3

Simplifying Data Operations for Formal Verification

Felice Balarin
Cadence Berkeley Laboratories
Berkeley, CA, USA
felice@cadence.com

Khurram Sajid
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX, USA
sajid@tarski.ece.utexas.edu

Abstract

Arithmetic operations on integers are very expensive to represent in formalisms that most automatic formal verification tools use. To deal with this problem, a verification methodology is proposed where systems that include such operations are simplified before they are verified. The simplifications are classified either as abstractions (ensuring that the original system satisfies all the properties satisfied by the simplified system, but not vice versa), reductions (the simplified system preserves exactly all the properties of the original one), and restrictions (the simplified system has the same properties as the original, provided certain constraints on the state variables are satisfied). In the proposed methodology, systems are simplified automatically, as instructed by the designer. The simplifications preserve the structure of the system, and can be performed in time and space that depends only on the size of the simplified system.

Keywords

formal verification, abstraction, reduction

1 INTRODUCTION

Formal verification has received significant attention as a supplement to simulation and prototyping in system verification. It has been applied to a wide range of systems including those implemented in software, hardware or a combination of both. Ideally, formal verification enables fully automatic proofs of system properties under any input conditions. Practically, its application so far has been quite limited due to the complexity of

the associated computation. Computations in formal verification are often linear (or low polynomial) in the number of states of the system, but that number is often too large. This is a well know *state explosion* problem.

There are two main sources of state explosion. The first is concurrency. Systems typically consist of many components, and the number of states is exponential in the number of components. State explosion due to concurrency has long been the focus of a large research effort (Kurshan 1994, Shiple *et al.* 1992, Clarke *et al.* 1992, Balarin *et al.* 1993, Graf and Loiseaux 1993). The basic idea behind these approaches is to analyze the components separately, and combine them only if necessary, and possibly after some simplifications. Unfortunately, this approach is not well suited for another, less well studied, source of state explosion: data operations, because even the atomic data operations can be too hard to represent.

Typically, data values are kept in variables of some standardized data type, e.g. 16-bit integers. Enumerating all values in a system with only a few 16-bit integers cannot be accomplished in a lifetime, even with the fastest available computers. One approach that can partly alleviate the problem is representing the system and the sets of states symbolically. *Binary decision diagrams* (BDDs) have proved successful for this purpose (McMillan 1993), but they too have serious problems representing data operations. It is well known that any BDD representation of multiplication is exponential in the number of input bits. Even systems without multipliers may be hard to represent, because the size of a BDD depends strongly on the ordering of variables that has to be fixed for the whole system, and it may be hard or impossible to find an ordering which is good for every data operator in the system.

Because of these problems, the state-of-the-art approach is to begin formal verification with an already simplified description of the system, from which most of the data operations have been eliminated. The obvious problem of this approach is that all proofs are valid only for this simplified model and no firm claims can be made about the description of the system that is actually used in the design process.

In this paper we propose a method of building this initial simplified model automatically, based on the instructions provided by the user. We are careful to limit the method only to those simplifications that do not require (explicit or implicit) enumeration of values of actual variables. In fact, the complexity of our method depends only on the size of the simplified system, and not on the number of states of the original one.

Rather than make our method language specific, we have developed our method on a model of computation that is simple and yet allows easy mapping from many popular system description languages like Esterel, VHDL, or Verilog (actually, since we limit ourself to finite-state systems, this is true only of subsets of these languages, but similar limitations are imposed by other tools, notably synthesis tools, as well).

We propose three classes of simplifications: abstractions, reductions and restrictions. *Abstractions* remove the details of behavior conservatively. Every property proven for an abstracted system also holds for the detailed one. The inverse does not hold. If the property fails on the abstraction it might be that the detailed system satisfies it, but that a wrong abstraction (for that property) was chosen. There is no efficient automatic procedure for choosing the right abstraction for a given property. Therefore, we propose that the user selects an abstraction. Based on this information, the abstracted model is generated automatically in a provably conservative way.

