Symbolic Exploration of Transition Hierarchies[*][†]

Rajeev Alur[‡] Thomas A. Henzinger[§] Sriram K. Rajamani[¶]

Abstract

In formal design verification, successful model checking is typically preceded by a laborious manual process of constructing design abstractions. We present a methodology for partially—and in some cases, fully—bypassing the abstraction process. For this purpose, we provide to the designer abstraction operators which, if used judiciously in the description of a design, structure the corresponding state space hierarchically. This structure can then be exploited by verification tools, and makes possible the automatic and exhaustive exploration of state spaces that would otherwise be out of scope for existing model checkers.

Specifically, we present the following contributions:

- A temporal abstraction operator that aggregates transitions and hides intermediate steps. Mathematically, our abstraction operator is a function that maps a flat transition system into a two-level hierarchy where each atomic upper-level transition expands into an entire lower-level transition system. For example, an arithmetic operation may expand into a sequence of bit operations.
- A BDD-based algorithm for the symbolic exploration of multi-level hierarchies of transition systems. The algorithm traverses a level-n transition by expanding the corresponding level-$(n-1)$ transition system on-the-fly. The level-n successors of a state are determined by computing a level-$(n-1)$ reach set, which is then immediately released from memory. In this fashion, we can exhaustively explore hierarchically structured state spaces whose flat counterparts cause memory overflows.
- We experimentally demonstrate the efficiency of our method with three examples—a multiplier, a cache coherence protocol, and a multiprocessor system. In the first two examples, we obtain significant improvements in run times and peak BDD sizes over traditional state-space search. The third example cannot be model checked at all using conventional methods (without manual abstractions), but can be analyzed fully automatically using transition hierarchies.

[*] A preliminary version of this paper appeared in the *Proceedings of the Fourth Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 98), *Lecture Notes in Computer Science* **1384**, Springer-Verlag, 1998, pp. 330–344.

[†] This research was supported in part by the Office of Naval Research Young Investigator award N00014-95-1-0520, by the National Science Foundation CAREER award CCR-9501708, by the National Science Foundation grant CCR-9504469, by the Air Force Office of Scientific Research contract F49620-93-1-0056, by the Army Research Office MURI grant DAAH-04-96-1-0341, by the Advanced Research Projects Agency grant NAG2-892, and by the Semiconductor Research Corporation contract 95-DC-324.036.

[‡] University of Pennsylvania and Bell Labs, alur@cis.upenn.edu

[§] University of California at Berkeley, tah@eecs.berkeley.edu

[¶] University of California at Berkeley, sriramr@eecs.berkeley.edu

1 Introduction

Formal design verification is a methodology for detecting logical errors in high-level designs. In formal design verification, the designer describes a system in a language with a mathematical semantics, and then the system description is analyzed against various correctness requirements. The paradigm is called *model checking* when the analysis is performed automatically by exhaustive state-space exploration. A variety of model checkers, such as COSPAN [HZR96], Murϕ [Dil96], SMV [CKSVG96], SPIN [HP96], and VIS [BSVH$^+$96] have been proven effective aids in the design of error-prone system components such as cache coherence protocols [CK96].

As we seek to enhance the applicability of model checking to complex designs, we are faced with the so-called *state-explosion* problem: the size of the state space grows exponentially with the size of the system description, making exhaustive state-space exploration infeasible. Consequently, to use the current tools effectively, one needs to focus on a critical system component. Since the behavior of an individual component typically depends on its interaction with other components, a component cannot be analyzed in isolation; rather, for a meaningful analysis, all relevant aspects of the surrounding system need to be identified. This process, called *abstraction*, is usually performed in an informal, manual fashion, and requires considerable expertise. Indeed, it is not uncommon that a successful verification or falsification run, using a few seconds of CPU time, depends on months of manual labor for constructing abstractions that are neither too coarse to invalidate the correctness requirements, nor too detailed to exhaust the tool capacities.

The goal of our research is to systematize and, whenever possible, automate the construction of useful abstractions. Our approach is to provide the designer, within the system description language, with operators for writing mental design abstractions into the system description. The judicious use of such operators is called *design for verifiability*, because it simplifies—and in some cases, eliminates—the process of "rediscovering" abstractions after the design is completed.

Our abstraction operators are motivated by the two main, orthogonal structuring principles for designs: (1) *spatial aggregation* together with hiding of spatial details, and (2) *temporal aggregation* together with hiding of temporal details. Type-(1) abstractions enable the design of components at different levels of spatial granularity: an ALU can be designed by aggregating gates, then used as an atomic block (after hiding internal gates and wires) in the design of a processor. Type-(2) abstractions enable the design of components at different levels of temporal granularity: an arithmetic operation can be designed by aggregating bit operations, then used as an atomic step (after hiding intermediate results) in the design of an instruction.

