its output places. Weights can be used on the arcs so that transitions can add and remove more than one token from a place. Since Petri net tokens are indistinguishable, abstraction is usually required when dealing with data (e.g., for software). Alternatively, colored Petri nets can be used, which allow tokens to be distinguished by "colors" that are commonly used to represent data [21, 30]. Theoretically, any colored Petri net can be converted into a (usually much larger) uncolored Petri net, a process known as *unfolding*.

2.1.2 Symbolic Model Checking

Within the model checking literature, techniques based upon the use of decision diagrams have come to be known as *symbolic* or *implicit* methods. To use these

methods, states of the finite state machine are assumed to be composed of K state variables, with $K > 1$. A set of states can be represented using a multi–valued decision diagram (MDD) [31], a directed acyclic graph containing terminal nodes 0 and 1, and non–terminal nodes which are labeled with one of the state variables x_k and contain pointers to other nodes for each possible value of variable x_k. Binary decision diagrams (BDDs) [6] correspond to the special case where each state variable can assume only values 0 and 1. Rules for construction and manipulation of BDDs and MDDs can be found in [6] and [31]. While MDDs require an exponential number of nodes in the worst case, they have been used successfully in practice to compactly represent large sets of states.

To use symbolic methods for model checking, it is also necessary to use a compact representation for the finite state machine. Traditionally this has been done using MDDs with $2K$ state variables to represent the transition relation of the finite state machine: if $(x_K, \ldots, x_1, x'_K, \ldots, x'_1)$ is contained in the MDD representation of the transition relation, then there is a transition in the finite state machine from state $(x_K, \ldots, x_1)$ to state $(x'_K, \ldots, x'_1)$. The MDD for the transition relation is constructed directly from the high–level model, thus avoiding the need to explicitly represent the finite state machine. More sophisticated techniques are used in practice (e.g., partitioned transition relations [7]) but are based on similar concepts.

Symbolic methods have been used quite successfully for verification of synchronous and asynchronous circuits (e.g., [7, 8, 38]), where BDDs are a natural fit due to the boolean state variables. For systems whose state variables are not necessarily boolean, such as manufacturing systems, software systems, and communications systems, BDDs can still be used by describing each state variable as several boolean variables (e.g., [25]), or MDDs can be used [11, 35].

2.2 Model Checking Implementations

2.2.1 Sequential Program Verification

Given a set of correctness assertions, model checking the source code of a sequential program is typically based on the three step paradigm of *abstraction*, *verification* and *refinement* [13, 19, 23]. The first step involves building an over–approximate abstraction P' of the program P. Abstraction is necessary to handle operations on infinite–domain variables present in the system. Thus, source code verification is concerned with handling programs with infinite data behavior, but control is still assumed to be finite. In the second step, the abstract model P' is verified against the desired property. In the event that the property is violated, a counter–example is generated. A counter–example in P', however, may not be a valid sequence of steps in the concrete system P owing to the abstractions performed on the variables. The steps used to obtain the counter–example will produce a set of constraints, which are fed into a constraint solver to determine if the constraints can be simultaneously satisfied or not. If the counter–example is indeed infeasible in the concrete system, P' is refined (the third step) and the three steps are repeated until a feasible counter–example is obtained or the property is satisfied.

2.2.2 Multi–threaded Program Verification

Multi–threading results in concurrent execution of a program segment by two or more threads. The different possible thread interleavings add a new dimension to the complexity of program verification, as the property must be checked for all possible interleavings. Infinite domain data is typically reduced to a finite domain, and control flow graphs are employed to represent behavior of each thread in a multi–threading environment. A finite state model is generated from a multi–threaded program by applying predicate abstraction (Magic:[10]) and/or program slicing (Bandera:[22]). While [10] uses weak simulation to decide the conformity of the model to its property, [22] feeds the model to well–established finite state model checkers such as Spin [26] or SMV [9].

2.3 Existing Model Checking Tools

The tools available for model checking focus primarily on verification of high–level models or on analysis of source code. In other words, the tools verify the

design model or source code but not a combination of both. For example, the tools Spin [26] and SMV [9] both take as input a high–level model, described using the Promela and SMV input language respectively, and verifies the model against desired properties written in temporal logic. On the other hand, source code verifiers like Slam [1] and Blast [24] perform automatic abstraction of programs to generate finite state models and uses constraint solvers to check for satisfiability of assertional properties. Bandera [22] verifies Java source code by translating the source code to finite state models that can be used as inputs to Spin and SMV. The central theme in all these tools is that either design model or the implementation is available in its entirety. We are not aware of any model checking tool that can verify implementations against a high–level design model, in particular when implementations of some, but not necessarily all, of the components are realized. Such a tool is necessary to bridge the gap between ensuring correctness of design and ensuring correctness of implementation in a realistic setting (i.e., where design and implementation change over time).