Reductions are equivalence transformation. The reduced system is equivalent to the

original in the sense that every property of interest holds for the original system if and only if it holds for the reduced one. We propose to combine reductions with abstractions to maximize simplifications and minimize user input.

Restrictions are conditional equivalences. The restricted system is equivalent to the original one, but only if certain constraints on the system variables are met. Such a simplification can still be used to prove properties, which are conditioned on these constraints being satisfied. We propose that a list of such conditions (if any are used) be generated as a side-product of the simplification procedure. The use of such a list is the user's responsibility.

The rest of this paper is organized as follows. In Section 2 we survey the related work. We then introduce the model of computation in Section 3 and propose various simplifications in Section 4. Finally, we discuss our implementation in Section 5 and conclude with Section 6.

2 RELATED WORK

Many attempts have been made to use abstractions to enable verification of larger systems, requiring different level of user interaction. On one side are fully automatic approaches, which have only a limited success, either because they are applicable to a limited class of systems (Wolper 1986), or because they are often too expensive or not generating significant simplifications (Balarin *et al.* 1993, Shiple *et al.* 1992). On the other hand of the spectrum are approaches where the user specifies both the abstraction function and the simplified system, and the tool only checks that the simplified system indeed abstracts the original one with respect to the given abstraction function (Kurshan 1994).

Our approach shares the middle ground with Clarke *et al.* (1992) and Graf and Loiseaux (1993) where the user specifies only the abstraction function and the rest is done automatically. In both of these approaches, components of the concrete system are constructed, then abstracted and finally combined together, where combining them might involve parallel composition (Graf and Loiseaux 1993), or Boolean connectives and quantification (Clarke *et al.* 1992). Thus, both of these approaches are sensitive to complexity of components, and neither is well suited for those components that are hard to represent, like multiplications (since both approaches use binary decision diagrams for representation, multiplication would require exponential storage). In contrast, we propose an abstraction procedure which depends only on the size of the abstraction. In addition, both Clarke *et al.* (1992) and Graf and Loiseaux (1993) start from a description in some language and build an internal representation of the abstracted model inside a specific verification tool. In contrast, our simplified models are described in the same language as the detailed ones, which provides an additional option to develop interfaces to different verification tools.

Our work is a related to that of Cousot and Cousot (1977) where one *abstractly interprets* a program. The disadvantage of this method is that the user must provide an abstraction function by constructing the abstract interpreter, which is obviously not practical. To be fair, the focus of that work was not automatic verification, but rather providing a framework to describe various program analysis techniques like constant propagation and data-flow analysis.

All of the references mentioned above deal with abstractions. We are not aware of any work on using restrictions for formal verification, probably because restrictions attach

certain qualifications to the proved properties. We feel that proving qualified properties is still valuable if unqualified properties are too hard to prove. The closest related work is the concept of *partial evaluation* (Consel and Danvy 1993) in the compiler community. There, one specializes a program for a fixed input to improve its run-time performance. The restriction we propose can be seen as specializing the description of a system given a subset of possible values of a variable (rather than a single fixed value).

3 MODEL OF COMPUTATION

In this section we define our model of computation, illustrate it with an example, and argue for the choices of model features that we have made.

3.1 Syntax and semantics

We consider systems described by the following:

- a non-empty set of *operators* V,
- a function $i : V \mapsto V^*$ assigning to every operator $v \in V$ a (possibly empty) sequence of *inputs*,
- a function r assigning to each $v \in V$ a finite and non-empty set of consecutive integers called a *range*; we extend this definition to any sequence in V^* by:

  $$r(v_1 v_2 \cdots v_n) = r(v_1) \times r(v_2) \times \cdots r(v_n) \ ,$$

  we also use r_V to denote a range of a string in which every $v \in V$ appears exactly once (i.e. r_V is the space of all possible operator values),
- a function d assigning to each $v \in V$ a subset of $r(i(v)) \times r(v)$ called a *definition*. We sometimes interpret* $d(v)$ as a subset of r_V. The two interpretations are equivalent, and we use $d(v)$ to denote both. If necessary, the distinction is provided by the context.

