Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 179–193
www.elsevier.com/locate/entcs

Model-based Runtime Verification Framework[1]

Yuhong Zhao Franz Rammig

Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany

Abstract

Model-based runtime verification is an extension to the state-of-the-art runtime verification, aimed at checking at runtime the system implementation against the system model (consistency checking) and the system model against the system specification (safety checking). Notice that our runtime verification works at the model level, thus, we do not need to strictly synchronize this runtime verification with the system execution. In fact, we mainly use the runtime information (current states) obtained from the system execution to reduce the state space (of the system model) to be explored. It means that this model-based runtime verification might run before or after the system execution, i.e., switch alternately between a preventive pre-checking mode and a maintaining post-checking mode. In order to make it run ahead of the system execution for as long time as possible, we present two possible strategies so that this runtime verification can selectively reduce the state space (of the system model) to be explored by making the system model enriched with probabilities and additional information derived and learned at the system testing phase.

Keywords: runtime verification, model-based runtime verification, on-the-fly ACTL/LTL model checking

1 Introduction

Model-driven Engineering (MDE) [11] is an efficient software engineering approach to complex systems development. According to MDE, we can follow three steps to develop a (complex) system:

(i) model the system according to the system specification,

(ii) verify the system model against the system specification, and

(iii) synthesize the system implementation (source code) from the system model.

To ensure the correctness of a system under development, verification techniques, e.g., model checking, theorem proving, and validation techniques, e.g., simulation

[1] This work is developed in the course of the Collaborative Research Center 614 - Self-Optimizing Concepts and Structures in Mechanical Engineering - Paderborn University, and is published on its behalf and funded by the Deutsche Forschungsgemeinschaft (DFG).

doi:10.1016/j.entcs.2009.09.035

and testing are usually applied respectively to check the system model against the system specification and the system implementation against the system specification as well as the consistency between the system implementation and the system model as shown in Fig. 1 (theorem proving is not discussed in this paper).

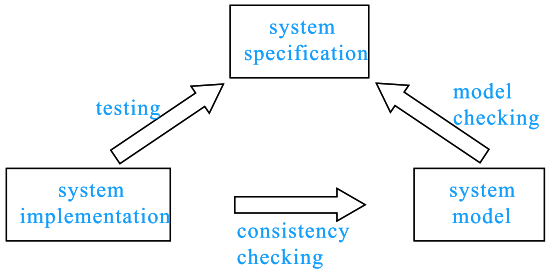

Fig. 1. check correctness for system under development

However, for a complex system, simulation and testing only explore *some* of the possible behaviors and scenarios of the system implementation. Although model checking conducts an *exhaustive* exploration of all possible behaviors, it has to face the *state explosion* problem. In addition, it is usually difficult to decide if the set of the system properties to be checked is *complete* or not.

In practice, different checking methods are presented to complement each other so that we could dig up more deep-corner errors in complex safety critical systems and thus increase our confidence to the correctness of the system implementation and the system model as well as the consistency between them.

In contrast to the offline checking techniques, the state-of-the-art runtime verification (Section 3) takes the system implementation and the system specification into account as shown in Fig. 2. The basic idea is to monitor the execution of the system implementation and in the meantime to check the so far observed execution trace against the system properties specified usually by LTL formulas.

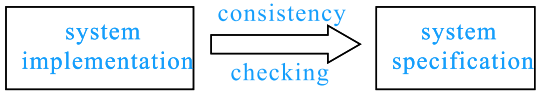

Fig. 2. state-of-the-art runtime verification framework

An actual state is usually monitored by means of instrumenting additional code to some proper places in the source code of the system implementation. The variables contained in the system properties to be checked can help to decide what variables need to be observed and where to add the corresponding instrumentation code into the system source code. Notice that the number and the places of the variables to be monitored might affect the granularity of the transitions between the monitored states as shown in Fig. 3. There usually exist some intermediate states between two concatenated monitored states.

This kind of runtime verification can only do post-checking, i.e., the checking progress always falls behind the system execution because the checking procedure

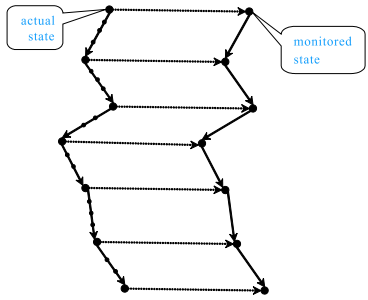

Fig. 3. actual execution trace and monitored execution trace

can continue only after a new state has been observed. Consequently, property violations are usually detected after they have already happened.