3 Model Checking Hybrid Models

We start with a high–level design model that is divided into components. Specifically, we consider a high–level design model M, consisting of n component models denoted by $M_1, M_2, \ldots, M_n$, and a desired property φ. Using techniques discussed in Section 2.1, a high–level model checker can be used to check the high–level model M against the property φ, which will result either in proof that M conforms to the desired property φ (typically written as $M \models \varphi$), or a generated *counter–example* witnessing the violation of φ.

As an example, consider a multi–process system containing writer and reader processes with a shared resource. Each writer process will perform some computation, and then modify the shared resource; reader processes will access the shared resource read–only (i.e., without modification) and perform some computation. For the system to function correctly, at most one writer process should modify the resource at a time, and no reader process should access the resource while it is being modified. A Petri net model for this system is shown in Figure 1(b), with components for a single writer and reader process shown in dashed rectangles. Multiple reader or writer processes can be represented by duplicating the dashed rectangles and using extra inhibitor arcs.

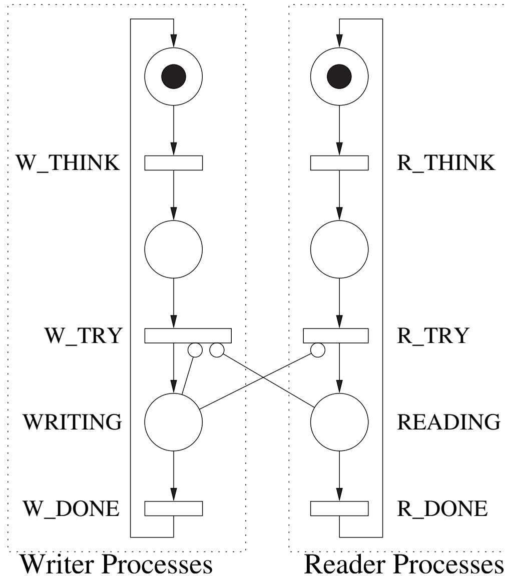
Now suppose that some, but not all, of the components have been implemented, and we wish to verify local properties of the implemented components and validate the result against the global model. We refer to the implementation of component i as I_i. Note that implementation I_i can have certain local properties, denoted by ψ_i, that it must satisfy for its correct functionality. Source code verification techniques (abstraction–refinement and/or slicing), described in Section 2.2, can then be applied to check whether $I_i \models \psi_i$. However, it is likely that the correct behavior of component i depends on its environment (the remaining components). As such, we must identify the conditions or constraints imposed on the environment required to satisfy the goal of $I_i \models \psi_i$.

Example implementations for the writer and reader processes are shown in Figure 1(a) and Figure 1(c). This implementation uses shared integer variables `writers` and `readers`, and a semaphore `lock` used to synchronize modifications to the shared variables. Note that variable `writers` counts the number of writer processes modifying the resource, and variable `readers` counts the number of reader processes accessing the resource. The code is designed to allow for multiple writer or reader processes, each executing its own copy of the code shown in the figure.

The long–term goal of this work is to develop model checking techniques to verify properties against $(I_i, M - M_i)$, where $M - M_i$ is the high–level model with component i removed, and $(I_i, M - M_i)$ is a hybrid model where the implementation of component i has been plugged into the remaining high–level model. For our running example, if I_i is the writer source code then $M - M_i$ is the high–level reader (Petri net) model. Note that, in general, $M - M_i$ is the environment for component i. We then can check $(I_i, M - M_i) \models \psi_i$ or $(I_i, M - M_i) \models \varphi$ to determine the requirements of the environment (if any) for satisfying local property ψ_i or global property φ. Section 4 describes our progress to-

```
1:while (1) {
2:  THINK();
3:  while (1) {
4:     wait(lock);
5:     if (readers > 0)
6:     then signal(lock);
7:     else break;
8:  }
9:  writers++;
10: assert(writers==1);
11: assert(readers==0);
12: WRITE();
13: writers--;
14: signal(lock);
15:}
```

(a) Writer code



(b) Design model

```
1:while (1) {
2:  THINK();
3:  wait(lock);
4:  readers++;
5:  signal(lock);
6:  assert(writers==0);
7:  READ();
8:  wait(lock);
9:  readers--;
10: signal(lock);
11:}
```

(c) Reader code

Figure 1. A component–based example: writers and readers

wards this goal, while Section 5 outlines our ongoing efforts to achieve this goal.

