Automatic Synthesis of Specifications from the Dynamic Observation of Reactive Programs

Bernard Boigelot[1]* and Patrice Godefroid[2]

[1] Université de Liège
Institut Montefiore, B28
B-4000 Liège Sart-Tilman, Belgium
boigelot@montefiore.ulg.ac.be

[2] Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566, U.S.A.
god@bell-labs.com

Abstract. VeriSoft [God97] is a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary C (or C++) code. VeriSoft can automatically detect coordination problems between the concurrent processes of a system. In this paper, we present a method to synthesize a finite-state machine that simulates all the sequences of visible operations of a given process that were observed during a state-space exploration performed by VeriSoft. The examination of this machine makes it possible to discover the dynamic behavior of the process in its environment and to understand how it contributes to the global behavior of the system.

1 Introduction

State-space exploration techniques are increasingly being used for analyzing the correctness of *concurrent reactive systems*. These techniques consist of exploring a directed graph, called the *state space*, representing the combined behavior of all concurrent components in a system. Existing state-space exploration tools can compute automatically a state space from a description of the concurrent system specified in a *modeling language*. Examples of such tools are CAESAR [FGM+92], COSPAN [HK90], CWB [CPS93], MURPHI [DDHY92], SMV [McM93], SPIN [Hol91], and VFSMvalid [FHS95], among others. These tools differ by the modeling languages they use for representing systems and properties, and by the conformation criteria according to which these representations are compared. But all of them are based on state-space exploration algorithms, in one form or another, for performing the verification itself. Some very complex

* "Aspirant" (Research Assistant) for the National Fund for Scientific Research (Belgium). The work of this author was done while visiting Bell Laboratories.

concurrent systems have been analyzed using state-space exploration techniques. In many cases, these techniques were able to reveal quite subtle design errors (e.g., [Rud92, CGH+93, BG96]).

Recently, it has been shown in [God97] how verification by state-space exploration can be extended to deal directly with "actual" descriptions of concurrent systems, e.g., *implementations* of communication protocols written in programming languages such as C or C++. This result was obtained by using a new search algorithm suitable for efficiently exploring the state spaces of such systems. This algorithm is used in *VeriSoft*, a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary C (or C++) code. VeriSoft can automatically detect coordination problems between the concurrent processes of a system. Specifically, VeriSoft searches the state space of the system for deadlocks, livelocks, divergences, and violations of user-specified assertions. An interactive graphical simulator/debugger is also available for following the execution of all the processes of the system. (See [God97] for details.)

In this paper, we argue that state-space exploration can give a deeper insight into the behavior of concurrent reactive systems than just checking specific formal properties. The state space of a system contains much information that can be used to better understand how the code is being exercised and how the different processes behave and interact with each other. However, extracting this information and presenting it to the user in a meaningful and convenient way is by no means a trivial task since state spaces of concurrent systems often contain millions of states and transitions.

To take up this challenge, we show in this paper how to automatically synthesize a specification, i.e., an abstract representation, for a reactive program from the observation of its executions. Precisely, we present a method to synthesize a finite-state machine that simulates all the sequences of visible operations of a process that were observed during a state-space exploration performed by VeriSoft. The examination of such a machine makes it possible to discover the dynamic behavior of the process in its environment and to understand how it contributes to the global behavior of the system.

In the next section, we define the state space of a concurrent system composed of processes executing arbitrary code written in a full-fledged programming language. In Section 3, we present an algorithm for synthesizing an abstract machine representing the observed behavior of a given process of the concurrent system being analyzed. The synthesis procedure includes a parameter that can be adjusted to obtain machines that represent the desired behavior with varying degrees of accuracy. We also describe an "on-the-fly" version of the algorithm for producing intermediate results while the state space of the system is still being explored. The synthesis algorithm has been implemented, and results of experiments are reported in Section 4. Several applications of this work are discussed in Section 5. The paper ends with a comparison of our approach with related work.