Spatial, type-(1) abstractions can be written into a system description using, for aggregation, the *parallel composition* of subsystems and, for hiding, the existential quantification of variables. According operators are provided by most system description languages. They are exploited heavily, both to facilitate the description of complex systems, and to obtain heuristics for coping with state explosion. For instance, in symbolic state-space exploration using BDDs, instead of building a single transition relation for the entire system, one typically maintains a set of transition relations, one for each component [TSL90].

By contrast, most system description languages do not provide operators for defining temporal, type-(2) abstractions. We have introduced such an operator, called **next**, and shown how it facilitates the description of complex systems, in a language called *Reactive Modules* [AH96]. In this paper, we show how the **next** operator can be exploited in symbolic state-space exploration to enhance the power of model checking.

Specifically, if M is a system description, and φ is a condition on the variables of M, then **next** φ **for** M describes the same system at a more abstract temporal level: a single transition of **next** φ **for** M aggregates as many transitions of M as are required to satisfy the condition φ, and hides the intermediate steps. For example, if M is a gate-level description of an ALU, and φ signals the completion of an arithmetic operation, then **next** φ **for** M is an operation-level description of the ALU. Mathematically, the semantics of **next** φ **for** M is defined as a two-level hierarchy of transition systems: each transition of the upper-level (e.g., operation-level) transition system abstracts an entire lower-level (e.g., gate-level) transition system. Then, by nesting **next** operators we obtain multi-level hierarchies of transition systems. The structuring of a state space into a multi-level transition hierarchy makes possible the exhaustive exploration of very large state spaces. This is because after the traversal of a level-n transition, the computed reach set for the corresponding level-$(n-1)$ transition system represents hidden intermediate steps and can be removed from memory.

In Section 2, we briefly review the language of Reactive Modules and give a simple example of a transition hierarchy. In Section 3, we introduce an algorithm for the symbolic exploration of transition hierarchies. In Section 4, we present experimental results that demonstrate the efficiency of our algorithm. For this purpose, we design a system comprising two processors with simple instruction sets, local caches, and shared memory. If we simply put together these components, using parallel composition but no **next** operator, the resulting flat transition system is far beyond the scope of existing model checkers. If, however, we use the **next** operator to aggregate and hide internal transitions between synchronization points before composing the various subsystems, the resulting transition hierarchy can be explored using the search routines of VIS, and correctness requirements can be checked fully automatically. Thus, the description of a design using **next** can eliminate the need for manual abstractions in verification.

Related work. The concept of temporal abstraction is inspired by the notion of *multiform time* in synchronous programming languages [BlGJ91, Hal93], and by the notion of *action refinement* in algebraic languages [AH89]. All of that work, however, concerns only the modeling of systems, and not automatic verification.

Temporal abstraction is implicitly present also in the concept of *stuttering* [Lam83]: a stuttering transition of a system is a transition that leaves all observable variables unchanged. Ignoring differences in the number of stuttering transitions leads to various notions of stutter-insensitive equivalences on state spaces (e.g., weak bisimulation). This suggests the following approach to model checking: for each component system, compute the appropriate stutter-insensitive equivalence, and before search, replace the component by the smaller quotient space. This approach, which has been implemented in tools such as the Concurrency Workbench [CPS93], requires the manipulation of the transition relations for individual components, and has not been shown competitive with simple search (cf. Section 3.1 vs. Section 3.2).

Partial-order methods avoid the exploration of unnecessary interleavings between the transitions of component systems. Gains due to partial-order reduction, in space and time, for verification have been reported both in the case of enumerative [HP94] and BDD-based approaches [ABH$^+$97]. By declaring sequences of transitions to be atomic, the **next** operator also reduces the number of interleavings between concurrent transitions. However, while partial-order reductions need to be "discovered" *a posteriori* from the system description, transition hierarchies are specified *a priori* by the designer, as integral part of the system description.