4 Current Work

4.1 Counter–example Analysis

In the domain of program verification, we proposed efficient solution mechanisms for reasoning about recursion [2] and for analyzing multiple counter–examples by ascertaining dependencies of variables at various program points [3, 4, 32]. Counter–example analysis is performed by identifying a slice of counter–example, execution of which and nothing else causes the erroneous behavior. The sequence of variable operations present in this slice is sufficient to prove or disprove the feasibility of the corresponding counter–example. Dependency analysis of variables is a program analysis technique widely deployed in the field of software engineering, for debugging and testing programs [18, 27, 40]. The essence of the technique is to perform *dynamic program slicing* [16, 28] to identify (a) constraints on variable valuations and (b) semantically dependent, possibly noncontiguous program segments (for debugging potential errors in programs).

Such a counter–example slice is referred to as a *focus statement sequence* (FSS). A statement in a counter–example is classified as a focus statement (a member of the FSS) if and only if it directly or indirectly affects the violation of an assertion. An important aspect of this work is that it can automatically identify the constraints over input variables (the variables *used* before before being *defined*) necessary for the feasibility of a counter–example. Negation of such constraints, referred to as *assumptions* in [4], leads to removal of the corresponding counter–example. In short, the pre-conditions over input variables required for correct functionality of the program can be identified by complementing the assumptions (obligation of the environment of the program).

The above projects form the starting point for generating constraints imposed by implementations on their environment. Furthermore, counter–example analysis for multi–threaded/multi–process implementations will follow similar techniques as described in [4].

4.2 MDD Extensions

To cope with potentially enormous state spaces during model checking of a hybrid model, a symbolic hy-

brid model checker will be developed. This tool will utilize several well–established techniques for model checking with MDDs, and some recent advancements in state space generation (e.g., [11, 34]). However, for a hybrid model, it is anticipated that additional information will be necessary for states or for edges in the finite state machine, in particular constraints that arise during counter–example analysis. This information will need to be incorporated into the MDD structure.

We are currently developing an extended MDD library to handle both traditional MDD structures and extensions of the form necessary to incorporate additional data such as constraints. Such data are typically stored either as values on MDD edges (e.g., [12]) or in additional terminal nodes (e.g., [25]). We are investigating the theoretical implications of each of these. A general–purpose MDD library that includes both edge–value and terminal–value capabilities will ultimately be exploited by our symbolic hybrid model checker, and will be necessary to evaluate large–scale systems.

5 Ongoing Work

Identification of interaction points. Given a high–level design model, identification of its interaction points can be done simply by examining the component models and their connections. For the running example Petri net in Figure 1(b), interaction points are determined by looking at arcs that cross the dashed rectangles. Interaction points for low–level implementation correspond to accessing of shared variables and semaphores, and may also include execution boundaries (e.g., the entry and exit points of a function), depending on the component interactions.

Model to implementation interface. Before an implementation can be plugged into its corresponding component in the high–level model, it is necessary to convert the implementation interaction points into the corresponding model interaction points. In particular, points in implementation that access semaphores or other shared variables must be synchronized with the corresponding features in the high–level model. Interactive computer tools will be designed to generate a mapping between the interaction points of a component's implementation the corresponding interaction points of the high–level component model.

Translation of properties. Once we have obtained the mapping between the high–level component's interaction points and the corresponding points in implementation, the next step is to identify the properties of design and incorporate them in the corresponding implementations. In other words, given a property (expressed in temporal logic) in terms of the high–level model, the property must be translated to include the implementation details using the mapping. For example in Figure 1(b), a desired property at design–level demands that write operations are performed in mutually–exclusive fashion and reading is allowed only when there are no writers.

$$\mathrm{AG}(\ (\texttt{\#tokens at WRITING} \leq 1) \wedge (\texttt{\#tokens at WRITING} > 0 \Rightarrow \texttt{\#tokens at READING} == 0)\ ).$$

The corresponding assertional requirements imposed on the implementations are preset at Lines 10 and 11 in Figure 1(a) and Line 6 in Figure 1(b).

Model Checking a Hybrid Model As suggested earlier, the integration of a high–level design model and a low–level implementation model produces a *hybrid* model. This is done in a similar fashion to sequential program verification; however, shared variables require special handling while slicing counter–examples, since a shared variable accessed by the implementation may have been modified by another component. As such, the obtained FSS is conditioned on the actions of the other components in a *constraint graph.* A constraint graph for the writer process implementation is shown in Figure 2; the assertion in Line 11 is violated if the double–circled state in the graph is reached. From the graph, it is clear that this can occur only if (a) shared variable `readers` is initially negative, or (b) semaphore `lock` is not initialized correctly, or (c) shared variable `readers` is modified without obtaining the lock.