2 Systematic State-Space Exploration using VeriSoft

We consider a concurrent system composed of a finite set $\mathcal{P}$ of *processes* and a finite set of *communication objects*. Each process $P_i \in \mathcal{P}$ executes a sequence of *operations*, that is described in a sequential program written in a full-fledged programming language such as C or C++. Such programs are deterministic: every execution of the program on the same input data performs the same sequence of operations. We assume that processes communicate with each other by performing operations on communication objects. Examples of communication objects are shared variables, semaphores, and FIFO buffers. At any time, at most one operation can be performed on a given communication object (operations on a same communication object are mutually exclusive). Operations on communication objects are called *visible operations*, while other operations are called *invisible*. The execution of an operation is said to be *blocking* if it cannot be completed. We assume that only executions of visible operations may be blocking.

The concurrent system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. We assume that every process in the system always eventually attempts to execute a visible operation. This implies that initially, after the creation of all the processes of the system, the system may reach a first and unique global state s_0, called the *initial global state* of the system. We define a *transition* as a visible operation followed by a finite sequence of invisible operations performed by a single process. A transition whose visible operation is blocking in a global state s is said to be *disabled* in s. Otherwise, the transition is said to be *enabled* in s. A transition t that is enabled in a global state s can be *executed* from s. Once the execution of t from s is completed, the system reaches a global state s', called the *successor* of s by t. The *state space* of the concurrent system is composed of the global states that are reachable from the initial global state s_0, and of the transitions that are possible between these. All operations on objects are deterministic, except one special operation "VS_toss". This operation takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is visible and nondeterministic: the execution of a transition starting with VS_toss(n) may yield up to $n + 1$ different successor states, corresponding to different values returned by VS_toss.

VeriSoft [God97] is a tool for systematically exploring the state space of a concurrent system as defined above. In a nutshell, every process of the concurrent system to be analyzed is mapped to a UNIX process. The execution of the system processes is controlled by an external process, called the *scheduler*. This process observes the visible operations performed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition between two global states in the state space of the concurrent system. By reinitializing the system, the scheduler can explore alternative paths in the state space. The scheduler also contains an implementation of a new search algorithm that make it possible to systematically and efficiently explore the state spaces of such systems without storing any intermediate states in memory. For finite acyclic state spaces, this search algorithm is guaranteed to terminate and can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results. We refer the reader to [God97] for a detailed presentation of VeriSoft.

In what follows, the only fact we will need about VeriSoft is that it can generate a *labeled tree* T representing the state space of a concurrent system. Each node n of T corresponds to a global state of the system. Each edge $(n, (a, P_i), n')$ of T corresponds to a transition in the state space from global state n to global state n', and is labeled by its visible operation a and by the identifier P_i of the process executing the transition. The root node of T corresponds to the initial global state s_0 of the system. Every path in T corresponds to a sequence of visible operations that has been observed during the state-space exploration. If the state-space search terminates, this implies that the state space of the system is finite and acyclic, and the final tree T generated by VeriSoft contains all the sequences of visible operations that each individual process can perform in the concurrent system. Of course, if the state-space exploration is stopped before its completion, the final tree T obtained represents only the part of the state space that has been explored.

The following definitions and notations will be used in the following sections. A *finite-state machine*, or machine for short, is a tuple $M = (S, A, \Delta, s_0)$, where S is a finite set of states, A is an alphabet, $\Delta \subseteq S \times A \times S$ is a transition relation, and $s_0 \in S$ is the initial state. A finite word $w = a_0 a_1 \ldots a_{n-1}$ is accepted by a machine M if there is a sequence of states $\sigma = s_0 \ldots s_n$ such that s_0 is the initial state of M and $(s_i, a_i, s_{i+1}) \in \Delta$ for all $0 \leq i \leq n-1$. A (labeled) tree can be viewed as a machine where (1) there is exactly one node, called the root, which no transitions enters, (2) every node except the root has exactly one entering transition, and (3) there is a path from the root to each state.

We also recall the following definitions (e.g., [Mil89]).

