

STATIC PROGRAM TRANSFORMATIONS FOR EFFICIENT SOFTWARE MODEL CHECKING

Shobha Vasudevan
Computer Engineering Research Center
The University of Texas at Austin
Austin, Texas, U.S.A.
shobha@cerc.utexas.edu

Jacob A. Abraham
Computer Engineering Research Center
The University of Texas at Austin
Austin, Texas, U.S.A.
jaa@cerc.utexas.edu

Abstract

Ensuring correctness of software by formal methods is a very relevant and widely studied problem. Automatic verification of software using model checking suffers from the state space explosion problem. *Abstraction* is emerging as the key candidate for making the model checking problem tractable, and a large body of research exists on *abstraction based verification*. Many useful abstractions are performed at the syntactic and semantic levels of programs and their representations.

In this paper, we explore abstraction based verification techniques that have been used at the program source code level. We provide a brief survey of these *program transformation* techniques. We also examine, in some detail, *Program Slicing*, an abstraction technique that holds great promise when dealing with complex software. We introduce the idea of using more specialized forms of slicing, *Conditioned Slicing* and *Amorphous Slicing*, as program transformation based abstractions for model checking. Experimental results using conditioned slicing for verifying safety properties written in temporal logic show the promise of these techniques.

Keywords: Formal verification, Model checking, Abstraction, Data abstractions, Abstract interpretation, Counterexample based refinement, Program slicing, Conditioned slicing, Amorphous slicing, Strong/Weak property preservation

1. Introduction

Designing a dependable complex system is a difficult problem and techniques for dealing with faults in a system have been studied widely by many researchers. A pioneer in this field is Dr. Algirdas Avizienis who developed the basic concepts, including the need for diversity in order to achieve fault tolerance [Avizienis and Kelly, 1984], [Avizienis and Laprie, 1986]. He also developed systematic approaches for designing fault-tolerant systems [Avizienis, 1997]. Reducing the number of design faults in the components is a necessary step in the design of a highly dependable system. This paper focuses on techniques to verify the correct operation of the software in the system.

As software begins to occupy a larger fraction of the overall system, faults in the software have a greater impact on system dependability, and ensuring dependability of the software portion of a system is becoming a leading concern. Toward this goal, software testing and debugging techniques as well as formal verification techniques are being explored as candidate solutions.

Among these, formal verification inspires the highest confidence due to the completeness of its approaches in ensuring correctness.

Model checking is an automatic technique used to formally verify programs [Clarke et al., 1986]. Temporal logic model checking typically requires a formal description of the model whose correctness needs to be established and a property specified in temporal logic. The main temporal logics used to specify properties are LTL [Manna and Pnueli, 1992], [Lichtenstein and Pnueli, 1985] CTL and CTL* [Clarke et al., 1986]. Complicated programs like hardware controllers and communication protocols have very large state spaces. In these cases, model checking techniques can suffer from state space explosion problems.

Since the number of states in the model grows exponentially with the number of variables and components of the system, model checking present day programs that have many hundreds of thousands of lines of code is computationally intractable.

In order to make model checking practically feasible, it is necessary to reduce the sizes of these models so that the computations have reasonable time and space requirements. It is also essential that the reduced models retain sufficient information to produce the same results as the original models, with respect to the properties being checked. These two requirements need to be balanced while creating these reduced models, i.e., while generating the *abstract* models from the original or *concrete* models. The process of constructing the abstract model from the concrete one is called *abstraction*. Abstractions are emerging as the key candidates for program verification using model checking.

Abstractions can be performed on the Kripke structure (state-transition model) of a program as well as on the program's source code. Abstraction techniques on Kripke structures are symmetry reduction [Emerson and Sistla, 1996], partial order reduction [Chou and Peled, 1996], cone of influence reduction [Krushan, 1994], parameterization, compositionality etc. Since the state space of even small programs can be extremely large, it may not be possible to build the Kripke structure for any reasonably sized program. In contrast, abstractions formed by static program analysis will scale well with program size, and are of high economic interest. We focus on these abstraction techniques based on *program transformations* in the rest of the paper.

This paper consists of three main parts. In the first, we give an overview of the abstraction techniques employed in the prior art for software model checking. We provide an extensive literature survey and also give a classification of the types of abstractions and their applicability. We then give an overview of *Program Slicing*, a program analysis technique that has been used for various software applications [Weiser, 1979]. We give some prior applications that use Weiser's *static* program slicing for creating abstractions for verification. In the third part, we explore more sophisticated slicing techniques, *Amorphous Slicing* and *Conditioned Slicing*. Our contribution to the state-of-the-art is to introduce the use of conditioned slicing as a new abstraction for software model checking. We provide some promising preliminary results on sample programs using the SPIN model checker.

2. Abstractions in Model Checking

Abstractions have been used extensively to reduce the computational complexity of model checking. The abstractions are *property preserving* [Loiseaux et al., 1995].

This implies that given a program and a property to be verified, the In this paper, we will mostly deal with abstractions that are created from t satisfaction of the property in the abstract program implies the satisfaction of the property in the concrete program. Property preservation can be *weak* or *strong*. Weak property preservation can be defined using the branching time μ -calculus defined in [Kozen, 1998]. Weak property preservation preserves the truth of properties from the abstraction to its concrete model.

A function α from the powerset of states of a state transition system S_1 to the powerset of states of another system S_2 is said to weakly preserve a property f , if for any state of S_1 that satisfies f , the states in S_2 also satisfy f .

If the converse is also true, then α is said to strongly preserve f . As a result, both truth and falsehood of properties are preserved from the abstraction to its concrete model. Diagnostic counter-examples are "carried over" to the concrete model.

Strong property preservation puts a lower bound on the size of suitable abstractions. Abstractions that result in strong property preservations are typically difficult to construct.

There exist two closely related frameworks for developing abstractions and proving their correctness. Simulation, [Milner, 1971], [Park, 1981] is about structural relation between abstract and concrete transition systems, representing the step relation of programs by means of an abstraction relation between abstract and concrete sets of states. Each concrete transition must be simulatable by an abstract transition.