Dealing with state explosion As discussed in Section 4.2, we will utilize symbolic methods to handle large systems. Constraint data can be incorporated

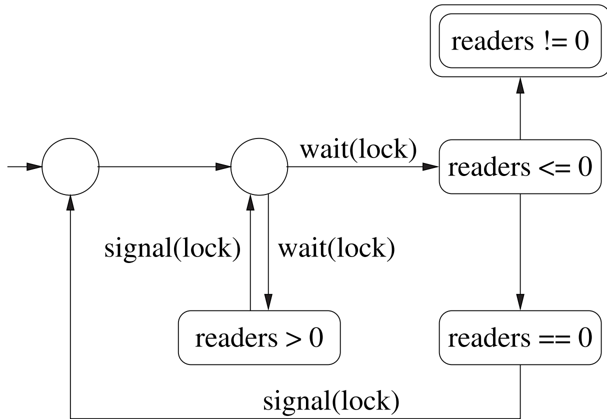


Figure 2. The constraint graph for the writer source code in Figure 1(a)

into an MDD using edge–value data and/or terminal node data, allowing for symbolic representations of large constraint graphs.

Counter–example analysis The primary objective of verification is to identify potential flaws in the design or implementation and use appropriate corrective measures. As such, analysis of counter–example forms an integral part of model checking. As alluded in Section 4.1, counter–examples in sequential programs are analyzed using slicing so that the user can zoom in specific program segments that are likely to be cause of the error. Similar techniques will be developed in the setting of multi–threaded programs where additional information regarding the interleaving of threads will be extracted to provide error–cause information.

In the domain of high–level design models, need for smart counter–example analyzers are also required to reduce the burden of understanding and decoding counter–example cause. We are exploring techniques for analysis of counter–examples for CTL formulas based on identifying strongly connected components present in the system design.

References

[1] T. Ball and S. K. Rajamani. Slam, 2003. `http://research.microsoft.com/slam`.

[2] S. Basu, R. L. Pokorny, K. N. Kumar, and C. R. Ramakrishnan. Resource constrained model checking for push-down systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 2002.

[3] S. Basu, D. Saha, Y.-J. Lin, and S. A. Smolka. Generation of all counter-examples for push-down systems. In *Formal Techniques for Networked and Distributed Systems (FORTE)*. Springer-Verlag, 2003.

[4] S. Basu, D. Saha, and S. A. Smolka. Counter-example analysis for Cimple debugging. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, 2004.

[5] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, C-35(8):677–691, Aug. 1986.

[7] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *Int. Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, Aug. 1991. IFIP Transactions, North-Holland.

[8] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, Apr. 1994.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science (LICS)*, 1990.

[10] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *Special Issue of Transactions on Software Engineering (TSE)*, 2004.

[11] G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: an efficient iteration strategy for symbolic state space generation. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.

[12] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer-Verlag.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstract refinement. In *Computer Aided Verification (CAV)*, 2000.

[14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[15] G. Delzanno and A. Podelski. Model checking in CLP. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

[16] M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Partial Evaluation and Semantic-Based Program Manipulation*, 1999.

[17] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Computer-Aided Verification (CAV)*, pages 324–336. 2001.

[18] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

[19] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[20] P. Godefroid. Model checking for programming languages using verisoft. In *Principles of Programming Languages (POPL)*, 1997.

[21] S. Gordon and J. Billington. Analysing the WAP class 2 wireless transaction protocol using coloured Petri nets. In *Application and Theory of Petri Nets 2000*, LNCS 1825, pages 207–226, Aarhus, Denmark, June 2000. Springer-Verlag.

[22] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to model-check properties of concurrent Java software. In *CONCUR 2001*, LNCS 2154, pages 39–58. Springer-Verlag, 2001.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, 2002.

[24] T. A. Henzinger, R. Majumdar, D. Beyer, R. Jhala, G. Sutre, and A. Chilpala. Blast. `http://embedded.eecs.berkeley.edu/blast/`.

[25] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming*, 56(1-2):23–67, 2003.

[26] G. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[27] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *International Conference of Software Engineering (ICSE)*, 1992.

[28] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Programming Language Design & Implementation (PLDI)*, 1988.

[29] C. N. Ip and D. L. Dill. Better verification through symmetry reduction. *International Journal on Formal Methods in System Design (FMSD)*, 1996.

[30] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 1, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[31] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[32] C. W. Keller, D. Saha, S. Basu, and S. A. Smolka. Focuscheck: A tool for model checking and debugging sequential c programs. In *Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, volume 3440, pages 563–569, Edinburgh, UK, 2005. Springer.

[33] Y. Kesten and A. Pnueli. Control and data abstraction:the cornerstones of pratical formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 2000.

[34] A. S. Miner. Saturation for a general class of models. In *1st Int. Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 282–291, Enschede, The Netherlands, Sept. 2004.

[35] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Application and Theory of Petri Nets 1999*, LNCS 1639, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag.

[36] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[37] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, 1982.

[38] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *Application and Theory of Petri Nets 1995*, LNCS 935, pages 374–391, Turin, Italy, June 1995. Springer-Verlag.

[39] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc. Ser. 2*, 42:230–265, 1936.

[40] M. Weiser. Programmers use slices when debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.