Notice that even if the properties are checked *correct* with this approach, it does not imply that the monitored execution trace conforms to the system model and the system model satisfies the same properties as well. The former depends on the consistency between the system implementation and the system model, while the latter depends on the granularity of the system model and the properties to be checked.

To overcome this problem, we extend this state-of-the-art runtime verification to our model-based runtime verification [15,16] that takes the system implementation, the system model and the system specification together into account as shown in Fig. 4. The basic idea is to check at runtime whether the monitored execution trace of the system conforms to the system model on the one hand and if a partial system model that covers the monitored states satisfies the system properties on the other hand.

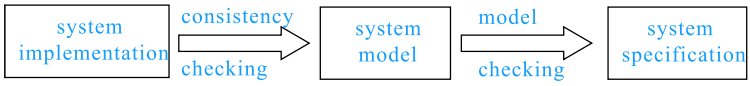

Fig. 4. model-based runtime verification framework

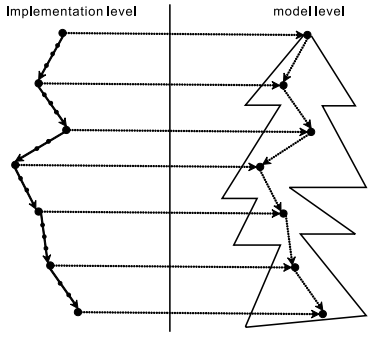

Fig. 5. model-based runtime verification procedure

Here the partial system model is obtained by exploring only such kind of states that can be reached from those current states monitored at runtime as shown in Fig. 5. Intuitively, if this partial system model is checked correct against the system specification, and the monitored states conform to the corresponding states in the partial system model, then we have more confidence to the correctness of the actual execution trace. It doesn't matter even if the rest of the system model might still contain some errors.

Because we check correctness at model level, this kind of runtime verification can make progress even if the current state of the system execution is not observed yet. That is, it is possible for the model-based runtime verification to run ahead of the system execution and thus the property violations might be detected before they have already occurred. In essence, the monitored (concrete) states are used to locate the corresponding (abstract) states in the system model so as to reduce the state space to be explored.

Therefore, the model-based runtime verification can do pre-checking and post-checking according to whether this runtime verification takes the leading position against the system execution or not. It looks as if the runtime verification and the system execution are involved into a two-player game. Of course, it is desirable that the runtime verification can run ahead of the system execution for as long time as possible in the course of the game. For this purpose, we present two strategies to make the runtime verification have more chance or higher probability to win against the system execution.

Notice that the properties to be checked here are requirements to the system model and thus we do not need to concern ourselves with the problem of the granularity of the transitions between the monitored states. In addition to LTL formulas, our model-based runtime verification can also check ACTL formulas.

The remainder of this paper is organized as follows: Section 2 first introduces the basic idea on the model-based runtime verification and then presents the two possible strategies to speed it up; Section 3 discusses the related work to the state-of-the-art runtime verification; and finally, Section 4 ends the paper with conclusion.

2 Model-based Runtime Verification Framework

2.1 Problem Statement

Originally, we did present this mode-based runtime verification technique [15,16] to ensure the safety and consistency for self-optimizing mechatronic systems [9], which consist of dynamic software components that can be optimized and even reconfigured at runtime. Of course, the application of this technique is not limited to such kind of dynamic systems. In this paper, we just take the self-optimizing mechatronic systems as an application context to illustrate the basic idea on the model-based runtime verification.

We regard the model-based runtime verification as a service of a Real-Time Operating System (RTOS). Thus, a possible application scenario is that a real-time system to be checked knows in advance when and where to trigger a model-based

runtime verification service. In the case of a self-optimizing mechatronic system, we suppose that the system send a checking request to the RTOS (on which it runs) some time before it actually does optimize or reconfigure itself. The operating system then invokes the model-based runtime verification service to check whether the system optimization or reconfiguration is safe or not. It is desirable if the runtime verification service could give a definite result before the optimization or reconfiguration is really done by the system.

Without loss of generality, let $M = \{M_1, M_2, \cdots, M_n\}$ be a real-time system model which contains n components $M_1, M_2, \cdots, M_n$ running in parallel. M can reconfigure itself at runtime by adding a new component M_i' to and/or removing an existing component M_i from M. Without doubt, it is mandatory for M to remain *safe* after this runtime reconfiguration. Here we consider such safety properties that can be specified as ACTL and/or LTL formulas. Notice that the discrete time extensions to ACTL and LTL formulas are just shorthand notations to the usual ACTL and LTL formulas [5].