Abstract interpretation [Cousot and Cousot, 1977] is the relation between concrete and abstract states by an abstraction function α from concrete sets of states to the smallest element of some abstract property lattice, which represents all the elements of the concrete sets. With α is associated γ , a concretization function, which associates with each abstract element the set of all concrete states represented by it, such that (α, γ) makes a Galois connection.

A difference between these two frameworks is that in the abstract interpretation framework, the computation of the abstract property is given emphasis. The theory of abstract interpretation formalizes the notion of *approximations*. The computation of abstract systems in simulation, however, does not lay emphasis on the precision of abstractions. In fact, simulation was proposed only in the context of strong preservation of properties.

Abstraction techniques are methods that can be used to construct abstractions from the concrete models. The decision as to which details need to be included in the abstraction (for the verification task) can be made manually or automatically.

Manual techniques include user chosen abstract interpretations. The manual construction of safe abstractions trades automation for generality, in the sense that no restrictions are imposed on the class of large (infinite) state systems amenable to the method. The abstractions considered are usually weakly preserving, which means that properties are preserved only from the abstraction to its concrete model. As a result, only the truth of properties is guaranteed to be preserved. The weak requirement guarantees the existence (at least for universal properties) of finite-state and arbitrarily small weakly preserving abstractions for any concrete model. Users are free to use their 'intuition' for the behavior of the concrete system to come up with an abstraction that they find suitable. There is no *a priori* guarantee regarding the number or the type of properties that can be verified using the abstraction. The appropriateness of the abstractions must be investigated by trial and error. The obligation of proving the safety of the abstraction, in general cannot be automated. Instead, the proof is done manually or with support from theorem provers.

Automatic construction of abstraction systems is more ambitious. Finite-state strongly preserving abstractions (which are small enough for model check-

ing) can only be automatically constructed for restricted kinds of large (infinite) state models. One such well-studied restricted domain is real-time systems, where continuous time gives rise to an infinite state space [Alur et al., 1993]. Only the systems whose behavior is guarded by certain linear constraint systems on the 'clock' variables are guaranteed to have finite-state strongly preserving abstractions that can be constructed automatically. Almost all existing model checkers for dense reactive systems (real time or hybrid) are based on automatically constructed strongly preserving abstractions. The idea is to let the abstract states be equivalence classes of concrete states, with respect to some behavioral equivalence or equivalence with respect to a property.

An abstraction technique is said to be *sound* if the abstract program is always guaranteed to be a conservative approximation of (*i.e.*, simulates) the original with respect to a set of specification properties. This means that a property holds in the original program if it holds in the abstraction. An algorithm is said to be *exact*, if the abstraction it constructs is bisimilar to the original program. This means that a property holds in the original program if it holds in the abstraction, and the converse is also true. An abstraction technique is *complete* if the algorithm will always find a finite state abstract program for the original program, if one exists. In terms of simulation, if the state-transition graph of the original program has a finite simulation quotient, then the algorithm can produce a finite simulation equivalent abstract program.

Abstractions can be organized broadly into data, control, configuration or communication abstractions. Data abstractions operate on data values and operations on the data. Control abstractions operate on the order of the operations within a process. Configuration and communication abstractions operate on the order of processes in a program as well on as the communication between programs.

Data Abstractions

Data abstractions abstract away some of the data information for creating smaller models. The abstract model can be derived from some high level description of the program like the program text. Data abstractions are typically manual abstractions. In [Clarke et al., 1994], the abstract transition systems are obtained by computing the abstractions of primitive operators as defined previously.

The programs are modeled as transition systems where states are n -tuples of variable values. The set of all program states is expressed as $D_1 \times D_2 \times \dots \times D_n$. A surjective function h maps each D_i onto a set of abstract values. The surjection then maps program states to abstract states. The resulting abstraction (called minimal abstraction by the authors) is approximated by statically analyzing the text of the program. These abstractions show weak property preser-

vation. The approximated abstract model may demonstrate more behaviors than the concrete model, but it is easier to build and verify. The abstractions proposed in this work include arithmetic operation abstractions (such as congruence modulo an integer), single bit abstractions for dealing with bitwise logical operations, product abstractions for combining the effect of abstractions, and symbolic abstractions.

For verifying programs involving arithmetic operations, congruence modulo a specific integer may be a useful abstraction. Thus, for any $h(i)$, i is replaced by $i \bmod m$. The values of an expression are, therefore, constrained to a smaller range, depending on m . When comparing the orders of magnitude of some quantities, the logarithmic representation is used instead of the actual data value. For programs that have bitwise logical operations, a large bit vector is abstracted to a single bit value, according to some function such as a parity generator. Symbolic abstractions can be used in situations where the enumeration of the data values is cumbersome. If the data value is changed to a symbolic parameter, there can be significant savings.

Kurshan [Kurshan, 1990] uses a similar framework for data abstractions for finite state system verification. The abstractions in this case, however, are not approximated by static analysis of the program text. Instead, they are computed as language homomorphisms in the algebra. The homomorphisms are specified by the user between the actual and the abstract processes, but are checked automatically.

In all the above abstractions, the infinite behavior of the system, resulting from the presence of variables with infinite domains, is abstracted. The abstractions are computed by means of abstract data types. For each variable to be abstracted, the abstract domain and the operations on the sets representable in the abstract domain are defined. An abstract model is then obtained by replacing each variable by one in the abstract domain and each operation by an abstract one. These are general predefined abstractions that are not aimed at specific properties.

Some variations to the above data abstraction techniques are found in [Bharadwaj and Heitmeyer, 1999]. Here, the focus is on properties of single states or transition state-pairs rather than execution sequences. The abstractions use *variable restriction* that eliminates certain variables. The data types of each eliminated variable are abstracted to a single value. This work also constructs abstractions with respect to a single property, as opposed to the generic abstractions constructed by other methods.

Data independent systems are those where the data values do not affect the control flow of the computation. In these cases, the datapath can be abstracted away entirely [P.Wolfer and V.Lovinfosse, 1990].

In hardware verification, important abstraction techniques use data abstractions [Moundanos et al., 1998] and uninterpreted function symbols [Bryant et al., 2001].