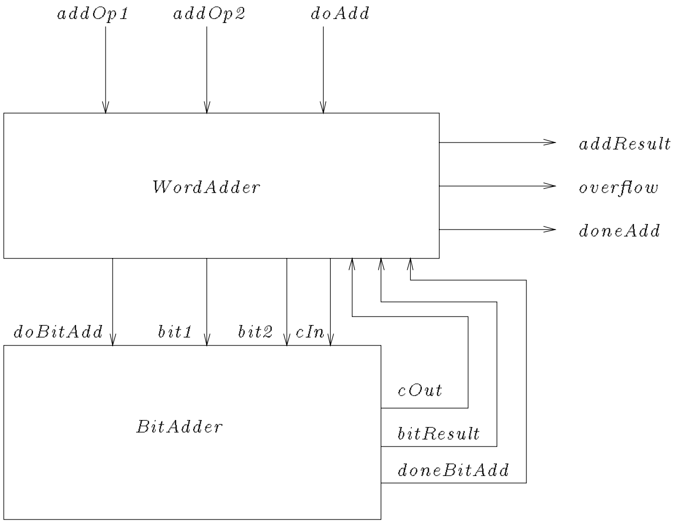

Figure 1: Word-adder and bit-adder

2 Example: From Bit Addition to Word Addition

2.1 Reactive Modules

We specify systems as Reactive Modules. A formal definition of reactive modules can be found in [AH96]; here we give only a brief introduction. The state of a reactive module is determined by the values of three kinds of variables: the *external variables* are updated by the environment and can be read by the module; the *interface variables* are updated by the module and can be read by the environment; the *private variables* are updated by the module and cannot be read by the environment (this distinction is similar to I/O automata [Lyn96]).

For example, Figure 2 shows a module that adds two words. The environment of the word-adder consists of two modules: a command module, which provides the operands to be added and an instruction that they be added, and a bit-adder, which is called repeatedly by the word-adder. Hence the word-adder has the external variables *addOp1* and *addOp2* of type WORD, which contain the two operands provided by the command module, the external variable *doAdd* of type BOOLEAN, which is set by the command module whenever the two operands should be added, and three output bits of the bit-adder: the sum *bitResult*, the carry-out *cOut*, and the flag *doneBitAdd*, which is set whenever a bit-addition is completed. The word-adder has the interface variables *addResult* of type WORD, which contains the sum, *overflow* of type BOOLEAN, which indicates addition overflow, *doneAdd* of type BOOLEAN, which is set whenever a word-addition is complete, and four input bits for the bit-adder: the operands *bit1* and *bit2*, the carry-in *cIn*, and the flag *doBitAdd*, which instructs the bit-adder to perform a bit-addition. The word-adder has the private variables *state* of type FLAGTYPE, which indicates if an addition is being executed, and *bitCount* of type LOGWORD, which tracks the position of the active bits during the addition of two words. We assume that, once a word-addition is requested, the command module keeps the variable *doAdd* true until the word-adder signals completion of the addition by setting *doneAdd* to true.

The state of a reactive module changes in a sequence of rounds. In the first round, the initial values of all interface and private variables are determined. In each subsequent round, the new values of all interface and private variables are determined, possibly dependent on some latched values of external, interface, and private variables from the previous round, and possibly dependent on some new values of external variables from the current round. No assumptions are made about the initial values of external variables, nor on how they are updated in subsequent rounds. However, in order to avoid cyclic dependencies between variables, it is not permitted that within a single round, a module updates an interface variable x dependent on the new value of an external variable y while the environment updates y dependent on the new value of x. This restriction is enforced by collecting variables into atoms that can be ordered linearly such that in each round, the variables within an atom can be updated simultaneously provided that all variables within earlier atoms have already been updated. Thus, a round consists of several subrounds—one per atom.

A round of the word-adder consists of four subrounds: first, the command module may provide new operands and issue an add instruction; second, the word-adder may initiate a bit-addition; third, the bit-adder may perform a bit-addition; fourth, the word-adder may record the result of a bit-addition and signal a completion of the word-addition. Accordingly, the interface and private variables of the word-adder are grouped into two atoms: *bit1*, *bit2*, *cIn*, *doBitAdd*, *state*, and *bitCount* are updated in the second subround of each round; *addResult*, *overflow*, and *doneAdd* in the fourth. The first and third subrounds of each round are taken by the command module and the bit-adder, respectively. The bit-adder, shown in Figure 3, needs one round for bit-addition, but can choose to wait indefinitely before servicing a request. A word-addition of two n-bit numbers, therefore, requires at least n rounds—one round for each bitwise addition. In the first of these rounds, the word-adder reacts to the command module, and the first bits may (or may not) be added. In the last of these rounds, the n-th bits are added, and the word-adder signals completion of the addition.

Figure 1 gives a block diagram of the word-adder composed with the bit-adder, and Figures 2 and 3 provide formal descriptions of both components. For each atom, the initial values of the variables and their new values in each subsequent (update) round are specified by guarded commands (as in UNITY [CM88]). In each round, unprimed symbols, such as x, refer to the latched value of the variable x, and primed symbols, such as x', refer to the new value. An atom *reads* the variable x if its guarded commands refer to the latched value of x. The atom *awaits* x if its guarded commands refer to the new value x'. The await dependencies between variables are required to be acyclic. A variable is *history-free* if it is not read by any atom. Since the values of history-free variables at any round do not influence the next round, they do not have to be stored during state-space traversal.