We assume that every operator v is designated as one of the following:

- a *primary input*, which must have no inputs,
- a *delay operator*, which must have a single input with the same range as the operator itself.
- a *data operator*.

Intuitively, the system evolves from some initial state (the state of the system being the outputs of the delay operators) as follows:

1. Primary inputs take any value consistent with their definition.
2. As soon as their inputs are sets, data operators instantaneously compute their output as specified by their definition. If the definition contains more than one value for given inputs, then one of them is chosen non-deterministically.

*Given some space R, a set $S \subseteq R$ can be interpreted as a set $\{(x, y) \in R \times Q \mid x \in S\}$ in some larger space $R \times Q$.

3. The delay operators propagate the values from their inputs to their outputs with some delay.

A *graph* of the system has one node for each operator and an edge (w, v) whenever operator w is an input of v. To ensure that the evolution of the system is well defined, we will only consider systems whose graphs are such that every cycle contains at least one delay element.

This model is abstract enough to generalize many finite-state models that have been considered. For, example if all delay operators are restricted to change states at the same time, this model specializes to finite-state automata (Hopcroft and Ullman 1979). With some different restrictions, it can be specialized to different models of computation like asynchronous circuits, or co-design finite-state machines (Chiodo *et al.* 1993).

The precise delay semantic is immaterial here, because our focus is on abstracting data operations. Most of the languages have several such operators built-in. Similarly, we assume that some data operators can be implicitly defined by declaring them to be of one of the pre-determined types. In particular we assume the following:

- *Arithmetic operators* $+, -, *, \text{mod}$ are implicitly defined with their usual meaning restricted to the given range. The result is unpredictable if the correct result is out of range. Formally, we require that any operator v of type $\circ \in \{+, -, *, \text{mod}\}$ has two inputs and that:

  $$d(v) = \{(x, y, z) \,|\, x \circ y = z \text{ or } x \circ y \text{ not defined or } x \circ y \notin r(v)\} \ .$$

- *Comparison operators* $=, <, >, \leq, \geq, \neq$ must have two inputs and range $\{0, 1\}$, and they are defined to be 1 if and only if the comparison is satisfied.
- Any operator v of *assignment* type $:=$ is required to have a single input, and it is defined by:

  $$d(v) = \{(x, y) \,|\, y = x \text{ or } x \notin r(v)\} \ .$$

- Any operator v of *if-then-else* type must have three inputs, the range of first one must be $\{0, 1\}$, and it is defined by:

  $$d(v) = \{(0, x, y, z) \,|\, z = x \text{ or } x \notin r(v)\} \cup \{(1, x, y, z) \,|\, z = y \text{ or } y \notin r(v)\} \ .$$

Data operators that are not of any of these types are said to be of *enumerated* type and their definition must explicitly be provided by enumerating all elements.

3.2 Example

Consider the system shown in Figure 1. It is a simplified submodule of a shock absorber controller (Chiodo *et al.* 1996). It checks for parasitic signals coming from the wheel. It has two delay operators X and Y, two binary valued primary inputs CLOCK and SENSOR, and a binary valued output ERROR. Every operator in the system either has the range $\{0, 1\}$ (denoted by thin lines), or $0 \cdots 2^{16} - 1$ (denoted by thick lines). SENSOR pulses are coming from the wheel and should come once for every revolution of the wheel, but

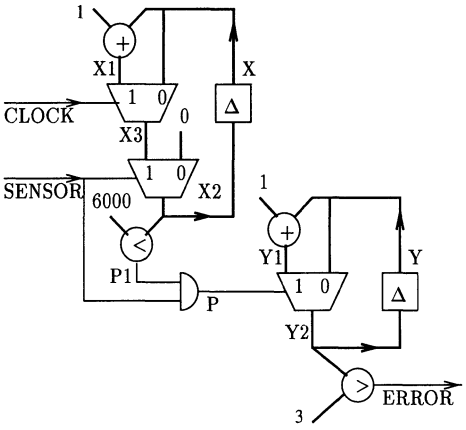

Figure 1 A module in a shock absorber controller.

occasionally parasitic pulses also occur. To check for them the system keeps in variable X the number of CLOCK pulses between two SENSOR pulses. If that number is less then 6000, that must have been a parasitic pulse because the wheel cannot possibly turn that quickly. The system counts the number of detected parasitic pulses (and keeps the count in variable Y), and emits an ERROR pulse if more than three are detected.