Abstract interpretation based abstractions

Many abstraction techniques can be viewed as applications of the abstract interpretation framework [Sifakis, 1983], [Cousot and Cousot, 1999], [Cousot, 2003].

The abstract interpretation framework establishes a methodology based on rigorous semantics for constructing abstractions that overapproximate the behavior of the program, so that every behavior in the program is covered by a corresponding abstract execution. Thus, the abstract behaviors can be exhaustively checked for an invariant in temporal logic. In abstract interpretation, abstractions are usually defined *a priori* for a particular type of analysis. The abstract version of the language semantics is constructed once with manual assistance. Producing a new abstract semantics for on-the-fly verification is a non-trivial task. Some tools such as Cospan [Kurshan, 1994], [Kurshan, 1990] and Bandera [Corbett et al., 2000], [Dwyer et al., 2001] try to do this task automatically.

Although data abstractions can be included in the realm of abstract interpretations, they form only a part of the possible abstract interpretations. Also, they may not hold over all of the system's execution semantics. Informally, abstract interpretation has a domain of abstract values, an abstraction function mapping concrete program values to abstract values and a set of abstract operations.

Some common abstract interpretations are briefly described for providing a flavor of the technique. *Sign abstraction* consists of replacing integers by their sign and ignoring their actual value. Such abstractions may be useful if there is a proposition like $x = 0$ for some integer x . Since all the information about x is not required, the sign abstraction may be applied, which keeps track of whether x is greater than (gt), less than (lt) or equal to zero (eq). The powerset of the abstract domain is now $\{\text{gt}, \text{lt}, 0\}$. Now, all the primitive operations are defined over this abstract domain, such that they satisfy every program execution.

Another common abstraction is *interval abstraction*, that approximates a set of integers by its maximal and minimal values. Thus, if a counter variable appears in a property, the counter can be replaced by the lower and upper limits of the counter.

Relational abstractions retain the relationship between a set of data values. For instance, a set of integers can be approximated by its convex hull. Abstract interpretations of functions is done by maintaining the parameter and the result (signature) of the function in the abstract domain. Fixpoint iteration can also

be thought of as an abstraction. A detailed treatment of these abstract interpretations can be found in [Cousot, 2001]. Abstract interpretation for μ -calculus is shown in [Dams, 1996], [Dams et al., 1997].

Since abstract interpretations are not specific to any given property or for any given program, they have the power of generality. The technique considers predefined specifications for all possible programs of a given language. However, this results in the practical problem of computing these abstract interpretations during static analysis, before the verification task begins. The theory of abstract interpretation is mainly concerned with soundness, and not completeness.

Counterexample Guided Refinement

A highly researched field is abstraction refinement techniques for model checking [Lee et al., 1996], [Pardo, 1997], [Pardo and Hachtel, 1998]. These techniques are typically automatic abstraction techniques. Counterexample guided abstraction refinement techniques have been widely studied. An approximation of the set of states that lie on a path from the initial state to a bad state is successively refined. The refinement is done by forward or backward passes, where each pass uses (or refines) the approximation computed by the previous pass. This process is repeated until a fixpoint is reached. If the resulting set of states is empty, the property is proven, since no bad state is reachable. Otherwise, the method does not guarantee that the counterexample trace found is genuine. In other words, the counterexample could be spurious due to the overapproximations. A heuristic is used to find a subset of the reachable states from the initial states. If there is a match, the error is genuine and can be reported as a bug.

Cho *et al.* [H.Cho et al., 1996] propose symbolic forward reachability algorithms that induce an overapproximation. The state bits are partitioned into mutually disjoint subsets and do a symbolic forward propagation on individual subsets. Some approaches also use symbolic backward reachability analysis [Cabodi et al., 1994]. Govindaraju and Dill [Govindaraju and Dill, 2000], [Govindaraju and Dill, 1998] allow for overlapping subsets as opposed to the mutually disjoint ones, and present a more refined approximation as compared to earlier schemes.

A model checker that uses upper and lower approximations to verify properties in temporal logic was proposed in [Lind-Nielsen and Andersen, 1999]. These approximation techniques guarantee completeness without rechecking the model after each refinement. A similar approach has been described in [Balarin and Sangiovanni-Vincentelli, 1993], and in Kurshan's localization reduction [Kurshan, 1994]. These are iterative techniques that are based on the

variable dependency graph. The localization reduction either leaves a variable unchanged, or replaces it by a non-deterministic assignment.

Predicate Abstraction. This technique was first introduced by Graf and Saidi [Graf and Saidi, 1997]. The predicates are related to the property that is being verified and are automatically extracted from the program text. Das and Dill [Das et al., 1999] use this technique with some variation to formally verify complex systems. While Graf and Saidi used monomials to represent the abstract state space, this work uses Binary Decision Diagrams (BDDs) as the representation. Clarke et al describe a related technique in [Clarke et al., 2000]. This work is based on *atomic formulas* that correspond to the predicates, but are used to construct an abstraction function. The abstraction function maintains a relationship between the formulas instead of treating them as individual propositions. The authors also introduce symbolic algorithms to determine if the abstract counterexamples are spurious. If a counterexample is spurious, the shortest prefix of the abstract counterexample that does not correspond to an actual trace in the concrete model is identified. The last abstract state in this prefix (the failure state) needs to be split into fewer abstract states by refining the equivalence classes in such a way that the spurious counterexample is eliminated. The extension of these forward algorithms for analyzing counterexamples to backward algorithms that do the same, are found in [Bensalem et al., 2003]. This can lead to completely different abstractions. This technique can also handle loop unfolding.

A related approach that performs predicate abstractions by syntactic program transformations automatically is presented in [Namjoshi and Kurshan, 2000]. The algorithm starts from the predicates in the specification formula. Predicates of the original boolean program are represented by Boolean variables in the abstraction. To preserve the correspondence between the predicate and the Boolean variables, the weakest precondition is calculated syntactically, and the Boolean variables are updated iteratively. This method of constructing abstractions is sound and complete. For programs with bounded non-determinacy, the algorithm does not need manual intervention using theorem provers to compute the abstract program, as other predicate abstraction based methods do.