Obviously, with the increase of the component number, the state space of the overall system would be too large to be exhaustively explored by the offline checking techniques within a reasonable time and memory overhead. However, by runtime verification, instead of composing all the components together, we go to check each individual component or some critical region in each component at runtime that might be affected by the reconfiguration of the system.

Let t_r be a time point at which a reconfiguration request is sent to the RTOS and t_0 ($t_0 > t_r$) be a time point at which the system starts to really do the reconfiguration. During this time interval $t_d = t_0 - t_r$, the RTOS triggers the runtime verification service to check the safety of this system reconfiguration. The main goal of the model-based runtime verification is to answer within the given time interval t_d whether M still maintains safe after the reconfiguration is really done. According to this answer, the RTOS can decide to accept or reject the reconfiguration request.

2.2 Pipelined Working Principle

Unfortunately, it is not possible under all circumstances to finish the checking process before the reconfiguration is really done by the system. To gain more checking time, we make the model checking run interleaved with the execution of the reconfigurable system in a pipelined working manner. The sequence diagram Fig. 6 illustrates the cooperation between the safety checking (ACTL/LTL model checking) and the reconfigurable system (real-time application).

Whenever the RTOS receives a checking request from the real-time application at time point t_r that the system wants to reconfigure itself at the time point t_0, it goes to invoke the verification service to check whether the reconfiguration is safe or not. The answer has to be given within the required time interval t_d ($= t_0 - t_r$) in this case. If lucky, the runtime verification might complete the checking task before the given time slot is over. However, it's usually not the case for most complex systems. Therefore, it is quite possible that, within this t_d time units (the first checking round), only the next $\Delta t_1 = t_1 - t_0$ time steps starting from the initial

given states are checked *Yes*, which means that the reconfiguration remains safe at least up to the coming Δt_1 time steps.[2] In this case, the RTOS does allow the real-time application to make the reconfiguration and execute forward Δt_1 time steps. During this period of time (the second checking round), the verification service continues to check, say the next $\Delta t_2 = t_2 - t_1$ time steps. Correspondingly, the real-time application can then go ahead the next Δt_2 time steps. It is an essential property of our approach that at each time point t_i ($i \geq 0$), the real-time application reports its current (concrete) state, say s'_i, to the verification service. Let σ be a mapping function from concrete states to abstract states of the system. With this runtime information s'_i, the verification service can identify in the system model the corresponding abstract state $s_i = \sigma(s'_i)$ and thus avoid searching the whole state space of the system model by only checking a sufficient partial state space reachable from this specific state s_i mapped from the concrete state s'_i. In this way, the working load of the safety checking can be reduced to a greater extent.

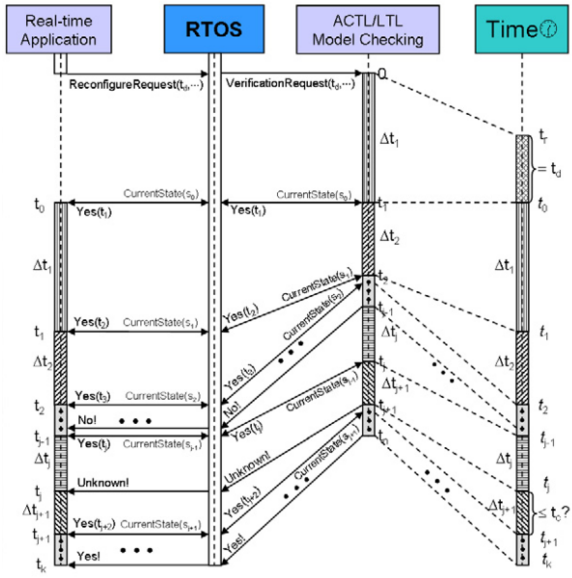

Fig. 6. pipelined working principle

On the other hand, if the verification service could not map the concrete state s'_i to an appropriate abstract state, then it means that an inconsistency has happened. If so, the runtime verification will stop the checking process and inform the RTOS to deal with this problem. If no inconsistency is checked out, this pipelined working procedure will continue until one of the following three cases happens:

Case *No*: if at some time point an error is detected, the runtime verification terminates with the answer *No* to the RTOS.