2.2 Flat vs. Hierarchical Models

We discuss two operations for building complex reactive modules from simple ones. The *parallel-composition operator* abstracts spatial complexities of a system by collecting the atoms of several modules within a single module. The *next operator* abstracts temporal complexities of a system by combining several rounds of a module as subrounds of a single round. Intuitively, if M is a reactive module and φ is a condition on the variables of M, then **next** φ **for** M is a module that, in a single round, iterates as many rounds of M as are necessary to make the so-called *aggregation predicate* φ true. Formal definitions of parallel composition and next-abstraction can be found in [AH96].

Consider, for example, the following two modules:

$$\begin{aligned}&\mathit{ConcreteAdder} = \mathit{WordAdder} \parallel \mathit{BitAdder}\\&\mathit{AbstractAdder} = \textbf{next}\ (\mathit{doAdd} \Rightarrow \mathit{doneAdd})\ \textbf{for}\ \mathit{ConcreteAdder}\end{aligned}$$

```
module $WordAdder$
type FLAGTYPE : {IDLE, WORKING};
interface $addResult$ : WORD;
        $bit1, bit2, cIn, doBitAdd, doneAdd, overflow$ : BOOLEAN
private $state$ : FLAGTYPE;
        $bitCount$ : LOGWORD
external $addOp1, addOp2$ : WORD;
        $doAdd, doneBitAdd, bitResult, cOut$ : BOOLEAN
atom controls $state, bitCount, doBitAdd, bit1, bit2, cIn$
    reads $doneBitAdd, bitResult, cOut$
    awaits $addOp1, addOp2, doAdd$
    init
        [] $true \rightarrow state' := \mathrm{IDLE}$
    update
        [] $(state = \mathrm{IDLE}) \wedge doAdd' \rightarrow$
            $state' := \mathrm{WORKING};\ bitCount' := false;\ doBitAdd' := true;$
            $bit1' := addOp1'[0];\ bit2' := addOp2'[0];\ cIn' := false$
        [] $(state = \mathrm{WORKING}) \wedge (bitCount < \mathrm{WORDLENGTH} - 1) \wedge doneBitAdd \rightarrow$
            $bit1' := addOp1'[bitCount + 1];\ bit2' := addOp2'[bitCount + 1];$
            $cIn' = cOut;\ bitCount' = bitCount + 1$
        [] $(state = \mathrm{WORKING}) \wedge (bitCount = \mathrm{WORDLENGTH} - 1) \wedge doneBitAdd \rightarrow$
            $doBitAdd' = false;\ state' := \mathrm{IDLE}$
endatom

atom controls $addResult, doneAdd, overflow$
    reads $bitCount$
    awaits $bitResult, doneBitAdd, cOut$
    init
        [] $true \rightarrow doneAdd' := false$
    update
        [] $(bitCount < \mathrm{WORDLENGTH} - 1) \wedge doneBitAdd' \rightarrow$
            $addResult'[bitCount] := bitResult'$
        [] $(bitCount = \mathrm{WORDLENGTH} - 1) \wedge doneBitAdd' \rightarrow$
            $addResult'[bitCount] := bitResult;\ doneAdd' := true;\ overflow' = cOut'$
endatom
endmodule
```

Figure 2: Word-adder

```
module BitAdder
interface doneBitAdd, bitResult, cOut : BOOLEAN
external bit1, bit2, cIn, doBitAdd : BOOLEAN
atom doneBitAdd, bitResult, cOut
    awaits bit1, bit2, cIn, doBitAdd
    init
        [] true → doneBitAdd' := false
    update
        [] doBitAdd' →
            bitResult' := bit1' ⊕ bit2' ⊕ cIn';
            cOut' := (bit1'&bit2')|(bit2'&cIn')|(cIn'&bit1');
            doneBitAdd' := true
        [] true → doneBitAdd' := false;
endatom
endmodule
```

Figure 3: Bit-adder

Each of the two modules is a model for an addition unit that consists of two components, a word-adder composed with a bit-adder. The two models differ only in their level of temporal granularity. In the flat model *ConcreteAdder*, the addition of two n-bit words takes at least n rounds. In the hierarchical model *AbstractAdder*, the addition of two n-bit words takes a single round. This is because the next-abstracted module combines into a single round as many rounds as are necessary to make either *doAdd* false or *doneAdd* true. In other words, in the flat model, bit-additions are atomic. Thus the flat model is adequate under the assumption that the addition unit is put into an environment that interacts with the addition unit only before and after bit-additions, but does not interrupt in the middle of a bit-addition. By contrast, in the hierarchical model, word-additions are atomic. Therefore the hierarchical model is adequate only under the stronger assumption that the addition unit is put into an environment that interacts with the addition unit only before and after word-additions, but does not interrupt in the middle of a word-addition. While the flat model is adequate in more situations, we will see that the hierarchical model can be verified more efficiently, and therefore should be preferred whenever it is adequate.

