Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 240 (2009) 97–112
www.elsevier.com/locate/entcs

Model Checking Merged Program Traces

Paulo Salem da Silva[a,1,2] and Ana C. V. de Melo[a,1,3]

[a] *Department of Computer Science*
University of São Paulo
São Paulo, Brazil

Abstract

During a program's execution, state information can be collected and stored in the form of program traces. With such traces, one can analyze dynamic properties of the program. In this paper, we consider the problem of merging multiple traces from the same program in order to compose an approximate temporal model of its behavior. With such a model one can perform model checking based on both linear- and branching-time logics. To this end, we formally define what we mean by program trace and present some algorithms to perform trace merging. We show that each of these algorithms yield a different kind of temporal model, appropriate for different kinds of analyses. Our method is motivated by the possibility of analyzing simulations in a way that has not been done so far, and thus is developed with the needs of such a domain in mind. To demonstrate the practical feasibility of the proposed theoretical approach, we explain how to actually perform model checking of our temporal models using the NuSMV tool. Moreover, we provide proof-of-concept Java implementations of the proposed trace merging algorithms, which output NuSMV specifications. We also describe a simple case study using this implementation.

Keywords: model checking, simulation, runtime verification

1 Introduction

The execution of a program can be seen as a sequence of discrete states. By collecting information about such states, one can assemble a *trace* of the execution, which can then be analyzed in order to reveal properties of the underlying program. Clearly, the kind of information that must be collected to build such traces varies according to the kind of analysis one wishes to perform. By the same token, the representation of traces depends on the analysis technique to be used.

Currently, there are several approaches designed to analyze single program traces, and there are good reasons for this. For instance, by devising methods for the analysis of isolated program traces, one is in better position to create methods

[1] The authors have received financial support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) for the production of the present work (Proc. 551038/2007-1).

[2] Email: salem@ime.usp.br

[3] Email: acvm@ime.usp.br

doi:10.1016/j.entcs.2009.05.047

that operate on-the-fly (i.e., during the execution of the program) and, hence, that can provide runtime analysis procedures. Moreover, linear-time logics (e.g., Linear Temporal Logic (LTL) [17]) are suitable for the analysis of temporal properties of such traces (i.e., because a trace can be seen as a time line), and this allows for a kind of dynamic model checking.

However, if the objective is not on-the-fly verification, but simply an analysis of a program through its runtime behavior, it is reasonable to try to collect and combine as much information as possible about it. This implies, in particular, that one should try to combine several traces in order to gain a better understanding of the underlying program. This kind of idea is pursued in a number of works for various specific purposes.

In this paper, we present a new method for building such a trace combination and show how to perform model checking on it. The motivation that guides our efforts and differentiate much of our ideas is the possibility of using this kind of technique to analyze computer-aided simulations. Typically, when performing simulations to study some phenomenon, one collects several statistics, but discards the simulations' traces. We aim, therefore, at using this information that is usually disregarded. This, we believe, can help one to study the behavior of such simulations in a new way.

It is important to note that the technique presented here might not actually be appropriate in other contexts, such as when one wishes formal guarantees concerning the presence or absence of software defects. As it will become clear, our method depends on some assumptions and can only generate approximate models, which might elicit wrong responses from a model checker (i.e., false positives and false negatives). Nevertheless, in many situations it is better to have such an approximate understanding of a system rather than no understanding at all. Moreover, the quality of our approximations can be easily improved by adding more and longer traces.

Our approach consists in the following steps:

(i) collect several traces from the same program;

(ii) merge these traces into what we call a *state-space*;

(iii) convert the state-space into the input language of some model checker;

(iv) perform model checking.

The main worry here concerns the appropriate definition of trace and strategies for merging them. Each strategy has its own strengths and weaknesses, and are formulated according to the kind of problems we expect to find when analyzing simulations. We provide both a conceptual account as well as a concrete Java [18] implementation of the merging algorithms. As a proof-of-concept, the implementation produces a suitable input for the the NuSMV [6] model checker, but the technique can be easily ported to other model checkers with similar expressivity.

We assume that the program traces are available and, thus, we do not provide a method for collecting them. In this respect, we just define a trace file format that our proof-of-concept tool can read. This way, we do not restrict the possible applications of our method to any particular kind of program.

The text is organized as follows. Section 2 contextualizes our work. Section 3

establishes the formal foundations upon which the merging algorithms are defined. In Section 4, these algorithms are given. Then, in Section 5 we show how to actually perform model checking in the merged traces. Section 6, in turn, presents some illustrative experiments employing a simple economic simulation. Finally, Section 7 reflects about what was achieved and discusses what remains to be done.