Formally, CLOCK and SENSOR are primary inputs defined by $d(\text{CLOCK}) = r(\text{CLOCK})$ and $d(\text{SENSOR}) = r(\text{SENSOR})$, i.e. they can take any value in their range, while 0, 1, 3 and 6000 are primary inputs defined by $d(0) = \{0\}$, $d(1) = \{1\}$, $d(3) = \{3\}$, and $d(6000) = \{6000\}$, i.e. these operators must keep a constant value (indicated by their name). The data operators in the system are:

- two comparison operators P1 and ERROR,
- two addition operators X1 and Y1,
- three if-then-else operators X2, X3 and Y2,
- a single enumerated operator P defined (as its symbol suggests) by:

$$d(\text{P}) = \{(0,0,0), (0,1,0), (1,0,0), (1,1,1)\} .$$

3.3 Model features

We have intentionally left out from our model several features that many system description languages have. We next argue for the choices we have made.

Finite-state Our model can describe only finite-state systems. The main reason is that any digital implementation (either pure HW or pure SW or mixed) is necessarily finite-state. Also, our goal is automatic formal verification, end efficient automatic tools exist only for finite-state systems. Many other tools operate only on finite-state subsets of HDL's. Rather then trying to define such a subset (which is often quite hard to do precisely), we have decided to restrict our model from the beginning.

Hierarchy and loops Contrary to the most languages, our model does not provide for hierarchical description. The main reason is that to our knowledge no automatic formal verification tools can exploit hierarchy to improve the efficiency. Thus, even though the lack of hierarchy will probably cause an increase in the description size, this is not likely to be the limiting factor overall. The size of the system that can be verified is most often limited by computations inside the verification tool, which are not affected by the hierarchy. Another reason is that the user may require different instantiations of the same module to be simplified differently, depending on the context. The easiest way to enable this is to let the user specify simplifications on a flattened model.

Similarly, we do not have loops in our model. Again, this is not a serious restriction because in most interesting cases (i.e. cases that can handled by synthesis and verification tools), they can be eliminated by unfolding, and the resulting growth of the description is not likely to be an overall bottleneck.

Finite precision, bitwise operation In our model, when the true result of some operation is out of range, the computed result is completely unpredictable. In practice, the computed result may vary according to encoding of integers and other implementation details. We did not want to include any particular encoding assumptions in our model. Instead, we leave the user an option of modeling the encoding effects explicitly (e.g. by defining the truncation operator as mod $2^{\#\text{ of bits}}$).

For the same reason, we have not included bitwise operators (operators like C's `|`, `&`, `<<`, `>>`) in our model. Their interpretation as integer operators is meaningful only under particular encoding. Thus, they are best defined and manipulated at the bit level (using operators of enumerated type).

4 ABSTRACTIONS, REDUCTIONS, RESTRICTIONS

4.1 Abstractions

The basic computation in all automatic formal verification tools involves representing and manipulating the *transition relation* (denoted by T) that relates primary inputs with inputs and outputs of delay elements. In our model the transition relation is specified implicitly by definitions of operators and can be computed by $T = \bigcap_{v \in V} d(v)$. Unfortunately, the transition relation is often too big to enumerate explicitly, or to represent with BDDs.