3 Symbolic Exploration of Hierarchical State Spaces

The reason why next-abstraction can be exploited by model checking can be seen as follows. If the semantics of a reactive module is viewed as a state-transition graph, with each round corresponding to a transition, then the hierarchical module **next** φ **for** M may have many fewer states than the flat module M. In particular, the **next** operator removes all states of M in which the aggregation predicate φ is not true; these states are called *transient.* When exploring the state space of the flat module, both transient and nontransient reachable states need to be stored as they are computed. By contrast, when exploring the state space of a hierarchical module, transient states need to be computed in order to find nontransient successor states, but once these states are found, the transient states need not be stored. Moreover, some of the variables that are not history-free in the flat model no longer influence (nontransient) successor states in the hierarchical model, and thus in effect become history-free. This results in further savings in memory for storing states.

The savings are particularly pronounced when hierarchical models are composed. Consider two flat models M and N, and two hierarchical models $M' = (\textbf{next}\ \varphi\ \textbf{for}\ M)$ and $N' = (\textbf{next}\ \psi\ \textbf{for}\ N)$. The hierarchical composition $M' \parallel N'$ is an adequate model for the composed systems if the aggregation predicates φ and ψ characterize the synchronization points of the two components; that is, if M interacts with N whenever φ becomes true, and N interacts with M whenever ψ becomes true. The possible interleavings of transitions of M and N may lead to an explosion of transient states of $M \parallel N$ (states in which either φ or ψ is false), which can be avoided by exploring instead $M' \parallel N'$. In other words, if the interaction between two component systems can be restricted, then some of the state-explosion problem may be avoided. Indeed, as we shall see, in complex systems with many components but well-defined interactions between the components, the computational savings, both in time and memory, can be enormous.

In the following, we first define the transition relations of composite and hierarchical modules from the transition relations of the components. Then we present a nested-search algorithm that explores the state space of a hierarchical module efficiently. The nested-search algorithm uses an implicit, algorithmic representation of the transition relation of a hierarchical module for image computation.

3.1 Explicit Definition of Transition Relations

The state-transition graph of a reactive module can be specified by a symbolic transition relation. Given a module M with variables X, the *symbolic transition relation* of M is a boolean function $T_M(X, X')$. Let $\hat{X}$ and $\hat{X}'$ be two states of M, i.e., valuations to the variables of M. Then the expression $T_M(\hat{X}, \hat{X}')$ evaluates to true iff there is an edge in the state-transition graph from state $\hat{X}$ to state $\hat{X}'$. All modules are built from individual atoms using parallel composition and the **next** operator. It is straightforward to construct the symbolic transition relation of an atom. For complex modules, the symbolic transition relation is defined inductively.

Parallel composition. Consider the module $M = M_1 \parallel M_2$. Let $T_{M_1}(X_1, X_1')$ and $T_{M_2}(X_2, X_2')$ be the symbolic transition relations of M_1 and M_2, respectively. Then the symbolic transition relation of M is given by the conjunction

$$T_M(X, X') = T_{M_1}(X_1, X_1') \wedge T_{M_2}(X_2, X_2').$$

Next abstraction. Consider the module $M = (\textbf{next}\ \varphi\ \textbf{for}\ N)$. Let $T_N(X, X')$ be the symbolic transition relation of N. For all natural numbers $i \geq 0$, define

- $T_M^0(X, X') = T_N(X, X')$;
- $T_M^{i+1}(X, X') = T_M^i(X, X') \vee (\exists Y)(\neg\varphi(Y) \wedge T_N(X, Y) \wedge T_M^i(Y, X'))$.

Let $T_M^l = T_M^{l+1}$ be the limit of the fixpoint computation sequence $T_M^0, T_M^1, T_M^2, \ldots$ (finite convergence is guaranteed for finite-state systems). Then the symbolic transition relation of M is given by

$$T_M(X, X') = \varphi(X) \wedge \varphi(X') \wedge T_M^l(X, X').$$

The reachable state set of a module can be computed by iterated application of the transition relation. For this purpose, it is theoretically possible to construct, using the above definitions, a BDD for the symbolic transition relation of a hierarchical module. In practice, however, during the construction the intermediate BDDs often blows up and results in memory overflow. For parallel composition, it is a common trick to leave the transition relation conjunctively decomposed and represent it as a set of BDDs, rather than computing their conjunction as a single BDD [TSL90]. Early quantification heuristics are then used to efficiently compute the image of a state set under a conjunctively partitioned transition relation. For next abstraction, we propose a similar approach.