Our Java implementation, as well as examples of input and output files, can be downloaded from the following URL:

http://www.ime.usp.br/~salem/papers/trace_merger.zip

2 Related Work

The analysis of isolated traces has received considerable attention recently. Such analysis either explore the traces explicitly (e.g., by employing some sort of linear-time logic model checking) or use traces implicitly (e.g., by instrumenting a program to perform invariant checks). On the first group, we may cite Geilen [11] as well as Finkbeiner and Sipma [10]. Both propose automatic methods to check, during runtime, if a given program violates an LTL specification. As far as we know, branching-time logics (e.g., Computational Tree Logic (CTL) [7]), which are also common in model checking, are not used in this context. On the second group, Design by Contract [15] approaches, such as JASS [2] and JML [4], are most significant. In such approaches, the programmer may explicitly state program invariants, as well as pre- and post-conditions of methods. Hence, while no explicit use of traces is made, there is the implicit assumption that every state of every trace must respect the given specification.

Concerning the analysis of multiple traces, there are a number of works that consider the problem of generating specifications from them. Boigelot and Godefroid [3] show a method to create approximate Finite State Machines from a program's executions. However, their approximations are different from ours in two important aspects. First, they assume the existence of correct execution *trees*, while we employ only *sequences* of states (i.e., what we call traces), which is a more restrictive assumption. So, in fact, they assume the existence of a structure similar to the one we try to create (i.e., directed graphs). Second, the quality of their approximations is given by the choice of the depth to employ when performing calculations in certain subtrees, while we give specific algorithms to create approximations and which yield different kinds of state-spaces.

Ammons *et al.* [1] present the notion of specification mining, an approach to discover protocols from execution traces of implementations. It employs code instrumentation and learning automata to create automata that approximate the specification of communication protocols between programs and Application Programming Interfaces (API). Their technique, though, is geared towards finding probable behavior, and as a result infrequent observations might be disregarded. Furthermore, they perform no simplifications in the traces themselves (i.e., without considering the frequency of the observed transitions over all traces), whereas each of our algorithms perform a specific kind of simplification, in order to provide state-spaces

suitable for distinct purposes.

With similar purposes, Lo *et al.* [13] investigate how to mine temporal logic formulae, Ernst [8] presents techniques to extract program invariants, and Ernst *et al.* [9] introduce Daikon, a system to extract such invariants. While our present work does not aim at discovering such information from traces, it will be clear that the resulting state-spaces from our algorithms can be used to such an end.

Model checking is traditionally employed as an exhaustive verification method, which demonstrates the presence or absence of some property on a formally specified system. However, this also results in efficiency problems, which motivates the development of approximate model checking algorithms, in which only a part of the possible execution paths are explored, such as the work of Cho *et al.* [5]. Our work is similar to this approach, but whereas their approximations are derived from formal analysis of the system, ours relies on the collected traces at runtime.

3 Formal Modelling

Let us establish a formal description of what we mean by trace and trace merging. With that, we shall be able to define precisely our algorithms and explain how model checking is performed.

The most basic definition we need is that of a *state property*.

Definition 3.1 A *state property* is a Boolean variable.

State properties denote the program characteristics we are interested in at individual program states. For example, questions such as "is the variable x currently positive?". They differ, thus, from general program properties, which can assert propositions about the temporal behavior of the program. In our approach, such general program properties are expressed as statements of some temporal logic during model checking (e.g., CTL) and, hence, do not need to be formally defined here.

When we are analyzing a program, we typically want to monitor many state properties. Consequently, the *state* of the program can be defined simply as a valuation for them.

Definition 3.2 Let $\mathscr{P}$ be a set of state properties. Then a *state* is a partial function $s : \mathscr{P} \to \{0, 1\}$.

We may now define a trace as a sequence of such states.

Definition 3.3 Let $\mathscr{P}$ be a set of state properties and $\mathscr{S}$ be a set of states over such properties. Then a *trace* is a sequence $(s_1, s_2, \ldots, s_n)$, where $s_i \in \mathscr{S}$, for every $1 \leq i \leq n$. We denote the length n of a trace t as $|t|$. Moreover, we say that s_1 is the *initial state* and that the ith position in the trace is the ith *instant*.

Finally, we may define the result of trace merging, which we call a *state-space*.

It is, essentially, a directed graph whose vertices are states and in which there is a special subset of initial states.