Definition 1. A machine $M_1 = (S_1, A_1, \Delta_1, s_0^1)$ *simulates* a machine $M_2 = (S_2, A_2, \Delta_2, s_0^2)$ if there exists a binary relation $R \subseteq S_1 \times S_2$ that satisfies the two following conditions:

- $(s_0^1, s_0^2) \in R$;
- whenever $(s_1, s_2) \in R$ and $(s_2, a, s_2') \in \Delta_2$, there exists a s_1' such that $(s_1, a, s_1') \in \Delta_1$ and $(s_1', s_2') \in R$.

Definition 2. Two machines $M_1 = (S_1, A_1, \Delta_1, s_0^1)$ and $M_2 = (S_2, A_2, \Delta_2, s_0^2)$ are *strongly bisimilar* if there exists a binary relation $R \subseteq S_1 \times S_2$ that satisfies the three following conditions:

- $(s_0^1, s_0^2) \in R$;
- whenever $(s_1, s_2) \in R$ and $(s_1, a, s_1') \in \Delta_1$, there exists a s_2' such that $(s_2, a, s_2') \in \Delta_2$ and $(s_1', s_2') \in R$;
- whenever $(s_1, s_2) \in R$ and $(s_2, a, s_2') \in \Delta_2$, there exists a s_1' such that $(s_1, a, s_1') \in \Delta_1$ and $(s_1', s_2') \in R$.

3 Synthesis Algorithm

Given a tree T representing (possibly a part of) the state space of a concurrent system, the problem addressed here is to synthesize a finite-state machine M that simulates all the sequences of visible operations of process $P_i \in \mathcal{P}$ that were observed during the exploration of T.

Since T typically contains transitions performed by all the processes of the system, we first compute the projection of T on the set of operations executed by P_i. This is done by hiding in T all the edges $e = (n, (a, P_j), n')$ corresponding to operations performed by processes other than P_i: for every such edge e, the origin of all the edges outgoing from the destination node n' of e is replaced by n, and the edge e is then discarded. The implementation of the projection algorithm also ensures that the resulting tree is *deterministic*, i.e., that all edges from a node have different labels. Moreover, the successor edges of each node are sorted. Let $T|i$ denote the tree returned by the projection algorithm. We call $T|i$ a *projected tree.*

For synthesizing an abstract machine for process P_i from $T|i$, we use a variant of an algorithm described in [BF72] that generates a finite-state machine for computing a given function f. Specifically, this algorithm takes as input a finite set S of words on an alphabet A and a function $f : A^* \mapsto Y$ that maps words in A^* to values in set Y. The algorithm then generates a finite-state machine M whose states are labeled by values in Y and such that the execution of M on any word $w \in S$ leads to a state labeled by $f(w)$.

In this section, we extend the procedure of [BF72] from words to trees, and adapt it to make it suitable for solving the problem addressed here. The basic idea of the modified algorithm is to define an equivalence relation between the nodes of the projected tree $T|i$, and to associate one state of the output finite-state machine to each equivalence class. Then, for every pair of nodes connected by an edge in the projected tree, a transition with the same label is added in the synthesized machine to connect the two states corresponding to the equivalence classes of these nodes. The synthesis procedure includes a parameter that can be adjusted to obtain machines that represent the desired behavior with varying degrees of accuracy.

Precisely, we proceed as follows. Let k be a positive integer. For each node n of the projected tree $T|i$, let $subtree(n, k)$ denote the subtree of $T|i$ that has n as its root and that contains all the successor edges and nodes of n up to depth k. This implies that all the paths in $subtree(n, k)$ contain at most k edges.

Definition 3. Two nodes n and n' of the projected tree $T|i$ are said to be k-*equivalent* if $subtree(n, k)$ and $subtree(n', k)$ are strongly bisimilar.