Lazy Abstraction. A more efficient method to compute abstractions in this paradigm is *Lazy Abstraction*, presented in [Henzinger et al., 2002],[Henzinger et al., 2003]. Intuitively, lazy abstraction proceeds as follows. The abstract state in which the abstract counterexample fails to have a concrete counterpart is called the pivot state. The pivot state suggests which predicates should be used to refine the abstract model. However, instead of building an entire new

abstract model, the refinement of the abstract model is done “from the pivot state on”.

Abstraction is done on-the-fly, and only up to the precision necessary to rule out spurious counterexamples. On-the-fly construction of an abstract transition system eliminates an often wasteful and expensive model construction phase. Model checking only the “current” portion of the abstract transition system saves the cost of unnecessary exploration in parts of the state space that are already known to be free of errors. The lazy abstraction algorithm terminates under a customary condition on the predicate theory (no infinite ascending chains of predicates) and an abstract condition on the program (finite trace equivalence), which has been established for many interesting classes of infinite-state systems.

Lazy abstraction is sound, since the counterexample refinement phase rules out false positives. In case an error is found, the model checker also provides a counterexample trace in the program showing how the property is violated. Automatic abstraction allows running the analysis directly on an implementation, rather than constructing an abstract model that may or may not be a correct abstraction of the system. The authors show that by always maintaining the minimal necessary information to validate or invalidate the property, lazy abstraction scales to large systems.

Earlier applications of the predicate abstraction type of the abstract interpretation approach [S. Graf and H. Saidi, 1997], [Bensalem et al., 1998], [Colon and Uribe, 1998] were dependent on the user identifying the set of predicates that influence the verification property and used general-purpose theorem proving to compute the abstract program. The user-driven discovery of relevant predicates makes them less effective for large programs. Recently, various decision procedures have been proposed to compute the set of predicates for the abstraction. The most common approach is to use error traces to guide the discovery of predicates. In [Clarke et al., 2000], the algorithm is based on BDD representations of the program. This is a draw back for large programs, where transition relation BDDs are commonly too large for efficient manipulation.

In [Ball and Rajamani, 2001], the SLAM toolkit is introduced, that generates an abstract *Boolean program* from a C program and a set of predicates. The SLAM tools can be used to find loop invariants expressible as Boolean functions over a given set of predicates. The loop invariant is computed by the model checker Bebop [Ball and Rajamani, 2000] using a fixpoint computation on the abstraction. Boolean programs are programs with the usual control flow constructs of an imperative language such as C, but in which all variables are of Boolean type. Boolean programs contain procedures with call-by-value parameter passing and recursion, and a restricted form of control nondeterminism. Since the amount of storage a Boolean program can access at any point is finite, questions of reachability and termination are decidable in the realm of

Boolean programs. The generation of the Boolean program is done by calling a theorem prover for each potential assignment to the current and next state predicates. The number of theorem prover calls could be very high. Several heuristics are used to reduce this number. Existing tools stop the computation after a user-specified number of calls, and add all remaining transitions for which the theorem prover call was skipped. This is a safe over-approximation, but will yield a potentially large number of unnecessary spurious counterexamples.

Other abstraction techniques

Some abstraction techniques that are highly automatic, but very specific to the program and the property being verified, are *variable hiding* [Dams et al.,] and program slicing [Weiser, 1984].

Variable Hiding. Variable hiding is a powerful program transformation technique that was used in [Dams et al.,], [Holzmann and Smith, 2000] for verifying C programs. This is an iterative refinement technique that creates overapproximations. In the first iteration, all assignments and function calls that are irrelevant to the property being verified are replaced with a no-op. All conditional choices that refer to irrelevant statements in the program are replaced by *nondeterministic choices*. The use of nondeterminism is a standard reduction technique that can be used to make a model more general. The nondeterminism tells the model checker that instead one specific computation, all possible outcomes of a choice should be considered equally possible. The original computation of the system is preserved as one of the possible abstracted computations, and the scope of the verification is therefore not restricted. If no property violation exists in the reduced system, we can safely conclude that no property violation can exist in the original application.

The resulting abstraction has weak property preservation. It is possible, for instance, that the full expansion of an error trace for a property violation detected in the abstraction does not correspond to a valid execution of the original application. If this happens, it constitutes a proof that the abstraction was too coarse. In that case, the counterexample generated provides clues for including some more statements to make the abstraction less coarse. Typically a few iterations of this type suffice to converge on a stable definition of an abstraction that can be used to extract a verifiable model from a program text.

An example of variable hiding is illustrated. A piece of code and the program transformations that are generated by variable hiding are given below.

```
h = A[i];  
r = r + (++A[i]);  
res = r + h;
```

```

if (r > MAX)
{
  m++;
  r = 0;
}

```

If the property involves m or h , we obtain the following abstraction after variable hiding. The non-determinism is introduced in the conditional statement, where the variables r or MAX are not present in the property.

```

h = A[i];
if (NONDET)
{
  m++;
}

```

An important abstraction paradigm is *program slicing* [Weiser, 1984],[Weiser, 1979]. We explore this technique in some detail in the next section, in the context of program verification.

3. Static Program Slicing

Slicing, in a general sense, is a program transformation which preserves some projection of the semantics of the original program. A particular approach to slicing is defined by describing the aspect of the program to be preserved and the nature of the transformations to be performed upon the program to construct the slice. The aspect of the program that must be preserved is captured by the slicing criterion. The published work on slicing is concerned with a very simple transformation: statement deletion. Therefore, slicing is the process of deleting commands from a program, while preserving some aspects of the behavior as captured by the slicing criterion. The original definition of a program slice was presented by Weiser [Weiser, 1979]. Only statically available information is used for computing slices; hence this type of slice is referred to as a *static slice*. Since then, various slightly different notions of slicing have been proposed, as well as a number of methods to compute them. Program slicing has been used in several tasks including testing, debugging, maintainence, complexity analysis, comprehension, reverse engineering, re-engineering and reuse. Recently, slicing has also been used as an abstraction for verification. We present here, some results on slicing for verification.