Definition 3.4 A *state-space* is a tuple $\langle \mathscr{S}, \mathscr{R}, \mathscr{I} \rangle$ where:

- $\mathscr{S}$ is a set of states;
- $\mathscr{R}$ is a relation over states;
- $\mathscr{I}$ is a subset of $\mathscr{S}$, called the *initial states*.

As we shall see below, the structure of the state-space depends on the kind of trace merging algorithm employed.

4 Merging Traces

We may now investigate how we can merge several traces in order to produce a state-space suitable, in particular, for branching-time model checking. In this section we present three algorithms for this, in descending order of simplicity. They differ on how much information is preserved from the original traces, which implies that the state-space generated by the algorithms might satisfy temporal properties that are not satisfied by a state-space without information loss. This, however, is not a problem as long as the appropriate merging algorithm is chosen for the desired verification. State-spaces with *less* information have the advantage of being more efficiently verified and might be sufficient in many cases. As one would expect, the more information is preserved, the larger is the resulting state-space.

Moreover, the merging might introduce new possible traces, owing to the implicit assumptions concerning which states should be considered equal. That is, when states from different traces are considered equal, they are merged into one, which connects the traces. This new connection might introduce new sequences of transitions, composed by parts of the original traces. Clearly, these new possibilities are only useful if the criterion of state equality is applicable in the problem being analyzed. Otherwise, the new transitions will be hindrances. Then, as each merging algorithm has its own notion of state equality, it follows that one should consider whether any of these notions are actually applicable in the problem to be analyzed.

In what follows we survey some situations in which each algorithm is adequate, thus motivating their use. For all the algorithms below, we assume that we have a set $\mathscr{S}$ of states over a set $\mathscr{P}$ of properties, and a set $\mathscr{T}$ of traces. Furthermore, we define that $|\mathscr{S}| = n$. To help in the presentation, we shall apply each technique to the example traces given in Figure 1.

4.1 State-Preserving Merger

One may build a state-space that preserves the states of $\mathscr{S}$ and employs $\mathscr{T}$ only to define the possible transitions among states. That is, we may ignore the instants when the transitions took place and only consider the fact that they eventually took place. This assumes that any given state depends only on the one immediately

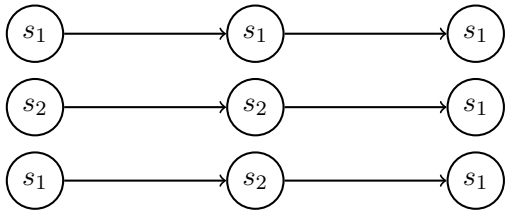

Fig. 1. Three example traces.

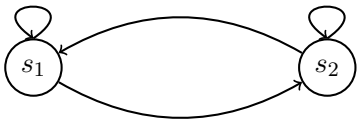

Fig. 2. The State-Preserving Merger of the example. Only the two states are preserved, but the several transitions found are all kept.

before it (i.e., a Markovian assumption). Figure 2 presents the result of applying this technique to the example traces of Figure 1. Algorithm 1 defines the technique precisely.

This merging strategy creates a state-space in which there are *more* transitions than in the original traces. This implies that, when analyzing it, every possible transition between states shall be considered, but impossible ones will also be found. Consider, for instance, the transformation from the traces shown in Figure 1 to the ones displayed in Figure 2. Besides the original traces, the model in Figure 2 clearly allows for a sequence of several s_2 states, whereas in the original traces we cannot find a trace with more than two consecutives s_2 states. The strategy, therefore, must be used with this characteristic in mind. It might be useful specially when checking safety temporal properties over states (e.g., whether the program may reach an undesirable state), in which it can guarantee the *absence* of certain traces at the price of eventually finding false-positives.

This algorithm makes strong assumptions and is limited to a particular set of verification tasks, which may not be always appropriate. But it has the advantage of keeping the state-space small. Since $|\mathscr{S}| = n$, the state-space can have at most n^2 transitions, and therefore the space required for its representation is $O(n^2)$.

4.2 Time-Preserving Merger

We may relax the Markovian assumption and keep temporal information. But to do so, we need to introduce the concept of *extended states*, which capture the idea that a state has a history.

Definition 4.1 Let s be a state and $t \in \mathbb{N}$, with $t \geq 1$. Then the pair (s, t) is an *extended state*. Moreover, we say that t is an *instant*.

Definitions for *extended trace* and *extended state-space* are analogous to those given previously. And for the sake of simplicity, we shall omit the *extended* qualification whenever it is clear from context.