To avoid this problem, we seek to replace a system (V, i, r, d) with a system $(\widehat{V}, \widehat{i}, \widehat{r}, \widehat{d})$ which is a small abstraction of (V, i, r, d) that can be manipulated by verification tools. We restrict our intention to cases where $V = \widehat{V}$, and the abstraction is induced by an *abstraction function* $\phi : r_V \mapsto \widehat{r}_V$. We will assume that the user provides the abstraction function by providing its components $\phi_v : r(v) \mapsto \widehat{r}(v)$, and then we automatically construct an abstract system such that its transition relation $\widehat{T}$ satisfies:

$$\forall x \in r_V : \quad x \in T \implies \phi(x) \in \widehat{T} \ . \tag{1}$$

It is well known that if an abstract transition relation satisfies (1), then to prove many interesting properties of the original system it suffices to prove them for the abstract one.

Specifically, this is true for any property in the language containment framework (Kurshan 1994), and for any property expressible as a $\forall CTL^*$ formula (Clarke *et al.* 1992).

Given an abstraction function ϕ we build the abstract system by providing abstract operator definitions $\widehat{d}$ satisfying the following:

$$\widehat{d}(v) \;=\; \{\phi(x) \in \widehat{r}_V \,|\, x \in d(v)\} \;.$$

For enumerated data operators this is easily accomplished: the definition is traversed, replacing every element with its abstraction. However, since the definitions of arithmetic operators are implicit, other approaches are necessary. Our goal is to construct abstract operators in time proportional to their size, and not in time proportional to ranges of concrete operators. To achieve this goal, we restrict the class of abstraction functions that we consider. Every abstraction function ϕ_v induces a partition of $r(v)$ such that two values x and y are in the same class if and only if $\phi_v(x) = \phi_v(y)$. We require every such equivalence class to be a set of consecutive integers. Equivalently, we require the abstract range $\widehat{r}(v)$ to be a partition of $r(v)$ into sets of consecutive integers, and that $x \in \phi_v(x)$ holds for all $x \in r(v)$. This restriction allows us construct abstract definitions of data operators by performing the arithmetic operation only on extremal points $\min(\phi_v(x))$ and $\max(\phi_v(x))$. For example, if v is an addition operator with inputs u and w, the $\widehat{d}(v)$ contains all $(\widehat{x},\widehat{y},\widehat{z}) \in \widehat{r}(u) \times \widehat{r}(w) \times \widehat{r}(v)$ which satisfies at least one of the following:

$$\min(\widehat{x}) + \min(\widehat{y}) \leq \max(\widehat{z}) \quad \text{and} \quad \max(\widehat{x}) + \max(\widehat{y}) \geq \min(\widehat{z}) \;, \tag{2}$$

$$\min(\widehat{x}) + \min(\widehat{y}) < \min(r(v)) \quad \text{or} \quad \max(\widehat{x}) + \max(\widehat{y}) > \max(r(v)) \;. \tag{3}$$

It is not hard to show that if (2) is satisfied, then there exists $(x,y,z) \in \widehat{x} \times \widehat{y} \times \widehat{z}$ such that $x + y = z$. Similarly, if (3) is satisfied, then there exists $(x,y,z) \in \widehat{x} \times \widehat{y} \times \widehat{z}$ such that $x + y \notin r(v)$. It is also easy to see that checks (2) and (3) can be implemented by a single traversal of $\widehat{r}(u) \times \widehat{r}(w) \times \widehat{r}(v)$.

For example, if the following abstraction function is given for the addition operator X1 in Figure 1 and its inputs 1 and X:

$$\phi(x) = \begin{cases} \{0\} & \text{if } x = 0 \;, \\ \{1\} & \text{if } x = 1 \;, \\ \{2 \cdots 2^{16} - 1\} & \text{otherwise.} \end{cases} \tag{4}$$

then the abstract definition of X1 is (for conciseness, we write 0, 1 and 2+ instead of $\{0\}$, $\{1\}$ and $\{2 \cdots 2^{16} - 1\}$):