DEFINITION 1 *Static slicing criterion*

A slicing criterion of a program P with an input alphabet Σ , is a pair $\langle i, V \rangle$ such that i is a statement in P and $V \subseteq \Sigma$. A set of statements I_s is said to *affect* the values of V at i in a given slicing criterion $\langle i, V \rangle$, if I_s defines a subset of V that is used in i .

DEFINITION 2 *Static slice for programs*

A slice S of a program P on a slicing criterion $\langle i, V \rangle$ is a subset of the statements of P that might affect the values of V at i .

An alternative method for computing static slices was suggested by Ottenstein and Ottenstein, [Ottenstein and Ottenstein, 1984] who restate the problem of static slicing in terms of a reachability problem in a program dependence graph (PDG). A PDG is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependences. The slicing criterion is identified with a vertex in the PDG, and a slice corresponds to all the PDG vertices from which the vertex under consideration can be reached.

Static Program Slicing for Verification

Static slicing has been used as a program transformation for software model checking. [Hatcliff et al., 2000], [Millett and Teitelbaum, 1998]. Slicing allows for strong preservation of properties. The primary advantage of slicing in verification is that the construction of the abstraction (slice) is completely automatic. State-of-the-art verification tools incorporate program slicing as a feature [Corbett et al., 2000]. Slicing is a very simple program transformation that only effects the syntax of the original program. Therefore, slicing can be used in conjunction with all the partial evaluation based abstraction techniques. The software can be preprocessed using slicing before applying the abstraction techniques. Slicing provides a safe approximation of the relevant portions of code and enables the scaling of abstraction based techniques and tools to more complicated systems. All other abstraction techniques, can thus be seen as complements to slicing. The property being verified is written in temporal logic, and the propositions within the temporal logic property form the variables in the slicing criterion.

Slicing is different from other abstraction techniques that sacrifice completeness for tractability and generality. While other techniques preserve correctness with respect to a generic class of properties, slicing preserves correctness with respect to a specific property. The abstractions created by slicing are sound and complete with respect to the property being checked.

Static slicing can be likened to the cone-of-influence reductions done by model checkers, since both these transformations have the same semantics. However, slicing is done at the source code level, as opposed to the cone-of-influence reduction. Slicing, therefore, is a pre-encoding mechanism, that does not build the state transition graphs. Hence, despite being the same type of abstraction as the cone-of-influence reductions, it still shows a benefit in performance.

In [Hatcliff et al., 2000], an interesting application of program slicing to verification has been illustrated. Slicing is used to assist the creation of relevant

abstract interpretations for abstraction based verification. Selecting appropriate abstract interpretations can be a non-trivial task for the user. The methodology of finding the variables in a program that can influence the program’s execution, (relative to a property’s propositions) is essentially based on heuristics. When a variable is determined to be potentially influential, its abstraction is refined to strengthen the resulting system model. If the variable is not found to be potentially influential, it is modeled with a point abstraction that ignores any effect it may have, until there is a case where the variable needs to be refined. The authors show that the information produced by pre-processing the program with slicing is exactly what they need to provide automated support for selecting appropriate abstract interpretations. Specifically, slicing identifies relevant variables, eliminates irrelevant program variables from consideration in the abstraction selection process, and reduces the size of the software (and thereby the size of the transition system) analyzed.

4. Specialized Slicing Techniques

We introduce the idea of using some specialized types of program slicing for verification. These slicing techniques have been used for a number of applications. However, to the best of our knowledge, this is the first application of these techniques for verification. Traditional static slicing produces very large slices [Korel and Laski, 1990]. Since these specialized slicing techniques are used to create smaller slices, we can also extend their use to verification, by exploiting the reduction in verification state space. Also, the sophisticated slicing techniques we employ for verification are not semantically equivalent to cone-of-influence reductions. We present some of our ideas on slicing based verification, and preliminary experimental results on some sample programs.

Amorphous Slicing

A variation of traditional program slicing called *amorphous slicing* [Harman et al., 2003], can produce smaller slices by abandoning the traditional requirement of syntax-preservation. Traditional, syntax-preserving program slicing simplifies a program by removing program components (i.e., statements and predicates) that do not affect a computation of interest. The resulting slice captures a projection of the semantics of the original program. In addition, traditional slicing requires that a subset of the original program’s syntax be maintained. This syntactic requirement is important when slicing is applied to cohesion measurement, algorithmic debugging and program integration. However, for applications such as re-engineering, program comprehension and testing, it is primarily the semantic property of a slice that is of interest.

DEFINITION 3 *Amorphous Slicing for programs*

An amorphous slice for a program P with respect to a slicing criterion (i, V) is a program q , such that starting from the same initial state, the same state is reached with respect to the variables in V at point i in both P and q .

An example of amorphous slicing, as explained in [Harman et al., 2003] is given below.

```
for(i=0, sum=a[0], biggest=sum; i<19; sum=a[++i])
  if (a[i+1] > biggest)
  {
    biggest = a[i+1];
    average = sum/20;
  }
```

The fragment was written with the intention that the variable `biggest` would be assigned the largest value in the 20-element array `a`, and that the variable `average` would be assigned the average of the elements of `a`. However, the fragment contains a bug which affects the variables `sum` and `average`, but not the variable `biggest`. To illustrate amorphous slicing, the variables `biggest` and `average` will be analyzed using both traditional syntax-preserving slicing and amorphous slicing. The static slice is given below.

```
for(i=0, sum=a[0], biggest=sum; i<19; sum=a[++i])
  if (a[i+1] > biggest)
  {
    biggest = a[i+1];
  }
```

However, the amorphous slice, constructed using transformations such as a loop unrolling (which has changed the loop bounds) is the following.

```
for(i=1, biggest=a[0]; i<20; ++i)
  if (a[i] > biggest)
  {
    biggest = a[i];
  }
```

The amorphous slice for the variable `average` is given below.

```
average=a[19]/20;
```

It can be seen that there is a bug in the program, from the amorphous slice.