Algorithm 1 *State-Preserving Merger.*
Input: *A set S of possible states and a set T of traces over S.*
Output: *A state-space.*

Let R be an empty binary relation
Let I be an empty set
foreach $t \in T$ **do**
  Let s_1 be the initial state of t
  $I \leftarrow I \cup \{s_1\}$

  foreach (s_i, s_{i+1}) *subtrace of t* **do**
    $R \leftarrow R \cup \{(s_i, s_{i+1})\}$
  end
end
Return $\langle S, R, I \rangle$

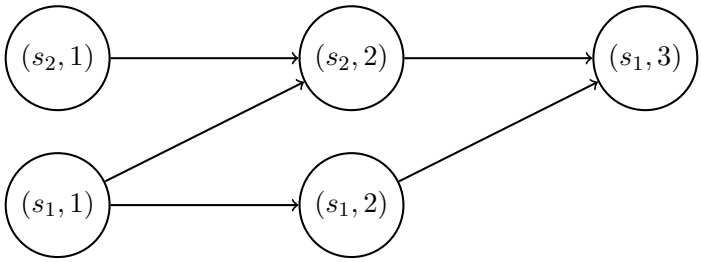

Fig. 3. The Time-Preserving Merger of the example. Notice that this state-space is composed of extended states. In instants 1 and 2, both s_1 and s_2 states are possible. But in instant 3, only s_1 is possible, as all traces converge to it.

Given these definitions, the algorithm can be described as follows. For each instant, we consider all states that appear in some trace and define a set of extended states for that particular instant. Then we connect the several elements in these extended states sets according to the transitions present in the traces. Figure 3 presents the result of applying this technique to the example of Figure 1. Algorithm 2 defines the method precisely.

Time-preserving merger employs all the available information in the traces. Hence, it is useful when one cannot afford many simplifications. For example, if we wish to check the presence of a particular subtrace with more than two states, this strategy will be required, since the others necessarily disregard some possible transitions in their simplifications. Such long subtraces, in turn, might be particularly needed when the time (measured in number of states) a program remains in some state is of importance (e.g., real time systems, computer simulations). Consider, for instance, the requirement that the program represented by the traces of Figure 1 must not contain a subtrace composed of three or more s_2 states (e.g., (s_1, s_2, s_2, s_2)). From the figure we can see that no trace violates the requirement and from Figure 2 we can see that the time-preserving strategy will generate a state-space that do not violate the requirement either. However, the state-preserving strategy we saw

Algorithm 2 *Time-Preserving Merger.*
Input: *A set S of possible states and a set T of traces over S.*
Output: *A state-space.*

Let S' be an empty set
Let R be an empty binary relation
Let I be an empty set

foreach $t \in T$ **do**
  Let s_1 be the initial state of t
  $S' \leftarrow S' \cup \{(s_1, 1)\}$
  $I \leftarrow I \cup \{(s_1, 1)\}$
end

Let n be the length of the largest trace in T
for $i \leftarrow 1$ **to** $n - 1$ **do**
  foreach $t \in T$ **do**
    if $|t| \geq i + 1$ **then**
      Let s_i be the i-th state in t
      Let s_{i+1} be the $(i+1)$-th state in t
      Let x be the tuple (s_i, i)
      Let y be the tuple $(s_{i+1}, i+1)$

      $S' \leftarrow S' \cup \{y\}$
      $R \leftarrow R \cup \{(x, y)\}$
    end
  end
end

Return $\langle S', R, I \rangle$

previously would clearly generate a state-space that violates the requirement, since it has a loop on state s_2.

This algorithm assumes that if the same state of affairs holds in the same instant in two different traces, then the future from that point on should contain the possibilities found on both traces. As a result, the generated state-space will allow for sequences of transitions that are not found in the original traces. Such a state-space can be useful when we believe that the time at which events happen is important to determine the subsequent future, because the new transitions that are inferred will allow us to consider possible behaviors that would not have been found otherwise.

Let us analyze the space required for the state-space produced by this strategy. Assume that the largest trace has length m. For every instant we have a set of extended states. In the worst case, all of these sets are of size n (i.e., contain extended versions of all states in $\mathscr{S}$). Clearly, then, in the worst case we have mn

extended states to consider. Moreover, each set of extended states can have, at most, n^2 transitions going out of it. From this it follows that the state-space may have, at most, $(m-1)n^2$ transitions. Thus, the size of the representation is about $mn + (m-1)n^2$, which is $O(mn^2)$.

4.3 Change-Preserving Merger

It is more informative to keep all temporal relations found on the collected traces, but it can be also rather expensive. An alternative approach is to modify the temporal-preserving algorithm to discard some extended states, while still maintaining temporal relations among the remaining ones, thus achieving a compromise between state-preserving and time-preserving methods.