3.2 Efficient Computation with Implicit Transition Relations

For model checking, it suffices to represent the symbolic transition relation of a module not explicitly, as a BDD, but implicitly, as an algorithm that given a state set, computes the set of successor states. This algorithm can then be iterated for reachability analysis and more general verification problems. Given a module M, the *single-step successor function* of M is a function R_M^1 from state sets of M to state sets of M. Let σ be a set of states of M. Then $R_M^1(\sigma)$ is the set of all states t of M such that there is a state $s \in \sigma$ with $T_M(s,t)$; that is, $R_M^1(\sigma)$ is the image of σ under the transition relation T_M. It is straightforward to compute $R_M^1(\sigma)$ for single atoms. For complex modules, the single-step successor function is computed recursively. In order to carry out this recursive compuation, we need for each module M, a function that computes the transition relation restriced to a set of states. Given a set of states σ, $T_M^1(\sigma)$ is a transtion relation of M restricted to the states in σ. Formally $T_M^1(\sigma) = \{(s,t) \mid (s,t) \in T_M \text{ and } s \in \sigma\}$.

Parallel composition. Consider the module $M = M_1 \parallel M_2$ and a set σ of states of M. Let $T_{M_1}^1(\sigma)$ and $T_{M_2}^1(\sigma)$ be the restricted transtion relations with respect to σ for M_1 and M_2, respectively. Then

$$T_M^1(\sigma) = T_{M_1}^1(\sigma) \wedge T_{M_2}^1(\sigma), \text{ and } R_M^1(\sigma) = \{t \mid \exists s.(s,t) \in T_M^1(\sigma)\}.$$

Next abstraction. Consider the module $M = (\textbf{next } \varphi \textbf{ for } N)$ and a set σ of states of M. Let $R_N^1(\sigma)$ be the image of σ for N, and let let $T_N^1(\sigma)$ be the transition relation of N restriced to σ. Then $R_M^1(\sigma)$ and $T_M^1(\sigma)$ are computed by the *nested-search* procedure described in Algorithm 3.2.

Algorithm 3.2

```
 {Given module $M = (\textbf{next } \varphi \textbf{ for } N)$, function to generate single-step successor $R_N^1$,
    function to generate restricted transiton relation $T_N^1$,
    and state set $\sigma$, compute $R_M^1(\sigma)$}
 {We will assume $\sigma \subseteq \varphi$}
 $FirstLevelImage := \{\}$
 $SecondLevelImage := R_N^1(\sigma)$
 $SecondLevelReachSet := \{\}$
 $SecondLevelTrans := \{(s,s) \mid s \in \sigma\}$
 $FirstLevelTrans := \{\}$
 $StepLevelTrans := T_N^1(\sigma)$
 loop
     $FirstLevelImage := FirstLevelImage \cup (SecondLevelImage \cap \varphi)$
     $SecondLevelReachSet := SecondLevelReachSet \cup (SecondLevelImage \cap \overline{\varphi})$
     $SecondLevelTrans := SecondLevelTrans \cup$
         $\{(s,t) \mid \exists s'.(s,s') \in SecondLevelTrans \wedge (s',t) \in StepLevelTrans \wedge t \notin \phi\}$
     $FirstLevelTrans := FirstLevelTrans \cup$
         $\{(s,t) \mid \exists s'.(s,s') \in SecondLevelTrans \wedge (s',t) \in StepLevelTrans \wedge t \in \phi\}$
     $SecondLevelImage := R_N^1(SecondLevelReachSet)$
     $StepLevelTrans := T_N^1(SecondLevelReachSet)$
 until ($SecondLevelReachSet$ does not change)
 $T_M^1(\sigma) := FirstLevelTrans$
 $R_M^1(\sigma) := FirstLevelImage$
 return($T_M^1(\sigma)$, $R_M^1(\sigma)$)
```

In contrast to a BDD for $T_M(X, X')$, which explicitly represents the transition relation of module M, the recursive algorithm for computing the function R_M^1 implicitly represents the same information.

In practice, a mixture of explicit symbolic representation of transition relations (for small modules) and implicit image computation (for complex modules) will be most efficient. We report on our experiences with nested search in the following section.

4 Experiments

The aim of our experiments is to investigate the efficiency of the proposed method for the automatic reachability analysis of complex designs. All experimental results reported in this paper were obtained by modeling the systems in Verilog and using the *vl2mv* Verilog compiler along with VIS [BSVH$^+$96]. We implemented a new command in VIS, called *abstract_reach*, based on Algorithm 3.2.