Since $T|i$ is deterministic, all subtrees of $T|i$ are also deterministic. Therefore, since the successor edges of each node in $T|i$ are sorted by the projection algorithm, checking whether $subtree(n, k)$ and $subtree(n', k)$ are strongly bisimilar can be done in time linearly proportional to the size of the smallest of both subtrees. Let $[n]_k$ denote the set of nodes of $T|i$ that are k-equivalent to n.

We now define formally the synthesized machine M_k.

Definition 4. Given a projected tree $T|i = (S, A, \Delta, s_0)$ and an integer $k > 0$, the nondeterministic abstract machine $M_k = (S_k, A_k, \Delta_k, s_0^k)$ is defined by

- $S_k = \{[n]_k | n \in S\}$,
- $A_k = A$,
- $\Delta_k \subseteq S_k \times A_k \times S_k$ is such that

$$([n]_k, a, [n']_k) \in \Delta_k \text{ iff } \exists (n, (a, P_i), n') \in \Delta,$$

- $s_0^k = [s_0]_k$.

This construction groups together the nodes of the projected tree $T|i$ that are k-equivalent. If subtrees corresponding to nodes of M_k that have already been generated are stored in a hash table, and if we assume that it takes $\mathcal{O}(1)$ time to access any of these trees, the overall worst-case time complexity of the above procedure is $\mathcal{O}(NB^k)$ where N is the number of nodes in $T|i$ and B is the maximum number of successor edges of a node in $T|i$.

We have the following.

Theorem 5. *Let $T|i = (S, A, \Delta, s_0)$ be a projected tree, let k be a positive integer, and let $M_k = (S_k, A_k, \Delta_k, s_0^k)$ be the corresponding abstract machine as defined in Definition 4. Then, M_k simulates $T|i$.*

Proof. Consider the relation $R \subseteq S_k \times S$ defined by $R = \{([s]_k, s) | s \in S\}$. Let us show that R is a relation satisfying the two conditions of Definition 1.

Since $[s_0]_k = s_0^k$, the first condition of Definition 1 is satisfied. Moreover, we know by Definition 4 that $\forall s \in S : \forall (s, (a, P_i), s') \in \Delta : ([s]_k, a, [s']_k) \in \Delta_k$. This implies that, for all $([s]_k, s) \in R$ and $(s, (a, P_i), s') \in \Delta$, we have $([s]_k, a, [s']_k) \in \Delta_k$ and $([s']_k, s') \in R$. Consequently, the second condition of Definition 1 is also satisfied, and M_k simulates $T|i$.

The following corollary is immediate.

Corollary 6. *Let $L(T|i)$ denote the language accepted by the projected tree $T|i$, and let $L(M_k)$ be the language accepted by the abstract machine M_k as defined in Definition 4. Then,*

$$L(T|i) \subseteq L(M_k).$$

The previous theorem formalizes the notion of "approximation" provided by M_k with respect to $T|i$. The level of approximation is determined by the parameter k. If k is small, the procedure may group together many different nodes of $T|i$, and hence may generate a very compact machine. Conversely, if k is greater or equal to depth$(T|i)$, the length of the longest path in $T|i$, no approximation is made: the resulting machine M_k and $T|i$ are strongly bisimilar and accept the same language.

The previous observation implies that, for every $T|i$, there exists a k such that $L(T|i) = L(M_k)$. More interestingly, it also implies that, for every $T|i$, there exists a k such that

$$L(T|i) = \{w \in L(M_k) : |w| \leq \text{depth}(T|i)\}. \qquad (1)$$

This property holds when, not only all the sequences of $T|i$ are represented in M_k, but also all the sequences of length smaller or equal to depth($T|i$) accepted by M_k correspond to sequences of operations contained in $T|i$: the approximation performed by the synthesis algorithm is then *exact* for sequences of operations whose length is limited to depth($T|i$). Given a projected tree $T|i$, it is possible to compute the smallest value of k that satisfies Condition (1) above. This value can be much smaller than the smallest value of k satisfying $L(T|i) = L(M_k)$, as we will see in Section 4.