But which extended states should we dismiss? That depends on what kind of questions we wish to answer with the resulting state-space. A reasonable criterion, we believe, is to keep only the necessary temporal information to distinguish between state changes that happen in the traces. So, for example, a trace of the form $(s_1, s_1, s_1, s_2, s_1)$ can be compressed into (s_1, s_2, s_1), which preserves the information that s_2 eventually appears. Our method, though, also preserves the original instant that the change takes place. Figure 4 shows the method applied to the example of Figure 1. Algorithm 3, which is just a variation of Algorithm 2, describes the method precisely.

This strategy is particularly suitable to check liveness temporal properties. That is, if one is interested in discovering whether the program eventually reaches a particular state from some other state, it is reasonable to exclude sequences of repeated states between them, which only slows down the search in this case. Furthermore, if the time that it takes for a change to happen is important, it can also be specified in a formula to be checked, since the original instants are preserved.

This change-preserving method is of interest only if the traces to be merged present a significant amount of segments that can be eliminated. The worst case scenario, however, is even worse than that of time-preserving merging. This happens because, unlike time-preserving merger, change-preserving can connect extended states that are not temporally consecutive. Let us outline its worst case complexity. Again, define m to be the length of the largest trace and let us have m sets of extended states, each containing the extended states that appear in some particular instant. Hence, again we shall have mn extended states to consider in the worst case. Now, however, each set of extended states may connect to all sets that come after it, except the one that comes immediately after it. Therefore, the total number of transitions will be, at most, $\sum_{i=1}^{m-2}(n-1)^2 i$, which is $\frac{m^2-3m+2}{2}(n-1)^2$. As a result, the total worst case space complexity is $O(m^2n^2)$.

To see this more clearly, consider a set of traces $\mathscr{T}_1$ which is a worst-case scenario to the time-preserving algorithm. Assume, moreover, that the traces in $\mathscr{T}_1$ have no transitions connecting equal states. Therefore, applying the change-preserving algorithm to $\mathscr{T}_1$ will yield the same state-space that applying the time-preserving one. Now consider another set of traces, $\mathscr{T}_2$, crafted with the exact repetitions that, when eliminated, would connect extended states temporally distant. Then,

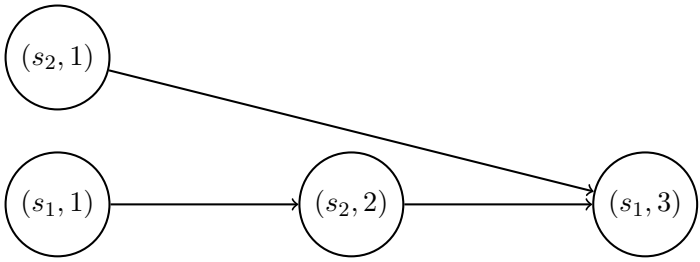

Fig. 4. The Change-Preserving Merger of the example. Compare this with Figure 3.

the result of merging $\mathscr{T}_1 \cup \mathscr{T}_2$ will yield an even worse state-space, because it will have transitions that connect both consecutive and not consecutive instants.

5 Performing Model Checking

Once the traces are merged, we have a state-space that can undergo model checking, using either LTL or CTL. Theoretically, there is not much more to be done. On the other hand, from a practical stand point, it is still necessary to provide an actual implementation that performs the model checking. To this end, we simply translate the state-spaces to the input language of an existing model checker, NuSMV. The translation specifies a Finite State Machine (FSM) whose states and transitions are those in our state-space, with the addition of a few new transitions to account for a technicality. Then, NuSMV can be used directly to investigate the temporal properties of the FSM.

5.1 Translating to a NuSMV FSM

NuSMV provides a rich specification language, including data types and modules for specification decomposition. To our purposes, though, only a subset of this power is needed. The specifications that we are interested in have the following characteristics:

States Let $\{s_1, s_2, \ldots, s_n\}$ be the states in our state-space. Then they are translated to NuSMV as the following variable declaration:

$$state : s_1, s_2, \ldots, s_n;$$

Time Let m be the length of the largest trace used to compose the state-space. Then time is captured in NuSMV with the following declaration:

$$time : 1..m;$$

Initial States Possible initial states and instants are given as a logical disjunction of expressions that describe each initial state as follows. Let (s, t) be an initial extended state. Then its logical expression in NuSMV is:

$$(state = s \wedge time = t)$$

Algorithm 3 *Change-Preserving Merger.*
Input: *A set S of possible states and a set T of traces over S.*
Output: *A state-space.*