Amorphous Slicing for Verification. We propose the idea of using amorphous slicing for verification. Since abstraction based verification does not require syntactic preservation of the program behavior, this slicing technique can

be exploited to form meaningful abstractions in this context. A program can be sliced with respect to a slicing criterion (i, V) , where V is a part of a property written in temporal logic. The resulting amorphous slice is an abstraction, that can now be model checked.

We give an example of how amorphous slicing could be used for verification. Consider the following piece of code

```
begin
  i = start;
  while (i <= (start + num))
  {
    result = K + f(i);
    sum = sum + result;
    i = i + 1;
  }
end
```

Consider the LTL property $G(\text{sum} > K)$. This property checks if the value of $f(i)$ is greater than 0. The slicing criterion derived from the property is $(\text{end}, \{\text{sum}, K\})$. The program transformations that can be applied on the code for amorphous slicing correspond to a couple of rules presented in [Harman et al., 2003]. The induction variable elimination rule is applied as a transformation. It replaces qualifying loops with two assignments that capture the requirements on array safety. The second assignment is the value of i in the final iteration of the loop. In cases where the loop variables are very large numbers, such a transformation would yield useful savings. The dependent assignment removal rule is applied for eliminating redundant assignment statements. The resulting amorphous slice is shown below

```
begin
  sum = sum + K + f(start);
  sum = sum + K + f(start + num);
end
```

We notice that the amorphous slice is a reduced program. When executed, the amorphous slice takes a fraction of the time taken to verify the original code on the SPIN model checker.

Amorphous slicing can be viewed as a type of term rewriting, that has been used by theorem provers for deductive verification. The difference is, however, in the fact that theorem provers and rewriters try to prove a property, entirely by rewriting. In the method that we propose, the rewriting is done only on a part of the program, that is decided by the slicing criterion. After the program has been reduced to a certain desired state, model checking can be used. The

advantage in this methodology is that the manual component of rewriting, as well as the task of proving the transformations correct, is now vastly reduced. We expect this approach to yield beneficial results.

Amorphous slicing does not guarantee that the resulting slice has the same signature (variables and functions) as the original program. Since amorphous slicing can allow any transformation to the program, and does not require any faithfulness to the original program, proving the correctness of the transformations is very important.

In order to prove these transformations correct, two approaches can be taken. The set of transformations or rules can be collected to form a rule base. The transformations in this rule base will then be proved correct separately. This rule base will be incomplete, but will have a high efficiency, due to fewer program states. Another procedure could be to prove the program transformations correct in an abstract interpretation framework, for a given concrete operational semantics of the language. Transformations can be control-based as well as data-based in amorphous slicing. Whenever slicing results in the elimination of the nodes in the control flow graph, it is possible to visualize this slicing as a form of rewriting. Whenever the slicing includes data transformations, the technique is better expressed in the abstract interpretation framework. An example of the rewriting rule base has been given in [Harman et al., 2003].

Conditioned Slicing

Canfora et al [Canfora et al., 1998] introduced the notion of *conditioned slicing*, that forms a theoretical bridge between static and dynamic slicing [Korel and Laski, 1988]. Conditioned slices are constructed with respect to a set of possible input states, characterized by a first order predicate logic formula. Conditioned slicing augments static slicing by introducing a condition that specifies the initial set of states in the criterion. This slicing technique, therefore allows slicing with respect to the initial states of interest, or initial constraints in the program. We present some basic definitions of conditioned slicing that appear in the literature.

DEFINITION 4 *Conditioned Slicing criterion*

Let Σ be the set of input variables to the program P . Let C be a first order predicate logical formula on the variables in Σ . A conditioned slicing criterion is a triple $\langle C, i, V \rangle$, where i is a statement in the program, and $V \subseteq \Sigma$.

DEFINITION 5 *Conditioned Slicing for programs*

A conditioned slice of a program P on a conditioned slicing criterion $\langle C, i, V \rangle$ consists of all the statements and predicates of P that might affect the values of the variables in V at i , when the condition C holds true.

Tip [Tip, 1995] introduced a more restricted form of conditioned slicing called constraint based slicing. In all these cases, the *condition* that specifies

the set of initial states, and is used for slicing is a first order predicate logic formula. We will refer to this condition as the *conditional predicate*, or simply *predicate*. We will use the term *conditioning* to mean the process of obtaining a conditioned slice with respect to a given conditional predicate C .

Conditioned slicing is a significant improvement over static, dynamic or quasi-static [Venkatesh, 1991] slicing, since it subsumes all of these as special cases [Canfora et al., 1998].

The static slicing criterion is captured by conditioned slicing, when the conditional predicate is always *true*. The slicing criterion then captures program behavior, regardless of initial state.

In situations where the initial set of constraints for the program analysis are known, this technique can be employed to get much smaller slices than those produced by static slicing. This technique can therefore be used to simplify the code, before applying a traditional static slicing algorithm. Conditioned slicing has been automated with significant success on C and WSL code [Daoudi et al., 2002], [Danicic et al., 2000].

Conditioned Slicing for Verification. Our technique aims at reducing state space of the program, by slicing away the part of the program irrelevant to the property being verified. We focus on safety properties that can be specified as temporal logic formulae of the form, *antecedent* \implies *consequent*. For these properties, we can use the antecedent to specify the set of initial states that we are interested in. *The antecedent therefore, forms the condition in the slicing criterion.* All the statements that would get executed when the antecedent is true (or the condition is satisfied) are retained in the slice. The statements on the paths that cannot get executed when the antecedent is false, are removed. The reduced program still preserves its behavior *with respect to the property being checked*. We therefore create property preserving abstractions using conditioned slicing.

All prior art in verification using program slicing uses static program slicing techniques. While slicing property specifications of the form *antecedent* \implies *consequent*, these techniques retain the set of all statements of the program where the antecedent is true, *as well as those where it is not*. This is because static slicing retains all possible executions of the relevant variables.