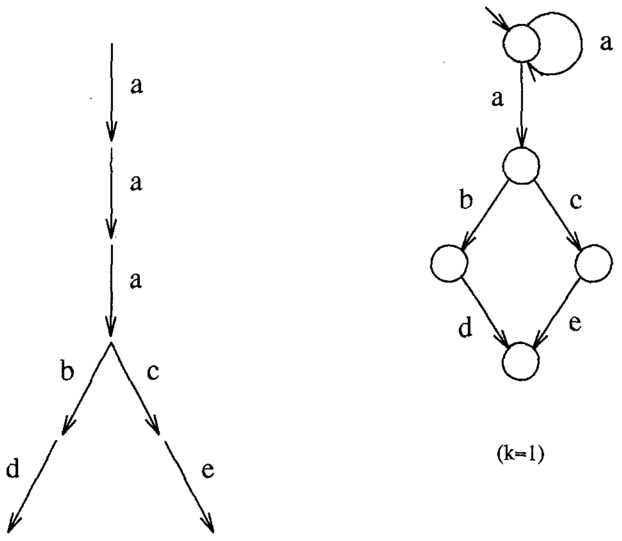

Fig. 1. Example of projected tree (left) and synthesized machine with $k = 1$ (right)

Example 1. Consider the projected tree on the left of Figure 1. The machine on the right of the figure is the abstract machine generated by the above procedure with $k = 1$. Nodes of the abstract machine correspond to nodes of the projected tree that have the same k-subtree. For instance, the initial state of the machine is the equivalence class of states that have only one transition labeled by a as successor. Because the abstract machine contains a cycle from the initial state in the abstract machine, the language of the projected tree is not equal to the language of the machine. The reader can check that the minimum value of k such that $L(T|i) = L(M_k)$ for this example is 3. The minimum value of k satisfying Condition (1) is 3 as well.

It is worth noticing that it is possible to generate parts of the machine M_k while the state space of the system is still being explored. This is useful for providing feedback to the user before completion of the search. Precisely, this can be done as follows. Let a node n of the projected tree $T|i$ be called *complete* once *subtree*(n, k) is completely known, i.e., when all the paths from n in $T|i$ contain at least k transitions or are known to be complete (because all the corresponding executions of the system are finite and have been completely explored). Whenever a complete node n is available in the projected tree, it can be passed to the synthesis algorithm, which can then test whether subtree(n, k) has already been visited; if this is not the case, a new state $[n]_k$ and new transitions can immediately be generated in M_k. By extension, such a state $[n]_k$ in M_k will also be called complete.

However, there are examples of concurrent systems for which this on-the-fly version of the synthesis algorithm is not helpful because no complete nodes are generated before the search ends. For instance, consider two processes P_1 and P_2 that can repeatedly perform a wait operation, enter a critical section, and then perform a signal operation. Assuming the value of the semaphore is initially 1, there is an execution of the system where P_2 loops forever while P_1 does not move, although P_1 is able to proceed eventually often. Because of the existence of this scenario, the root node of the projected tree $T|1$ will never be complete: there exists an execution of the system where the execution of the first operation of P_1 is continually postponed, preventing the k-subtree of the root node to be completely defined. This pathological case shows that the on-the-fly variant we have just described is mainly useful for concurrent systems without loosely-coupled processes.

4 Example

The synthesis algorithm described in the previous section has been implemented to be used in conjunction with VeriSoft. It has been tested on several implementations of concurrent systems. In this section, we present in detail the results obtained for one of them, a 2500-line concurrent C program controlling robots operating in an unpredictable environment. More precisely, this program represents a concurrent system composed of six processes that communicate via shared memory and semaphores. Two of the processes control robots that collect objects randomly dropped on a table by a third robot, represented by a third process. The three other processes are used to simulate the rest of the environment of the robots.