Case *Unknown*: if at some time point, say t_j, although the checking result is *positive*, but the time interval $t_{j+1} - t_j$ is not more than a pre-defined time bound

[2] Here a time step means a transition (step) at model level, which in turn represents some time units at implementation level.

t_c, the minimal (time) distance that the runtime verification is required to run ahead of the real-time application, then, we give up the checking process and report *Unknown* to the RTOS.

Case *Yes*: if a sufficient partial state space that covers the actual execution trace of the real-time application is successfully checked, then, we report definitely *Yes* to the RTOS and terminate the safety checking process (while still continuing the consistency checking). From now on, the reconfigured system can execute safely and consistently.

Notice that the first two cases only mean that the detected errors might happen in the future, because we check at model level and thus do not know whether the errors are spurious or not. To avoid the errors really to happen, we have to conservatively choose to reject the reconfiguration request and inform the real-time application that an error might emerge in the future. That is, the RTOS might raise an exception together with a counterexample (if necessary). How to handle the exception is application domain specific, thus we do not discuss this here.

The implementation of a component is in fact a refinement of the model of the component, i.e., the model is an abstraction of the implementation of the component. Thus, an ACTL/LTL formula being *true* at the model level implies that it is also *true* at the implementation level, while its being *false* at the model level does not imply that it is also *false* at the implementation level. In this sense, our runtime safety checking is conservative due to its being applied to the model level. However, the benefits of predicting and thus avoiding potential errors are gained just due to its being applied to the model level.

In fact, Fig. 6 just demonstrates an ideal pipelined cooperation between the reconfigurable system and the runtime verification via the RTOS as intermediary without considering any implementation details. To make this model-based runtime checking feasible, the following preconditions are supposed to be *true*:

- Each component in the reconfigurable system should be modeled as a finite state machine and be checked *correct* off-line under the given assumptions about the environments, on which it depends, at the design phase.
- Time bound should be attached to the eventuality operators (if any) in the ACTL/LTL formulas to be checked to avoid checking fairness conditions.
- The processing speed of the runtime verification should be sufficiently faster than that of the reconfigurable system.

In addition, we also suppose that the implementation (source code) of each component in the reconfigurable system is generated correctly from the model of the corresponding component and is instrumented by appropriate code so as to identify concrete states while the component is running.

2.3 Model Checking Methodology

It is easy to see that the pipelined working principle between the runtime verification and the system execution requires that safety checking should be done on-the-fly in a

top-down way as shown in Fig. 7. Each time an actual concrete state s'_i is monitored and successively mapped to the corresponding abstract state s_i in the system model (by consistency checking), the runtime verification needs to only explore the partial state space starting from s_i.

Let M be a system model and f be an ACTL/LTL formula to be checked. Both the ACTL and the LTL formulas can be transformed into (*Büchi*) automata by tableau construction [5]. Without loss of generality, let B be the automaton constructed from f. In case that f is ACTL formula, in order to make ACTL model checking done on-the-fly in a top-down way, we go to check the simulation pre-order between M and B incrementally [13], because M satisfies B if and only if B simulates M. To do this, we encode the properties of the simulation relation between M and B into a weakly negative Horn (NHORN) formula, a special type of CNF (Conjunctive Normal Form) formula, and then check if the NHORN formula is satisfiable or not (called NHORNSAT problem) in polynomial time. [2] presents an efficient on-the-fly algorithm to resolve HORNSAT problems, which receives one Horn clause at a time and allows fast queries about the satisfiability of the whole formula so far received. The dualization of this algorithm also gives an efficient linear time on-the-fly solution to the NHORNSAT problem. In case that f is LTL formula, we do LTL model checking on-the-fly by means of reachability analysis [12], which is also suitable for liveness properties, such as absence of deadlock or livelock. Of course, the above on-the-fly approaches to ACTL and LTL model checking need to be adapted and extended to fit the pipe-lined working manner between the system execution and runtime verification. However, due to the space limitation, we refer to [15] for more details although we have already updated the algorithms mentioned there since then.