4.1 Multiplier

We model a word-multiplier that functions by repeated addition, using the word-adder described earlier. With help of the **next** operator, we can model the multiplier at various levels of temporal detail. We experiment with two options:

- Model 1: For addition, use the flat module *ConcreteAdder* explained in Section 2. In this model, bit-additions appear as atomic actions of the multiplier.
- Model 2: For addition, use the hierarchical module *AbstractAdder*. In this model, word-additions appear as atomic actions.

We perform reachability analysis with both models. Model 1 is given to VIS directly, and reachability analysis is performed using the *compute_reach* command of VIS. In order to analyze Model 2, we use the *abstract_reach* command with the aggregation predicate $doAdd \Rightarrow doneAdd$. As a result, the states in which *doAdd* is true and *doneAdd* is false become transient states.

We experiment with two 4-bit operands and an 8-bit result. In this case, Model 1 has 68 latches and 1416 gates. After the next abstraction, 24 of these latches become history-free; that is, their values in a state do not influence the next states of Model 2. In particular, the local variables of the adder become history-free, and hence, are represented by trivial functions in the BDD that represents the reachable states. Table 1 shows the peak BDD sizes for both models.

4.2 Cache Coherence Protocol

We describe the various components of a generic cache coherence protocol before discussing our results. Each cache block can be in one of three states: INVALID, READ_SHARED, or WRITE_EXCLUSIVE. Multiple processors can have the same memory location in their caches in the READ_SHARED state, but only one processor can have a given location in the WRITE_EXCLUSIVE state. There is a directory that, for each memory location, has a record of which processors have cached that location and what states (READ_SHARED, WRITE_EXCLUSIVE) these blocks are in. Due to want of space, we will not explain the protocol formally. An example scenario gives the general flavor. Suppose that Processor 1 has a location in WRITE_EXCLUSIVE, and Processor 2 wants to read this location. First Cache 2 records a write miss and communicates that to the directory. The directory then sends a message to Processor 1, requesting it to move the state of the block under consideration from WRITE_EXCLUSIVE to READ_SHARED. Cache 1 acknowledges this request and also sends the latest version of the data in this block to the directory. The directory then services Cache 2 with this data, and Cache 2 gets the block as READ_SHARED. Each

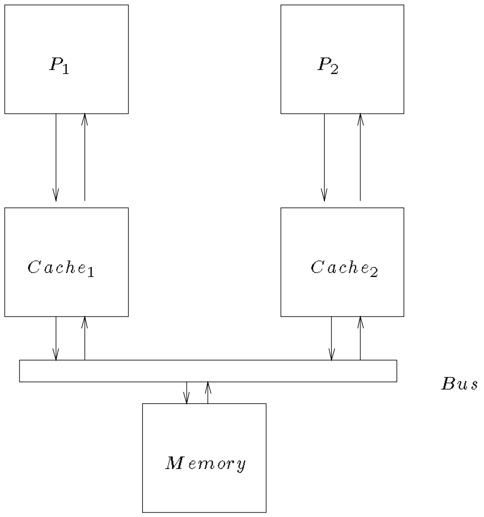

Figure 4: Processor-Memory System

of these steps involves a transaction on the bus, which could take an arbitrary number of rounds due to the asynchronous nature of the bus.

We experiment with two levels of temporal granularity. Model 1 is a flat model of the memory system, and Model 2 is a hierarchical model that abstracts temporal detail about the bus. While a bus transaction can consume multiple rounds in Model 1, it is forced to always complete in a single round in Model 2. For our experiments, we choose a 1-bit address bus and 1-bit data bus. In this case, Model 1 has 44 latches, of which 6 latches become history-free in Model 2. The peak BDD sizes during reachability analysis for both models are reported in Table 1.

4.3 Processor-Memory System

Aiming for a more dramatic improvement over flat modeling, we compose several systems whose interactions are limited. We put together two processors, each with an ALU consisting of the adder and multiplier described earlier, and the cache protocol, to obtain a complete processor-memory system. A block diagram of the system is shown in Figure 4. The processors have a simple instruction set: **load/store** register to/from memory, **add** two register operands, **multiply** two register operands, **compare** two registers, and a conditional **branch**. Again we experiment with two models. Model 1 is flat, and Model 2 is constructed by composing next-abstracted versions of the multipliers, adders, and bus protocol.