After exploring the state space of this system for a few minutes, VeriSoft reported a scenario composed of 29 transitions (as defined in Section 2) that led to a divergence. A divergence occurs when a process does not attempt to execute any visible operation for more than a given (user-specified) amount of time. After replaying this scenario at the C level using the VeriSoft simulator, it was easy to see that the problem was caused by an error in a "while" loop in the C code for one of the processes, and to understand under which circumstances the execution of that process was trapped inside the loop. This error was then corrected, and VeriSoft was used again to test whether the modification solved the problem without introducing new errors.

When the depth of the search is limited to 100 transitions, the tree representing the state space explored by VeriSoft contains about 380000 transitions, and can be completely explored in about 4 hours on a SparcStation 20. The tree can be saved in a file of about 12 Megabytes. This tree was used as input for our synthesis algorithm in the following experiments. All the abstract machines reported in what follows were generated in a few minutes of computation.

The finite-state machines synthesized by the algorithm of Section 3 with $k = 1$ for the processes 1, 2, 4 and 6 are shown in Figure 2. These processes synchronize with each other by executing the visible operations *semsignal* and *semwait* on semaphores that are identified by the first argument of the operation. The second argument specifies the value to be added (resp. subtracted) to the value of the corresponding semaphore after the execution of semsignal (resp. semwait). For all these processes, the minimum value of k satisfying Condition (1) is 1. Incomplete states are not shown. Increasing the value of k has little or no effect on the produced machines for these processes, as long as k is sufficiently smaller than the depth of the projected tree. When k reaches this threshold, the cycles in the graphs are unfolded and become sequences. The machines obtained for processes 1 and 6, which control the two robots collecting objects on the table, are identical. The machine synthesized for process 3 does not contain any transitions.

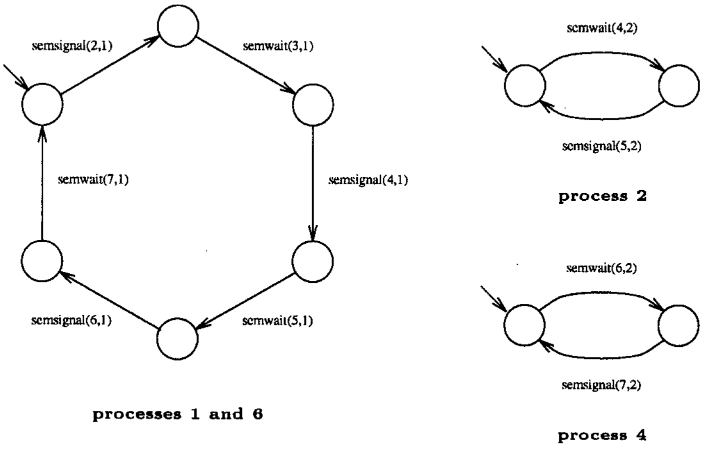

Fig. 2. Abstract machines for processes 1, 2, 4 and 6

The abstract machines generated for process 5 with $k = 1$ and $k = 2$ are shown in Figure 3. Process 5 is the process that periodically drops new objects on the table. It uses the visible operation VS_toss to randomly select locations on the table for placing new objects. When the selected location is already occupied by another object, the process attempts to find another location that

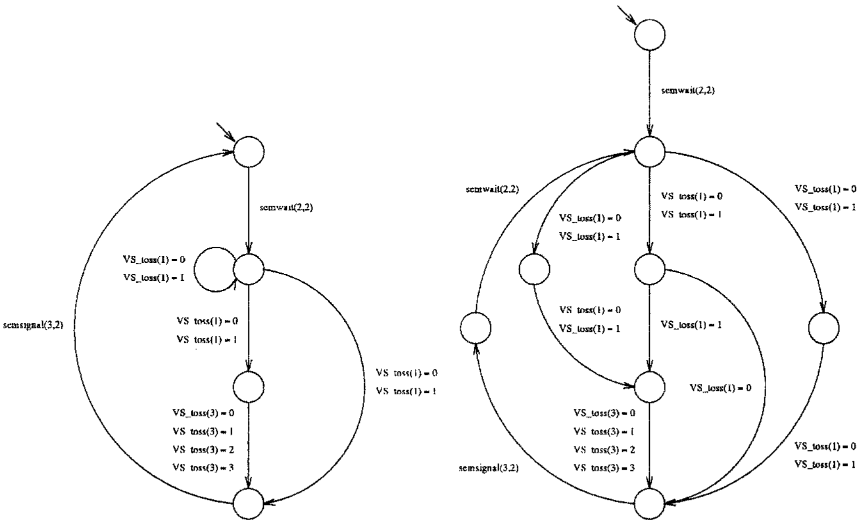

Fig. 3. Abstract machines for process 5 with $k = 1$ (left) and $k = 2$ (right)