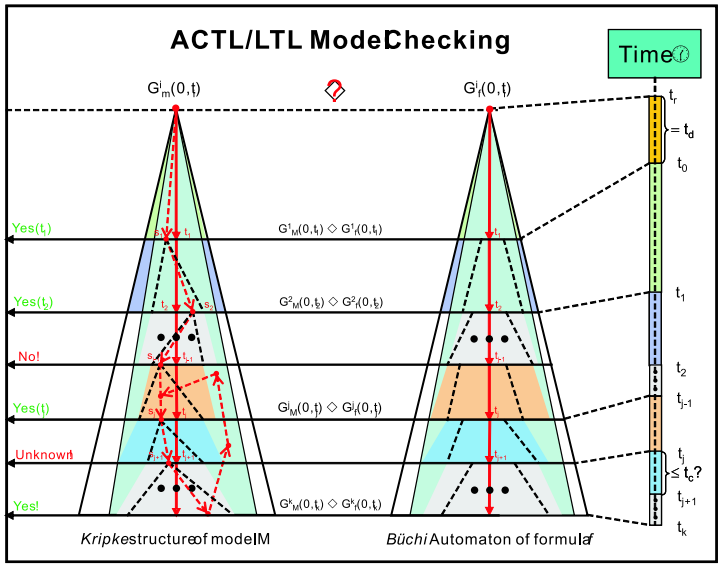

Fig. 7. On-the-fly ACTL/LTL model checking methodology

A standalone prototype for runtime model checking invariants, LTL and ACTL

properties is implemented. We have done some experiments for randomly generated state transition graphs so far. The execution traces are also generated randomly from the corresponding state transition graphs by simulation, which can simplify the monitoring procedure to capture the runtime information (current states) used for runtime model checking. In order to imitate the runtime verification running faster than the system execution, we make our runtime verifier run more time than the system execution does at each checking round. For example, if the speed of the runtime verification is supposed to be two times faster than that of the system execution, then we make our runtime checker run two times more time than the system execution does. In this way, we can estimate the performance of our model-based runtime verification to some degree.

According to our experience, it is good enough to make runtime verification look ahead two or three time steps at each checking round, for it takes not so much time and memory overhead. At present, we plan to do further experiments using the following two benchmarks:

- VLTS [3] benchmark suite derived from the communication protocols and concurrent systems in real life.
- BEEM [4] benchmark set derived from mutual exclusion algorithms, communication protocols and so on in research or industry area.

Obviously, the efficiency of our model-based runtime verification is highly affected by the branching factors of the system models under test. It is worth noting that the above benchmark models contain thousands or even millions of states, but the average degrees of most of them are less than 10 and the highest average degree among them is 40.

Compared to the usual (off-line) model checking, our model-based runtime verification can reduce the state space to be explored by using the monitored states obtained while the system is running. On this view, the computational complexity of the model-based runtime verification is less than that of the traditional model checking. Compared to the usual runtime verification, our model-based runtime verification checks the system properties at the model level while just using the monitored states to do consistency checking and then to shrink the state space to be explored. As a result, the computational complexity of the model-based runtime verification is greater than that of the conventional runtime verification. However, if we make our model-based runtime verification look ahead only several time steps at each checking round, then its computational complexity in terms of time and memory overhead will be closer to that of the state-of-the-art runtime verification. In addition, our model-based runtime verification can check more general properties specified by ACTL and/or LTL formulas, since [7] shows that the property patterns to be checked in practice are usually not very complex.

[3] http://www.inrialpes.fr/vasy/cadp/resources/
[4] http://anna.fi.muni.cz/models/

2.4 Game between Runtime Verification and System Execution

2.4.1 Pre-checking and post-checking

Ideally, we wish that the model-based runtime verification could always run enough time steps ahead of the system execution. However, we have to face the reality that the runtime verification might fall behind the system execution. To deal with this problem, we introduce two checking modes: *pre-checking* and *post-checking*. We say that the runtime verification is in pre-checking mode, if the runtime verification runs ahead of the system execution; otherwise, it is in post-checking mode.

In pre-checking mode, the runtime verification can predict violations before they really happen. In post-checking mode, violations can be detected after they have already happened. In fact, the model-based runtime verification can also predict violations even in post-checking mode because our runtime verification checks at the model level. If an error is found at some place in the partial state space being checked but without overlap with the monitored execution trace, then we can predict that the error might happen in the future. In any case, both checking modes are useful for safety-critical systems.

Notice that the model-based runtime verification can observe the actual execution trace of the system once it falls behind the system running. This means that the runtime verification only needs to explore a rather small state space in post-checking mode. Recall that the processing speed of the runtime verification is supposed to be faster than that of the system execution. Therefore, it is reasonable to argue that there still exists chance for the runtime verification to pass over the system running quickly.

On this view, it seems as if the runtime verification and the system execution are involved into a two-player game. In the course of the game, we say that the runtime verification *win*s against the system execution, if the runtime verification takes the leading position for a longer time than the system execution does. To achieve this purpose, we need to find a better strategy to make the runtime verification have more chance or higher probability to win against the system execution.