$$\begin{aligned} \widehat{d}(\mathrm{X1}) \;=\; \{\; & (0,\; 0,\; 0),\; (1,\; 0,\; 1),\; (2+,\; 0,\; 2+), \\ & (0,\; 1,\; 1),\; (1,\; 1,\; 2+),\; (2+,\; 1,\; 0), \\ & (0,\; 2+,\; 2+),\; (1,\; 2+,\; 0),\; (2+,\; 1,\; 1), \\ & \qquad\qquad\; (1,\; 2+,\; 1),\; (2+,\; 1,\; 2+), \\ & \qquad\qquad\; (1,\; 2+,\; 2+),\; (2+,\; 2+,\; 0), \\ & \qquad\qquad\qquad\qquad\; (2+,\; 2+,\; 1), \\ & \qquad\qquad\qquad\qquad\; (2+,\; 2+,\; 2+)\;\} \;. \end{aligned} \tag{5}$$

Conditions akin to (2) and (3) are readily defined for other arithmetic operators, except

for the mod operator. Even mod is straightforward if the right operator (e.g. y in x mod y) is restricted to be a constant. We adopt this restriction, and argue that it is not serious, because it is very rarely violated in practice.

4.2 Reductions

Often, abstractions enable additional optimizations. For example, assume that the abstraction function (4) has been specified for inputs 1 and X of the addition operator X1 in Figure 1 (but not for X1 itself). Based on this input abstraction, we can abstract the range of X1 into equivalence classes $\{0\}$, $\{1\}$, $\{2\}$ and $\{3 \cdots 2^{16} - 1\}$ *without any additional loss of information.* This is true because the input abstraction makes it impossible to distinguish between elements of a class.

In general, simplifications that do not entail any loss of information are called *reductions.* Given some operator v with input w (the extension to multiple outputs is straightforward) two values x and x' of v are equivalent and can be reduced to a single equivalence class if:

$$\forall y \in r(w) : \ (y, x) \in d(v) \text{ iff } (y, x') \in d(v) \ . \tag{6}$$

This condition describes how operator inputs can be used to reduce its outputs, but the converse is also true, some input can be reduced based on other inputs and the output. If an operator v has inputs u and w, two values x and x' of u are equivalent *with respect to* v if:

$$\forall y \in r(w),\ z \in r(v) : \ (x, y, z) \in d(v) \text{ iff } (x', y, z) \in d(v) \ . \tag{7}$$

Two values of some operator u are equivalent and can be reduced to a single equivalence class if they are equivalent with respect to every v of which u is an input.

The reduction rules (6) and (7) can easily be applied to enumerated operators in polynomial time by traversing their definition. However, for arithmetic operators, they can be applied efficiently only if relevant operators (w in (6), v and w in (7)) have already been abstracted to a small number of equivalence classes. In that case we can construct the new equivalence class by manipulating only the extremal points of existing ones.

Theoretically, one can search for possible reductions even on a model that is not abstracted. But this is likely to be very costly, and not likely to produce significant simplifications. Therefore, we propose to use reductions in a limited way, with a purpose of minimizing the user involvement as much as possible. We propose that the user specifies abstractions only of some operators. These abstractions are then *propagated* to abstract all the other operators. The rules of propagation are:

forward rule: an operator can be abstracted using (6) if all of its inputs have already been abstracted,

backward rule: an edge (u, v) can be abstracted using (7) abstracted if u is an input of v, and v and the rest of its inputs have already been abstracted,

fan-out rule an operator u can be abstracted if for every v of which u is an input, the edge (u, v) has already been abstracted.

If the user specifies abstraction functions for primary inputs and delay operators, then

all other operators can be abstracted using only the forward rule. But that is not necessarily the smallest set of abstraction functions that allow abstractions of all the operators. Unfortunately, no simple characterization of such a set is available presently.

4.3 Restrictions

Many properties of interest are of the form *if* P *then* Q, where P is some predicate on operators in the system. If the sole purpose of the simplification is proving such a property, then we can freely change operator definitions for those values that do not satisfy P. The typical P might be *"overflow does not occur"*. We can then eliminate from the operator definitions all cases that cause an overflow. This is likely to simplify the model. In addition, if we are convinced that our modifications have eliminated all the overflows, then we can simplify the property as well and proceed by trying to prove Q only. We stress that this *restriction* is property specific: the simplified model is valid only for properties of the form *if* P *then* Q.