is available (this procedure also involves calls to VS_toss). The minimum value of k satisfying Condition (1) is 2. Indeed, chosing $k = 1$ causes the synthesis algorithm to consider the two successive occurrences of a same operation as executions of the same cycle of M_1 (cf. second state of M_1). This cycle generates sequences of operations that are not represented in $T|i$. Chosing $k = 2$ yields the optimal machine that generates only sequences of $T|i$ (see Section 3). Greater values of k makes the synthesis algorithm generate less compact machines.

5 Applications

Much information about the behavior of a system can be obtained from the examination of the abstract machines generated by the synthesis algorithm.

Information about the *test coverage* of the search performed by VeriSoft can be obtained from the abstract machines since they contain the visible operations have been exercised during the search. For instance, the finite-state machine synthesized for process 3 in the example of the previous section does not contain any transitions. This means that this process was never able to execute a visible operation during the scenarios represented in the explored part of the state space.

Since the synthesized machines represent partial descriptions of the individual processes of the system, they make it possible to *discover properties* of the behavior of these processes without formally specifying any property. Examining these machines can help in identifying suspicious and erroneous behaviors. This is also useful for *selecting scenarios* for testing purposes. For instance, unexpected behaviors in an abstract machine can help in designing test cases to exhibit these behaviors. These scenarios can then be executed and examined in detail at the implementation level with an interactive simulator.

The synthesized abstract machines can also provide valuable information about the *overall communication and synchronization structure* of the concurrent system. For the example of the previous section, one can see from the synthesized machines that the coupling between the different processes is very tight: processes 1 and 6 enforce a strict synchronization ordering between processes 2, 4 and 5. The amount of parallelism in the system is very limited. This also reveals a potential weakness in the design of the synchronization structure of this system: a failure (death) of one process should quickly block all the other processes of the system.

The synthesis algorithm provides information on the *regularity* of the state space of the system. Indeed, the synthesis algorithm detects recurrent patterns of operations in the observed (finite) behaviors, and groups them in the generated abstract machines. Extrapolating repetitive behaviors can help predicting the (very long or even infinite) behaviors exhibited in the unexplored parts of the state space.

Finally note that our synthesis algorithm can be a very effective way to present a huge amount of data (e.g., 12 Megabytes of data) on a complex concurrent program (2500 lines of C code spread over 12 files) in a very compact form (a few tens of states and transitions) that can easily be examined by the user. When the generated abstract machines are too large to be examined, the user has the possibility to compute more abstract machines by modifying the labels corresponding to visible operations. For instance, labels of operations that contain values of parameters (e.g., a message being sent or received) can be simplified by masking out the values of some of these parameters from the label name. This reduces the number of possible labels for the transitions of the abstract machine, and hence the size of the machine.

6 Conclusions and Comparison with Related Work

We have presented a technique for automatically synthesizing a finite-state machine that simulates all the sequences of visible operations of a given process (executing arbitrary C or C++ code) that were observed during a state-space exploration performed by VeriSoft. The level of abstraction is determined by the set of labels of the transitions of the abstract machine, while the level of approximation can be adjusted by modifying the value of the parameter k of the synthesis procedure. This technique makes it possible to discover the behavior of processes for which the code is unknown or unavailable, or to visually detect anomalies in the dynamic behavior of processes in their environment.