Notice that the checking speed of the runtime verification is heavily affected by the out-degrees of the states in the system model. Let d be a predefined upper bound for the out-degrees of the states in the system model. We say that a state is *critical*, if the number of the outgoing transitions of the state exceeds the threshold d.

Obviously, if there exist too many critical states in the system model, then the system model becomes very broad. In this case, the runtime verification could only look ahead a short distance within the same time interval. To speed it up, we'd better make the runtime verification have some intelligence so that it can intentionally reduce the state space to be checked whenever necessary.

2.4.2 Enrich system model with probabilities

Recall that the source code of the system implementation is usually validated by simulation and testing. Therefore, we should learn some heuristic knowledge at the

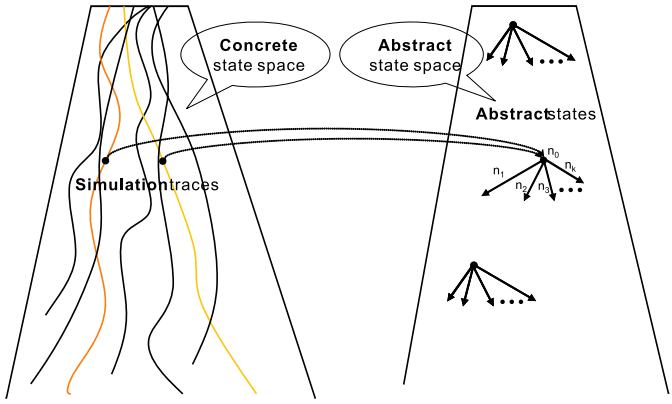

Fig. 8. derive probabilities from testing

system testing phase. For each critical state in the system model, we calculate how many simulation traces go through this critical state and among these simulation traces, we further calculate for each outgoing transition of this critical state how many simulation traces pass through this transition. In this way, we can estimate the probabilities of the outgoing transitions of the critical states in the system model as shown in Fig. 8.

Of course, these probabilities might not be very accurate. In fact, we just use them to set an initial order to the outgoing transitions of the critical states. This order will be updated later at runtime. For each critical state s_i, we order the outgoing transitions of s_i decreasingly according to the probabilities of these transitions. We then classify the first d transitions as the *major* transitions and the rest as the *minor* transitions of s_i.

Let $selected(s_i)$ be a set of the major transitions of s_i. During the checking procedure, each time a critical state is reached, the runtime verification can intentionally explore only the transitions in $selected(s_i)$. In this way, the state space to be checked is reduced to some extent. The runtime verification could thus look ahead further even within a limited time interval. The selected transitions are the major transitions of the critical states, which have higher probabilities than those of the unselected transitions. Thus, the chance is high that the monitored states do fall inside the selectively reduced state space having been checked safe.

Of course, we have to consider the worse case that the monitored states might fall outside the intentionally reduced state space as shown in Fig. 9. Let s_j be the first monitored state that falls outside the reduced state space and s_i be the critical state inside the reduced state space and closest to s_j. Then, we make the runtime verification switch to post-checking mode and continue the checking procedure from s_i.

Meanwhile, the runtime verification needs to adjust the selected transitions of s_i. There are different ways to do so. Let $deviated(s_i)$ be a set of the transitions from s_i that can reach s_j. A way to update the selected transitions of s_i is to replace some of the transitions in $selected(s_i)$ with the transitions in $deviated(s_i)$. Thus, if the runtime verification visits s_i again, it can intentionally explore the partial state

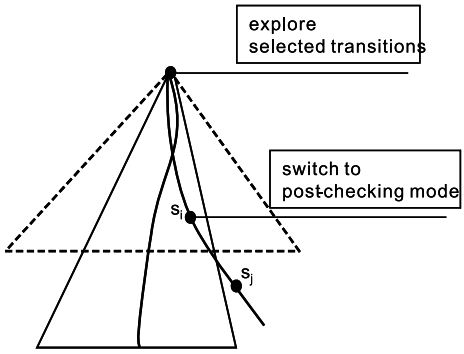

Fig. 9. intentionally reduced state space

space covering s_j.

2.4.3 Enrich system model with additional information

Notice that the system implementation is a refinement of the system model. This means that a concrete state in the system implementation usually contains more information than the corresponding abstract state in the system model. Thus, another strategy to speed up the runtime verification is to add this extra information to the related transition of the abstract state as shown in Fig. 10.