We choose an 1-bit wide address bus and a 2-bit wide data bus. In this case, Model 1 has 147 latches, of which 36 latches become history-free in Model 2 (15 latches in each multiplier, and 6 in the cache protocol). Here, reachability analysis for Model 1 is beyond the capability of current verification tools. However, fully automatic reachability analysis succeeds for Model 2, which structures the design using the **next** operator. Consider, for example, the situation where both processors start a multiplication at the same time. In Model 1, there are several transient states due to the interleaving of independent suboperations of the two multipliers. These transient states are entirely absent in Model 2. Indeed, nested search (Algorithm 3.2) is the key to verifying this example: we run out of memory when trying to compute an explicit representation of the

	Peak BDD size		Time for reachability (sec)	
Example	Model 1	Model 2	Model 1	Model 2
Multiplier	26979	6561	157	122
Cache Coherence	42467	21498	310	227
Proc/Mem System	space out	53103	*	816
with Program	space out	9251	*	153

Table 1: Experimental Results

transition relation for Model 2.[1]

4.4 Processor-Memory System with Programs

Finally, we add programs to the processors of the processor-memory system. Processor 1 computes $A*B$. Processor 2 computes $C*D$. Processor 1 then adds up both results and computes $A*B+C*D$. The two processors synchronize through a flag in memory. The last entry of the table shows the reachability results for the processor-memory system constrained by these programs. Again, Model 2 completes reachability using *abstract_reach*, whereas Model 1 does not. Thus we are able to verify invariants on Model 2, such as confirming that the results computed by the programs are correct.

5 Conclusions

We introduced a formal way of describing a design using both temporal and spatial abstraction operators for structuring the description. The temporal abstraction operator **next** induces a hierarchy of transitions on the state space, where a high-level transition corresponds to a sequence of low-level transitions. We exploited transition hierarchies in symbolic reachability analysis and presented an algorithm for proving invariants of reactive modules using hierarchical search. We tested the algorithm on arithmetic circuits, cache coherence protocols, and processor-memory systems, using an extension of VIS. The experimental results are encouraging, giving fully automatic results even on systems that are amenable to existing tools only after manual abstractions. Transition hierarchies can be exploited to give efficiencies in enumerative reachability analysis as well [AH96]. We are currently building a formal verification tool for reactive modules, called MOCHA, which will incorporate both symbolic and enumerative hierarchical search as primitives.

While the **next** operator is ideally suited for abstracting subsystems that interact with each other at predetermined synchronization points, it does not permit the "out-of-order execution" of low-level transitions. We currently investigate additional abstraction operators, such as operators that permit the temporal abstraction of pipelined designs.

References

[AH89] L. Aceto and M. Hennessy. Towards action refinement in process algebras. In *Proc. of LICS 89: Logic in Computer Science*, pages 138–145. IEEE Computer Society Press, 1989.

[1] The transition relation for the multiplier module alone, as computed by VIS, has 7586 BDD nodes and is composed of 6 conjunctive components. Even "and"-ing the components together results in memory overflow.

[ABH$^+$97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state-space exploration. In *Proc. of CAV 97: Computer-Aided Verification*, LNCS 1254, pages 340–351. Springer-Verlag, 1997.

[AH96] R. Alur and T.A. Henzinger. Reactive modules. In *Proc. of LICS 96: Logic in Computer Science*, pages 207–218. IEEE Computer Society Press, 1996.

[BlGJ91] A. Benveniste, P. le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[BSVH$^+$96] 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 *Proc. of CAV 96: Computer-Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.

[CK96] E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[CKSVG96] E.M. Clarke, K.L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Proc. of CAV 96: Computer-Aided Verification*, LNCS 1102, pages 419–422. Springer-Verlag, 1996.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[CPS93] R.J. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[Dil96] D.L. Dill. The $Mur\phi$ verification system. In *Proc. of CAV 96: Computer-Aided Verification*, LNCS 1102, pages 390–393. Springer-Verlag, 1996.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, 1993.

[HP94] G.J. Holzmann and D.A. Peled. An improvement in formal verification. In *Proc. of FORTE 94: Formal Description Techniques*, pages 197–211. Chapman & Hall, 1994.

[HP96] G.J. Holzmann and D.A. Peled. The state of SPIN. In *Proc. of CAV 96: Computer-Aided Verification*, LNCS 1102, pages 385–389. Springer-Verlag, 1996.

[HZR96] R.H. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In *Proc. of CAV 96: Computer-Aided Verification*, LNCS 1102, pages 423–427. Springer-Verlag, 1996.

[Lam83] L. Lamport. What good is temporal logic? In *Proc. of Information Processing 83: IFIP World Computer Congress*, pages 657–668. Elsevier Science Publishers, 1983.

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, 1996.

[TSL90] H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite-state machines using BDDs. In *Proc. of ICCAD 90: Computer-Aided Design*, pages 130–133. IEEE Computer Society Press, 1990.