However, in property based verification, *we do not need to check the states where the antecedent is false*. In these cases, static slices might be too large and include statements that are not of interest. We introduce a precise abstraction on the basis of conditioned slicing, *Antecedent Conditioned Slices*. We present an example to show how conditioned slicing can be performed for a given property. Let the property that needs to be verified be $G((N < 0) \implies (B = f(A)))$. The slicing criterion for all slicing techniques will be extracted from this property. The static slice for the code in Figure 1 for the slicing criterion

```

begin
1:      read(N);
2:      A = 1;
3:      if (N < 0)
         {
4:          B = f(A);
5:          C = g(A);
         }
      else
6:          if (N > 0)
             {
7:              B = f'(A);
8:              C = g'(A);
             }
          else
             {
9:              B = f''(A);
10:             C = g''(A);
             }
11:     print(B);
12:     print(C);
end

```

Figure 1. Example Program written in psuedocode

$\langle 11, B \rangle$ would be as shown in Figure 2. The conditioned slice for the same code, with respect to the slicing criterion $\langle C, 11, B \rangle$, where C corresponds to the predicate $(N < 0)$ would be as shown in Figure 3. This shows that the conditioned slice is much smaller in size than the static slice.

Preliminary Results. We provide experimental results of conditioned slicing using the SPIN model checker on the Group Address Registration Protocol and the X.509 Authentication Protocol. The Group Address Registration Protocol (GARP) is a datalink-level protocol for dynamically joining and leaving multicast groups on a bridged LAN. The X.509 is a CITT recommendation for an authentication protocol. It gives details of a method of allowing a user agent to send a password to a system agent in a safe manner that is not vulnerable to interception or replay. The source code and the properties can be found in [Lafuente, 2002].

All experiments were performed using a 450 MHz Pentium dual processor with 512 MB RAM. A memory limit of 512 MB was given for running these properties, with a max search depth of 2^{20} steps. The time taken to

```

begin
1:      read(N);
2:      A = 1;

3:      if (N < 0)
4:          {
          B = f(A);
          }
        else
6:          if (N > 0)
7:              {
              B = f'(A);
              }
          else
9:              {
              B = f''(A);
              }
11:     print(B);

end

```

Figure 2. Static Slice of program

```

begin
1:      read(N);
2:      A = 1;
3:      if (N < 0)
4:          {
          B = f(A);
          }
end

```

Figure 3. Conditioned Slice of program

check the properties, in seconds, is given in Table 1. The conditioned slicing of the source code was based on the properties that were written in the form *antecedent* \implies *consequent*.

Property P1 is from the assertions present in the source code of the GARP. Property P4 is from the assertions that find the protocol errors in the X.509

Table 1. Performance of conditioned slicing over regular model checking

Property	Unsliced	Conditioned Sliced	Property Proved
P1	91.65	1.72	Yes
P2	145.78	8.44	Yes
P3	145.36	8.41	Yes
P4	154.96	1.95	Yes
P5	117.81	10.23	Yes

source code. In these cases, the conditionals in the code presented the antecedent, while the consequents were the assertions themselves. Properties P2, P3 and P5 were written as LTL properties.

P2 corresponds to the LTL property $G((p \wedge (\neg(m))) \implies (\neg(q) \wedge \neg(r)))$, and P3 corresponds to the LTL property $G((p \wedge (\neg(m))) \implies (\neg(r)))$ where

```
p = empty(llc_to_regist[i])
m = (leavetimer != true)
q = (r_state != lv_imm)
r = (r_state != out_reg)
```

P5 corresponds to the LTL property $G((p) \implies (q))$ where

```
p = (macuser1[pid]@user1_end)
q = (r_state != out_reg)
```

Thus, conditioned slicing yields very promising results, as is evident from the table.

5. Conclusion

Abstraction techniques are necessary for the analysis of complex systems with very large state spaces. The table below gives a comparison of all the abstraction techniques we have mentioned in this paper. The comparison criteria are the degree of automation, the generic nature and applicability of the technique, the amount of dependence of the technique on individual properties for verification and finally the type of abstraction that we obtain by the technique.

The various program transformation techniques that we have discussed in this paper are steps toward building dependable computing systems. State-of-the-art tools must incorporate many of these techniques to deal with design faults in software. Effective analysis of programs entails that the techniques be applied compositionally. “A verification methodology is judged on the basis of how well it conciliates *correctness*, *automation*, *precision*, *scaling up* and *performance efficiency*” [Cousot, 2003]. “Software reliability is the grand

Table 2. Comparison of program transformation techniques

Technique	Automation	Generality	Property Dependence	Type of Abstraction
Data Abstractions	Low	High	Low	Overapproximation
Abstract Interpretation	Low	High	Low	Overapproximation
Counterexample guided refinement	Medium	Medium	High	Overapproximation
Predicate Abstraction	Medium	Medium	High	Overapproximation
Lazy Abstraction	Medium	Medium	High	Overapproximation
Variable Hiding	Medium	Low	High	Overapproximation
Amorphous Slicing	Low	High	High	Depends on type of rewrite
Static Slicing	High	Low	High	Exact
Conditioned Slicing	Medium	Low	High	Exact

challenge of the next decade” [Cousot, 2001]. It is thus the great challenge of the computer science community to analyze these factors for all abstraction based verification techniques and integrate these techniques optimally into a framework, such that the correctness and robustness of the system is ensured for all programming paradigms.

References

- Alur, Rajeev, Henzinger, Thomas A., and Ho, Pei-Hsin (1993). Automatic symbolic verification of embedded systems. In *IEEE Real-Time Systems Symposium*, pages 2–11.
- Avizienis, A. and Laprie, J. (1986). Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74:629–638.
- Avizienis, A. (1997). Toward systematic design of fault-tolerant systems. *Computer*, 30(4):51–58.
- Avizienis, A. and Kelly, J. P. J. (1984). Fault tolerance by design diversity: Concepts and experiments. pages 67–80.
- Balarin, F. and Sangiovanni-Vincentelli, A. L. (1993). An iterative approach to language containment. In *Proceedings of the 5th International Conference on Computer Aided Verification*, pages 29–40.
- Ball, Thomas and Rajamani, Sriram K. (2000). Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130.
- Ball, Thomas and Rajamani, Sriram K. (2001). Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057:103–122.
- Bensalem, Saddek, Graf, Susanne, and Lakhnech, Yassine (2003). Abstraction as the key for invariant verification. In *Symposium on Verification celebrating Zohar Manna’s 64th Birthday*.
- Bensalem, Saddek, Lakhnech, Yassine, and Owre, Sam (1998). Computing abstractions of infinite state systems compositionally and automatically. In *Computer-Aided Verification, CAV ’98*, volume 1427, pages 319–331.