Let S' be an empty set
Let R be an empty binary relation
Let I be an empty set
Let $P[]$ be an array that maps traces into extended states

foreach $t \in T$ **do**
  Let s_1 be the initial state of t
  $I \leftarrow I \cup \{(s_1, 1)\}$
  $S' \leftarrow S' \cup \{(s_1, 1)\}$
  $P[t] \leftarrow (s_1, 1)$
end

Let n be the length of the largest trace in T
for $i \leftarrow 1$ **to** $n - 1$ **do**
  foreach $t \in T$ **do**
    if $|t| \geq i + 1$ **then**
      Let s_{i+1} be the $(i+1)$-th state in t
      Let s_p be the first coordinate of the pair in $P[t]$

      if $s_{i+1} \neq s_p$ **then**
        Let y be the tuple $(s_{i+1}, i + 1)$
        $S' \leftarrow S' \cup \{y\}$
        $R \leftarrow R \cup \{(P[t], y)\}$
        $P[t] \leftarrow y$
      end
    end
  end
end

Return $\langle S', R, I \rangle$

Transition Transitions are also specified as logical disjunctions of expressions that describe each individual transition as follows. Let $((s_1, t_1), (s_2, t_2))$ be a transition between extended states. Then its logical expression in NuSMV is:

$$(state = s_1 \wedge next(state) = s_2 \wedge time = t_1 \wedge next(time) = t_2)$$

In all of the above rules, if time is not relevant (e.g., as in state-preserving merging), it is simply not specified.

The translation is almost straightforward. But because the future is technically endless, we need to add some transition to states that have no transitions (e.g., those

which are at the end of the longest traces). We do this by inserting a loop in such states, which means that when we don't have data to infer the future, we simply assume that it'll remain the same.

Figure 5 present the specification for the example shown in Figure 3.

5.2 Exploring the NuSMV FSM

For the most part, the FSM can be explored as any other NuSMV FSM. That is, by appending the desired logical specification formulae at the end of the FSM file and running NuSMV. For example, if we want to check the CTL formula $AF(state = s_1)$, we merely append it to the end of the generated FSM file as:

```
SPEC AF (state = s1)
```

However, it is necessary to bear in mind that some states have artificial loops on them, which we added during the translation in order to make sure every state has a successor. This decision is mostly harmless, but it'll imply that care must be taken when specifying logical formulae that are supposed to be true in every state of a temporal path, since they will not be evaluated correctly in the states with these artificial loops. For example, if we want to check the CTL formula $AG(state = s_1 \rightarrow AF(state = s_2))$, we must also specify a maximum instant to evaluate the AG connective. So, if our traces have around 200 states, the 100th instant might be a good threshold and would yield the following specification:

```
SPEC AG ((state = s1 & time < 100) -> AF (state = s2))
```

While limiting the verification to a particular number of states might seem a crude mechanism in general model checking, in our case it is a reasonable approach, since the traces we have access to are themselves subject to such limitation. So if we want to cover all available information, it is sufficient to fix such a maximum instant as the size of the largest trace (e.g., in the above example, 200). And though it is always desirable to cover all the available information, for efficiency reasons one might need to give up part of it, thus making this kind of instant limitation useful.

6 Case Study

The approach described above can, of course, be used with any program capable of being instrumented to generate a trace file. In this section, however, we shall consider only simulators, which are a particular kind of program that, we believe, can be greatly benefited by our method. While approaches for computer-aided simulation vary, some of them (e.g., [16,14]; see also [12]) are based on specifying the behavior of some system to be studied as an executable program instrumented in special ways. Currently, the analysis that is performed using such simulations is based mostly on graphical visualization and compilation of statistics concerning the value of variables. No method to analyze their temporal behavior through logical

```
MODULE main
  VAR
    state:{s2, s1};
    time: 1 .. 3;
  INIT
    (state = s1 & time = 1)|
    (state = s2 & time = 1)
  TRANS
    (state=s1 & next(state)=s1 & time = 1 & next(time) = 2)|
    (state=s1 & next(state)=s2 & time = 1 & next(time) = 2)|
    (state=s1 & next(state)=s1 & time = 2 & next(time) = 3)|
    (state=s2 & next(state)=s2 & time = 1 & next(time) = 2)|
    (state=s1 & next(state)=s1 & time = 3 & next(time) = 3)|
    (state=s2 & next(state)=s1 & time = 2 & next(time) = 3)
```

Fig. 5. The NuSMV specification of the example given in Figure 3.

specifications exists.