Sometimes, P is not explicitly a part of the property, but it is an invariant of the system. For example, the system might be designed in a way that overflow never occurs. Then, we can also restrict the system to P, but again the simplified model is valid only if P is an invariant of the system, which is a claim that has to be proven (possibly using a different proof method or a different abstraction of the system).

One sort of invariants that is easy to prove is the case when the definition of an operator is not an onto mapping. In particular, integer constants are often represented as operators with no inputs, the same range as other integers in the system, and a single value in the definition. For example, since in Figure 1 the definition of input 1 of the addition operator X1 is $d(1) = \{1\}$, we can safely reduce the abstract definition (5) of X1 to:

$$\begin{array}{lll} \widehat{d}(\mathrm{X1}) = \{ & (1, & 0, & 1), \\ & (1, & 1, & 2+), \\ & (1, & 2+, & 0), \\ & (1, & 2+, & 1), \\ & (1, & 2+, & 2+) \ \} \ . \end{array} \tag{8}$$

To allow a limited sort of restriction, we allow the abstractions ϕ_v to be partial functions, i.e., we allow the user to define ϕ_v only on a sub-range of v. Then the abstract operators are constructed on these sub-ranges only, and all the other values are ignored. Restrictions can be propagated similarly to reduction. For example, if it assumed that inputs to an operator never take certain values, then it might be possible to deduce that the output can never take some value, and vice versa. If the restrictions are induced by integer constants, propagating them includes standard constant constant propagation, but it also includes simplifying operators if some of the operands are constant.

We also allow the user to postulate that certain operators never overflows. For example, such a restriction would further simplify the definition (8) of X1 to:

$$\begin{array}{lll} \widehat{d}(\mathrm{X1}) = \{ & (1, & 0, & 1), \\ & (1, & 1, & 2+), \\ & (1, & 2+, & 2+) \ \} \ . \end{array} \tag{9}$$

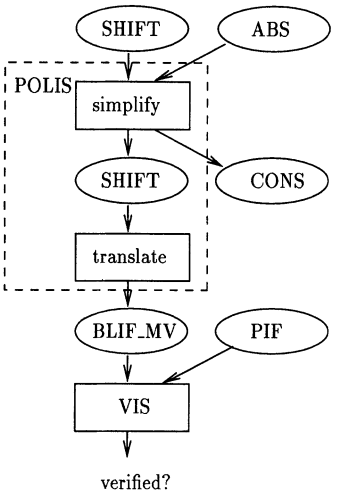

Figure 2 Formal verification flow.

To prove that X1 indeed does not overflow using this simplified model, it must be shown that input X never takes value 2+. Therefore, we say that X$\neq$ 2+ is the *weakest no-overflow pre-condition* for X1. Notice that even if X can take value 2+ this does not necessarily mean that an overflow can occur. Rather, it means that the current abstraction is not appropriate to show that there is no overflow.

5 IMPLEMENTATION

We have implemented the proposed procedure inside the HW/SW co-design system POLIS (Chiodo *et al.* 1994). POLIS allows analysis and design of systems described as a network of *co-design finite-state machines* (CFSMs), which are essentially the same as the model of computation presented in Section 3. CFSMs are specified in an intermediate format called SHIFT.

In our implementation, the overall verification flow is as shown in Figure 2, where data is represented by ovals and actions are represented by rectangles. We start from a SHIFT description of the systems, and based on the instructions in the ABS file, we build a simplified SHIFT description. The format of the ABS file is very simple: for every variable that is to be simplified the user provides end-points of intervals representing abstract values and restriction bounds, and a list of operators that do not overflow is given. This is not necessary for constants which are reduced automatically. Besides a simplified SHIFT file, the simplification also generates a CONS file which contains weakest no-overflow pre-conditions of the operators listed in the ABS file.