Our synthesis algorithm can generate very compact and faithful finite-state machines from a huge amount of data. For the example considered in Section 4, it synthesized a handful of small finite-state machines satisfying Condition (1) from a state-space tree of about 380000 transitions. It is worth emphasizing that our technique is effective because it is used in conjunction with a tool for *systematically* exploring the state space of a concurrent system. If the synthesis algorithm was used in conjunction with traditional testing and debugging tools for distributed and parallel programs (e.g., see [CMN91, NM92, SS94]), the synthesized machines would likely be much less compact. Indeed, since these tools explore random paths in the state space, a same local state of a process might then be associated with different k-subtrees each time it is visited, and hence be represented by several states (equivalence classes) in the synthesized machine.

This work also proposes an original approach to reverse engineering [CC90]. Indeed, traditional reverse engineering methods and tools are based on static analysis techniques for extracting information about the structure of complex programs (e.g., see [WNC95]). In contrast, our approach does not rely on any specific assumption about the static structure of the programs used to represent the behavior of the processes, which can actually be written in any language. Moreover, it is also applicable to processes for which no code is available. Finally, it makes possible a much closer examination of the behavior of a process since it is based on the dynamic observation of its executions.

Other approaches to the finite-state machine synthesis problem have been proposed. Statistical methods using neural networks [DM94] are based on probabilities calculated from observations of the input language. These methods are very robust with respect to "input noise", i.e., when the observation of the input language may not be entirely reliable, but are much less efficient and difficult to use. Statistical methods can also be combined with algorithmic techniques into a "hybrid" method [MQ88] based on Markov models. This method has no advantages with respect to the synthesis algorithm we used since there is no input noise in the problem addressed here. Moreover, this hybrid method is not always able to produce a machine accepting exactly the input language when it exists. A detailed comparison of these different methods can be found in [CW95], where synthesis algorithms are used to generate a structured representation of the development process of a software-production organization from events recorded during the various tasks performed in the organization.

Acknowledgments

We wish to thank Glenn Bruns for helpful comments on this paper.

References

[BF72] A.W. Biermann and J.A. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, June 1972.

[BG96] B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the ACCESS.bus protocol using SPIN. In *Proceedings of Formal Methods Europe'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 465–478, Oxford, March 1996. Springer-Verlag.

[CC90] E. H. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[CGH$^+$93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Apllications.* North-Holland, 1993.

[CMN91] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, pages 491–530, October 1991.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[CW95] J. E. Cook and A. L. Wolf. Automatic Process Discovery through Event-Data Analysis. In *Proceedings of the 17th Conference on Software Engineering*, Seatle, April 1995.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, Cambridge, MA, October 1992. IEEE Computer Society.

[DM94] S. Das and M. C. Mozer. A Unified Gradient-Descent/Clustering Architecture for Finite-State Machine Induction. *Advances in Neural Information Processing Systems*, 6:19–26, 1994.

[FGM$^+$92] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. of the 14th International Conference on Software Engineering ICSE'14*, Melbourne, Australia, May 1992. ACM.

[FHS95] A. R. Flora-Holmquist and M. Staskauskas. Formal validation of virtual finite state machines. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, pages 122–129, Boca Raton, April 1995.

[God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 1990.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MQ88] L. Miclet and J. Quinqueton. Learning from Examples in Sequences and Grammatical Inference. In *Syntactic and Structural Pattern Recognition*, volume 45 of *NATO ASI Series F – Computer and Systems Science*, pages 153–171. Springer-Verlag, 1988.

[NM92] R. H. B. Netzer and B. P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proceedings of Supercomputing'92*, pages 502–511, Minneapolis, 1992.

[Rud92] H. Rudin. Protocol development success stories: Part I. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.

[SS94] R. S. Side and G. C. Shoja. A debugger for distributed programs. *Software Practice and Experience*, 24(5):507–525, May 1994.

[WNC95] L. Wills, Ph. Newcomb, and E. Chikofsky, editors. *Proceedings of the Second Working Conference on Reverse Engineering*, Toronto, July 1995. IEEE.