Clearly, then, our method can be applied to allow such logical analyses to be performed, by instrumenting simulators to generate a trace every time the simulation is run. Let us then explore a concrete example of such an application. We shall consider a simple fictitious model for the economic behavior of nations. Our aim is to study the relation between some kinds of economic events, such as economic depressions. The program is provided together with the merging algorithms' implementations, which can be downloaded at the URL given in Section 1.

Our model has the following variables of interest: inflation, economic activity and standard of living. At each simulation step, their values are updated according to some rules. The simulator simply runs the model several times, collecting traces and, in the end, producing a trace file that can be read by our trace merger.

We are not interested in a random combination of the model's variables. Instead, we choose to focus our resources in particular combinations that have some economic meaning of interest. In this experiment, we chose to study some types of economic growth, stagnation and depression. Our states are the following:

- *Growth type 1 (g_1).* The best kind of economic growth, with low inflation and a good standard of living.
- *Growth type 2 (g_2).* Economic growth with no guarantee of standard of living.
- *Depression type 1 (d_1).* The worst kind of economic depression, with high inflation and a bad standard of living.
- *Depression type 2 (d_2).* A less severe kind of economic depression.
- *Stagnation (stagnation).* The absence of either growth or depression.
- *Other (other).* Other combinations of the model properties which are not of interest to the experiment.

To invoke the simulator, one must specify the number of traces, the number of

states per trace and the output file. In our experiment, we did that as follows:

```
$ java -jar economicmodel.jar 200 100 sim_traces.txt
```

Once the simulation is finished, the trace merger must be called, with the merging algorithm to be used, the trace file and the desired NuSMV output file name. We shall use the time-preserving merger in this experiment, as follows:

```
$ java -jar tracemerger.jar tp sim_traces.txt sim_model_tp.smv
```

This gives us a file ready to be read by NuSMV. But we still need to specify the logical properties that we want to investigate. We choose the following:

- Are depressions or stagnation inevitable? In CTL:

$$AF(state = d_1 \vee state = d_2 \vee state = stagnation)$$

- Can depressions be eventually followed by growth? Recall that the last states in the FSM are looping, so that specifications referring to their future will be inaccurate. Hence, to specify the desired property in CTL, we must choose a maximum instant for the depressions to start, so that the future that follows remains correct. We arbitrarily chose the 50th instant. Notice also that a change-preserving trace merger would be sufficient to analyze this property. We get the following CTL proposition:

$$AG((time < 50 \wedge (state = d_1 \vee state = d_2)) \rightarrow EF(state = g_1 \vee state = g_2))$$

- Is a long continuous sequence of type 2 depressions possible? Since in CTL it is not possible to have a variable to range over a set of values, we cannot specify the duration of this sequence as the difference between its last and first instants. But we can specify sequences by using the fact that CTL provides an operator to refer to the successor state. To use this, notice also that we need a time-preserving trace merger, so that both instants and all successor states are preserved. In CTL, we have something of the following form (the '...' indicate that more EX expressions are coupled):

$$EF(state = d_2 \wedge EX(state = d_2 \wedge EX(state = d_2 \wedge \ldots)))$$

- Can growth be immediately followed by stagnation? In CTL:

$$EF((state = g_1 \vee state = g_2) \wedge EX(state = stagnation))$$

To answer these questions, we open the NuSMV file we have just generated and append them to it. Then we just run NuSMV with this input. As it turns out, only the second and the third propositions above are true. But, of course, if the simulation had being implemented in a different manner, different results could arise.

Furthermore, since the state-space employed is an imperfect approximation of the actual system, it is necessary to assume that the answers given by the model

checker might be wrong. Hence, one may wonder how these answers are useful at all. To address this concern, two things must be understood. First, these uncertain answers are indeed valuable if more rigorous methods are not available, since they do reflect the behavior of the system, albeit in an incomplete manner. And, second, the technique is not supposed to assert infallible propositions, but only to *suggest* possible behaviors, given the available data and the suppositions concerning the simulated system. The user is expected to use such suggestions as guidelines for further scrutiny, either by providing more and longer traces to our algorithms or by employing other methods (e.g., writing specific test cases for the detected behavior, inspecting the source code).

For example, the user might suspect that the first proposition above was actually true and might chose to investigate it further. To this end, he might run the simulation again, but recording longer traces in order to be able to detect events that, he imagines, will only happen in distant futures. He might also manually examine the simulation's source code in order to look for possible implementation errors concerning the variables referred by the proposition.