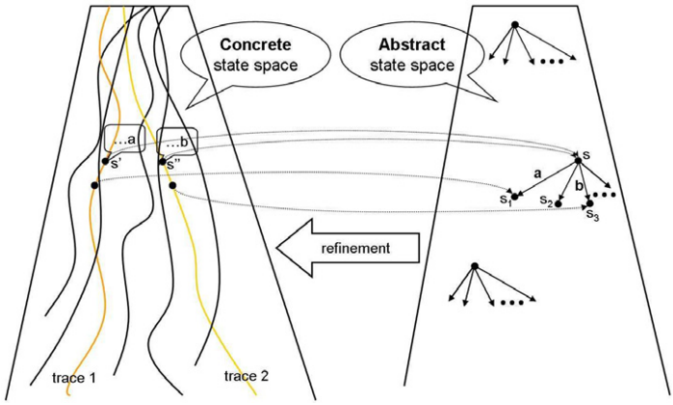

Fig. 10. learn additional information from testing

Suppose that the two concrete states s' and s'' in the system implementation map to the same abstract state s in the system model. If there exist a next state of s' mapping to s_1 and a next state of s'' mapping to s_3, then we add the extra information a in s' and b in s'' to the transitions (s, s_1) and (s, s_3) respectively. In this way, whenever s' (or s'') is monitored, the runtime verification can reduce the state space by only selecting the transition with the additional information a (or b). However, adding too much extra information to the system model might increase the complexity of the system model as well as the overhead of the communication between the runtime verification and the system execution. Therefore, we only add the extra information to the critical states in the system model.

3 Related Work

As mentioned in Section 1, the main characteristic of the state-of-the-art runtime verification is to monitor the actual executions of the program under test and then check if the monitored execution traces conform to the required specification.

Typically, [3] presents runtime checking for the behavioral equivalence between a component implementation and its interface specification by writing the interface specification in the executable AsmL so that one can synchronously run the interface specification and the component implementation while monitor if they are equivalent on the observed behaviors; [1] presents runtime certified computation whereby an algorithm not only produces a result for a given input, but also proves that the result is correct with respect to the given input by deductive reasoning; [14] presents runtime checking for the conformance between a concurrent implementation of a data structure and a high-level executable specification with atomic operations by first instrumenting the implementation code to extract the execution information into a log and then executing a verification thread concurrently with the implementation while using the logged information to check if the execution conforms to the high-level specification; [4] presents monitoring-oriented programming (Mop) as a light-weight formal method to check conformance of implementation to specification at runtime by first inserting specifications as annotations at various user selected places in programs and then translating the annotations into an efficient monitoring code in the same target language as the implementation during a pre-compilation stage. Similar to Mop, Temporal Rover [6] is a commercial code generator allowing programmers to insert specifications in programs via comments and then generating from the specifications the executable verification code, which are compiled and linked as part of the application under test. In addition, Java PathExplorer (JPaX) [10] is a runtime verification environment for monitoring the execution traces of a Java program by first extracting events from the executing program and then analyzing the events via a remote observer process.

What's more, [8] extends the usual runtime verification techniques to on-line verify and steer a Discrete Event System (DES) by looking ahead into a partial system model to predict violations and then applying steering actions to prevent them. This method requires that the time delay for the DES to move from the current state to the next state must be long enough so that the runtime checking has sufficient time to explore a partial system model, which is generated after the current state is known.

Recall that our model-based runtime verification explores the system model even before the current state is known and then shrinks the state space after the current state is known. That is, the progress of our runtime verification is not strictly bound to the execution of the system implementation, i.e., it may run before or after the system execution. As long as the processing speed is fast enough, the runtime verification could keep running certain time steps before the system execution and then tell the system how many time steps ahead is safe. Also, our runtime verification can check more general properties specified by ACTL and/or LTL formulas.

4 Conclusion

Validation and verification are widely-accepted techniques to ensure the correctness of a system under development. Verification checks the correctness of the system at the model level while validation checks at the implementation level. State-of-the-art runtime verification combines verification with system execution, aimed at checking the consistency of a monitored execution trace against the system specification by passively observing an actual execution trace while the system is running. Notice that in this approach even if the monitored execution trace is checked correct with respect to the system properties, it does not mean that this monitored execution trace really conforms to the system model and the system model satisfies the same properties as well.