The next step is to build a BLIF_MV model of the system from the simplified SHIFT. BLIF_MV is the format used by the VIS formal verification tool (Brayton *et al.* 1996) to represent communicating finite state automata. This step amounts to providing an

automata semantics to CFSMs (Balarin *et al.* 1996). The BLIF_MV model is then analyzed by the VIS tool to determine whether it satisfies the properties specified in the PIF file.

We have used the described verification flow to verify the following property of the shock absorber module shown in Figure 1:

> If four SENSOR signals arrive between two CLOCK signals, then ERROR is generated.

This obviously covers only a small portion of the intended behavior, but this property can be proved on a very simplified system (Balarin *et al.* 1996), and such simple "sanity checks" often reveal interesting bugs. In Balarin *et al.* (1996) we have shown how simplifications of the kind described here dramatically improve verification times and memory usage (the BDD representation of the original model could not be built because after several hours of computation it exceed the memory limit of 480Mb, while the simplified model was verified in seconds). We are now able to generate the simplified model automatically in 4 seconds of CPU time, with an additional burden on the designer to specify a 3-line ABS file.

6 CONCLUSIONS

Arithmetic operations on integers pose a significant problem to automatic formal verification tools, because they are hard or even impossible to represent efficiently in formalisms that most of these tools use (BDDs or enumeration of all values). It is thus necessary to simplify systems that include such operations before they can be verified. We have proposed several automatic procedures to simplify such systems based on the user-specified instructions.

There are two main contributions that distinguish our approach from others. Firstly, our simplifications operate on a data-flow like representation and preserve the structure of the system. Thus, they can be considered syntactic transformations, and they are not limited to any particular verification tool. The simplified system is in a form that can easily be translated to any format used by popular automatic formal verification tools, including SMV (McMillan 1993), S/R (Kurshan 1994), and BLIF_MV (Brayton *et al.* 1996).

Secondly, we have chosen the simplification such that the time and space required to perform the simplification depends only on the size of the simplified system and not on the number of values the variables in the original system can take. In the future, we plan to extend our approach to other useful classes of simplifications that preserve this important property.

ACKNOWLEDGMENTS

We thank Luciano Lavagno for several illuminating discussions.

REFERENCES

Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language

containment. In Costas Courcoubetis, editor, *Proceedings of Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993*, pages 29–40. Springer-Verlag, 1993. LNCS vol. 697.

Felice Balarin, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. Formal verification of embedded systems based on CFSM networks. In *Proceedings of the 33th ACM/IEEE Design Automation Conference*, June 1996.

R.K. Brayton, A. Sangiovanni-Vincentelli, G.D. Hachtel, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R.K. Ranjan, T.R. Shiple, G. Swamy, T. Villa, A. Pardo, and S. Sarwary. VIS: A system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of Computer Aided Verification: 8th International Conference, CAV'96, Rutgers, NJ, July, 1996.* Springer-Verlag, 1996. LNCS vol. 1102.

Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993.

Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, August 1994.

M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, S. Yee, and A. Sangiovanni-Vincentelli. A case study in computer-aided codesign of embedded controllers. *Design Automation for Embedded Systems*, 1(1-2):51–67, January 1996.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proc. Principles of Programming Languages*, January 1992.

C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Ann. ACM Symp. on Principles of Prog. Lang.*, pages 493–501. 1993.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th Ann. ACM Symp. on Principles of Prog. Lang.* 1977.

S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In Costas Courcoubetis, editor, *Computer Aided Verification: 5th International Conference, CAV'93, Elounda, Greece, June/July 1993, Proceedings*, pages 71–84. Springer-Verlag, 1993. LNCS vol. 697.

J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.

Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

Thomas R. Shiple, Massimiliano Chiodo, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Automatic reduction in CTL compositional model checking. In *Proc. Fourth Workshop on Computer-Aided Verification*, pages 225–238, Montreal, June 1992. Also appeared in Lecture Notes in Computer Science, vol. 663.

P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th Ann. ACM Symp. on Principles of Prog. Lang.* January 1986.