7 Conclusion

In this work we presented an approach to merge program traces so that one can perform branching-time model checking on them. We described both abstract algorithms and actual implementations. Our motivation, we stated, is to provide a way to use traces from simulations for analytical purposes. The case study given reflects this objective. However, the algorithms are general and can be applied in arbitrary programs, as long as the required assumptions are met. For instance, one may use them as a formal complement to software testing, but only as a way to *suggest* possible problems.

We highlighted the merging techniques which seem most interesting to us. But they are not the only ones possible, and perhaps for different problems other manners may be more appropriate. For example, the change-preserving algorithm of Section 4.3 also preserves the instant that a change takes place, but maybe this is not necessary for some applications. Furthermore, we purposefully disregarded techniques in which no merging would take place and every trace would be analyzed independently, since these would not belong to the core subject of the paper.

The amount and length of traces required to generate representative space-states can be quite large. This problem must still be addressed, and we believe that one promising way to deal with it is to selectively discard traces. That is, by having criteria of trace relevance in relation to some logical property of interest, one may keep only those traces that are more likely to be useful. Research needs to be done to find such criteria, but it seems to us that static analysis techniques might be helpful.

Our method is clearly incomplete, in the sense that it can always be the case that some important transition was not captured in the traces. It is still necessary, then, to develop coverage indices to measure how much of the possible traces have

been in fact analyzed. Again, we think that static analysis might be useful (e.g., because we could know *a priori* that some traces would be impossible, which would increase our coverage estimation).

Finally, the state-space produced by merging traces can also be useful to other things besides model checking. As an approximate representation of an underlying program, it might be interesting to apply machine learning algorithms to it, in order to extract new information about its behavior. For example, it might be the case that every time a particular state comes up, another state never appears, but we are unaware of this relation. A machine learning algorithm could, perhaps, reveal it to us.

References

[1] Ammons, G., R. Bodík and J. R. Larus, *Mining specifications*, SIGPLAN Not. **37** (2002), pp. 4–16.

[2] Bartetzko, D., C. Fischer, M. Moller and H. Wehrheim, *Jass - java with assertions* (2001).

[3] Boigelot, B. and P. Godefroid, *Automatic synthesis of specifications from the dynamic observation of reactive programs*, in: *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems* (1997), pp. 321–333.

[4] Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino and E. Poll, *An overview of JML tools and applications* (2003).

[5] Cho, H., G. Hachtel, E. Macii, B. Plessier and F. Somenzi, *Algorithms for approximate FSM traversal based on state space decomposition*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **15** (1996), pp. 1465 – 1478.

[6] Cimatti, A., E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, *NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking*, in: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, LNCS **2404** (2002).

[7] Clarke, E. M. and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, in: *Logic of Programs, Workshop* (1981), pp. 52–71.

[8] Ernst, M. D., "Dynamically Discovering Likely Program Invariants," Ph.D. thesis, University of Washington (2000).

[9] Ernst, M. D., J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz and C. Xiao, *The Daikon system for dynamic detection of likely invariants*, Science of Computer Programming **69** (2007), pp. 35–45.

[10] Finkbeiner, B. and H. Sipma, *Checking finite traces using alternating automata*, Form. Methods Syst. Des. **24** (2004), pp. 101–127.

[11] Geilen, M., *On the construction of monitors for temporal logic properties*, Electr. Notes Theor. Comput. Sci. **55** (2001).

[12] Gilbert, N. and S. Bankers, *Platforms and methods for agent-based modeling*, Proceedings of the National Academy of Sciences of the United States **99** (2002).

[13] Lo, D., S.-C. Khoo and C.Liu, *Mining temporal rules from program execution traces*, in: *3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA'07)*, 2007.

[14] Luke, S., C. Cioffi-Revilla, L. Panait and K. Sullivan, *Mason: A new multi-agent simulation toolkit* (2004), paper and toolkit are available at http://cs.gmu.edu/ eclab/projects/mason/.

[15] Meyer, B., *Applying "design by contract"*, Computer **25** (1992), pp. 40–51.

[16] North, M., N. Collier and J. R. Vos, *Experiences creating three implementations of the repast agent modeling toolkit*, ACM Transactions on Modeling and Computer Simulation **16** (2006), pp. 1–25, toolkit is available at http://repast.sourceforge.net/.

[17] Pnueli, A., *The temporal logic of programs*, in: *18th IEEE FOCS*, 1977, pp. 46–57.

[18] Sun Microsystems, *Java technology* (2007), http://java.sun.com/.