As an extension to this runtime verification, our model-based runtime verification takes the system implementation, the system model and the system specification together into account. The basic idea is to check at runtime whether the monitored execution trace of the system conforms to the system model on the one hand and whether a partial system model that covers the monitored states satisfies the system properties on the other hand. Due to its working at the model level, our runtime verification can make progress even if the current state of the system running is not observed yet. In fact, the monitored states are used in consistency checking and then used to shrink the state space to be checked by the runtime verification. Therefore, the model-based runtime verification may run before or after the system execution as if there exists a competition between them. In this sense, our model-based runtime verification is more flexible than state-of-the-art runtime verification.

Of course, we have to pay some cost for this flexibility. The computational complexity of the model-based runtime verification is less than that of the offline model checking but greater than that of the state-of-the-art runtime verification. However, our experience shows that it is good enough to have the model-based runtime verification look ahead only several time steps at each checking round, which make its computational complexity in terms of time and memory overhead much closer to that of the state-of-the-art runtime verification.

In order to speed up our runtime verification so that it can take the leading position against the system execution for as long time as possible, two possible strategies are presented for this runtime verification to intelligently reduce the state space to be explored whenever needed by making the system model enriched with probabilities and additional information derived and learned at the system testing phase. Considering that the property patterns to be checked in practice are usually not very complex [7], we believe that the model-based runtime verification can be used to check more general properties specified by ACTL and/or LTL formulas.

References

[1] Arkoudas, K. and M. Rinard, *Deductive Runtime Certification*, in: *Proceedings of the 2004 Workshop on Runtime Verification (RV 2004)*, Barcelona, Spain, 2004.

[2] Ausiello, G. and G. F. Italiano, *On-line algorithms for polynomially solvable satisfiability problems*, J. Log. Program. **10** (1991), pp. 69–90.

[3] Barnett, M. and W. Schulte, *Spying on components: A runtime verification technique*, in: G. T. Leavens, M. Sitaraman and D. Giannakopoulou, editors, *Workshop on Specification and Verification of Component-Based Systems*, 2001.

[4] Chen, F. and G. Rosu, *Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation*, in: *Proceedings of the 2003 Workshop on Runtime Verification (RV 2003)*, Boulder, Colorado, USA, 2003.

[5] Clark, E. M., O. Grumberg, Jr and D. A. Peled, "Model Checking," MIT Press, 1999.

[6] Drusinsky, D., *The Temporal Rover and the ATG Rover*, in: *SPIN*, 2000, pp. 323–330.

[7] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Patterns in property specifications for finite-state verification*, in: *ICSE '99: Proceedings of the 21st international conference on Software engineering* (1999), pp. 411–420.

[8] Easwaran, A., S. Kannan and O. Sokolsky, *Steering of discrete event systems: Control theory approach*, Electr. Notes Theor. Comput. Sci. **144** (2006), pp. 21–39.

[9] Frank, U., H. Giese, F. Klein, O. Oberschelp, A. Schmidt, B. Schulz, H. Vöcking and K. Witting, "Selbstoptimierende Systeme des Maschinenbaus - Definitionen und Konzepte," Number Band 155 in HNI-Verlagsschriftenreihe, Bonifatius GmbH, Paderborn, Germany, 2004, first edition.

[10] Havelund, K. and G. Rosu, *Java PathExplorer — a runtime verification tool*, in: *Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS'01)*, Montreal, Canada, 2001.

[11] Kent, S., *Model driven engineering*, in: *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods* (2002), pp. 286–298.

[12] Schuppan, V. and A. Biere, *Efficient reduction of finite state model checking to reachability analysis*, International Journal on Software Tools for Technology Transfer (STTT) **5** (2004), pp. 185–204.

[13] Shukla, S., D. J. Rosenkrantz, H. B. Hunt III and R. E. Stearns, *A HORNSAT Based Approach to the Polynomial Time Decidability of Simulation Relations for Finite State Processes*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society **35** (1997).

[14] Tasiran, S. and S. Qadeer, *Runtime Refinement Checking of Concurrent Data Structures*, in: *Proceedings of the 2004 Workshop on Runtime Verification (RV 2004)*, Barcelona, Spain, 2004.

[15] Zhao, Y., S. Oberthür, M. Kardos and F.-J. Rammig, *Model-based runtime verification framework for self-optimizing systems*, Electr. Notes Theor. Comput. Sci. **144** (2006), pp. 125–145.

[16] Zhao, Y., S. Oberthür and F. Rammig, *Runtime model checking for safety and consistency of self-optimizing mechatronic systems*, in: *Proceedings of the 7th International Heinz Nixdorf Symposium: Self-optimzing Mechatronic Systems*, Paderborn, Germany, 2008.