- Bharadwaj, R. and Heitmeyer, C. L. (1999). Model checking complete requirements specifications using abstraction. *Automated Software Engineering: An International Journal*, 6(1):37–68.
- Bryant, R. E., German, S., and Velev, M. N. (2001). Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Trans. Comput. Logic*, 2(1):93–134.
- Cabodi, G., Camurati, P., and Quer, S. (1994). Symbolic exploration of large circuits with enhanced forward/backward traversals. In *Proceedings of the conference on European design automation*, pages 22–27.
- Canfora, G., Cimitile, A., and Lucia, A. De (1998). Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40:595–607.
- Chou, C. and Peled, D. (1996). Formal verification of a partial-order reduction technique for model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 241–257.
- Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263.
- Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169.
- Clarke, Edmund M., Grumberg, Orna, and Long, David E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542.
- Colon, Michael and Uribe, Tomas E. (1998). Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304.
- Corbett, James C., Dwyer, Matthew B., Hatcliff, John, Laubach, Shawn, Păsăreanu, Corina S., Robby, and Zheng, Hongjun (2000). Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448.
- Cousot, P. (2001). Abstract interpretation based formal methods and future challenges, invited paper. In *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156.
- Cousot, P. (2003). Automatic verification by abstract interpretation, invited tutorial. In *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, pages 20–24.
- Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252.
- Cousot, P. and Cousot, R. (1999). Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95.
- Dams, D. (1996). *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology.
- Dams, D., Gerth, R., and Grumberg, O. (1997). Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291.
- Dams, D., Hesse, W., and Holzmann, G. J. Abstracting C with abC. pages 515–520.
- Danicic, S., Fox, C., Harman, M., and Hierons, R. (2000). Consit: A conditioned program slicer. pages 216–226.
- Daoudi, M., Ouarbya, L., Howroyd, J., Danicic, S., Marman, Mark, Fox, Chris, and Ward, M. P. (2002). Consus: A scalable approach to conditional slicing. In *IEEE Proceedings of the Working Conference on Reverse Engineering*, pages 181–189.

- Das, S., Dill, D., and Park, S. (1999). Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171.
- Dwyer, M. B., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu, C. S., Robby, Zheng, H., and Visser, W. (2001). Tool-supported program abstraction for finite-state verification. In *International Conference on Software Engineering*, pages 177–187.
- Emerson, E. Allen and Sistla, A. Prasad (1996). Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131.
- Govindaraju, S. G. and Dill, D. L. (1998). Verification by approximate forward and backward reachability. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 366–370. ACM Press.
- Govindaraju, Shankar G. and Dill, David L. (2000). Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 115–119.
- Graf, S. and Saidi, H. (1997). Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83.
- Harman, M., Binkley, D., and Danicic, S. (2003). Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64.
- Hatcliff, J., Dwyer, M. B., and Zheng, H. (2000). Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353.
- H.Cho, G.Hachtel, E.Macii, B.Pleisser, and F.Somenzi (1996). Algorithms for approximate fsm traversal based on state space decomposition. *IEEE TCAD*, 15(12):1465–1478.
- Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002). Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70.
- Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2003). Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag.
- Holzmann, G. J. and Smith, M. H. (2000). Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87.
- Korel, B. and Laski, J. (1988). Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163.
- Korel, B. and Laski, J. (1990). Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195.
- Kozen, D. (1998). Results on the propositional mu-calculus. In *Theoretical Computer Science*, volume 27, pages 333–354.
- Kurshan, R. P. (1990). Analysis of discrete event coordination. In *Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 414–453.
- Kurshan, R. P. (1994). *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press.
- Lafuente, Alberto Lluch (2002). Database of promela models. <http://www.informatik.uni-freiburg.de/~lafuente/models/models.html>.
- Lee, W., Pardo, A., Jang, J., Hachtel, G., and Somenzi, F. (1996). Tearing based automatic abstraction for ctl model checking. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 76–81.
- Lichtenstein, O. and Pnueli, A. (1985). Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107.
- Lind-Nielsen, J. and Andersen, H. (1999). Stepwise CTL model checking of state/event systems. In *CAV'99: Computer Aided Verification*.

- Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. (1995). Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44.
- Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.
- Millett, L. and Teitelbaum, T. (1998). Slicing promela and its applications to model checking.
- Milner, R. (1971). An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence.*, pages 481–489.
- Moundanos, Dinos, Abraham, Jacob A., and Hoskote, Yatin V. (1998). Abstraction techniques for validation coverage analysis and test generation. *IEEE Trans. Comput.*, 47(1):2–14.
- Namjoshi, K. S. and Kurshan, R. P. (2000). Syntactic program transformations for automatic abstraction. In *Computer Aided Verification*, pages 435–449.
- Ottenstein, K. J. and Ottenstein, L.M. (1984). The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184.
- Pardo, A. and Hachtel, G. (1998). Incremental CTL model checking using BDD subsetting. In *Proceedings of the Design Automation Conference*.
- Pardo, Abelardo (1997). *Automatic Abstraction Techniques for Formal Verification of Digital Systems*. PhD thesis, University of Colorado at Boulder.
- Park, D. (1981). Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag.
- P.Wolfer and V.Lovinfosse (1990). Verifying properties of large sets of processes with network invariants. 407:68–80.
- S. Graf and H. Saidi (1997). Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83.
- Sifakis, J. (1983). Property preserving homomorphisms of transition systems. *Lecture Notes in Computer Science*, 164:458–473.
- Tip, F. (1995). *Generation of Program Analysis Tools*. PhD thesis.
- Venkatesh, G. A. (1991). The semantic approach to program slicing. *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 26(6):107–119.
- Weiser, M. (1979). *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis.
- Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.