On-Line Model Checking as Operating System Service*

Franz J. Rammig, Yuhong Zhao, and Sufyan Samara

Heinz Nixdorf Institute, University of Paderborn
Fürstenallee 11, D-33102 Paderborn, Germany
franz@upb.de

Abstract. A complementary verification method for real-time application with dynamic task structure has been developed. Here the real-time application is developed by means of Model-Driven Engineering. The basic verification technique is given by model checking. However, the model checking is executed at run-time whenever some reconfiguration of the task set takes place. Instead of exploring the entire state space of the model to be checked, only a partial state space at model level covering the execution trace of the checked task is explored. This on-line model checking can be seen as an extension to the traditional schedulability acceptance test which is needed anyway in systems with dynamic task set. Therefore this runtime verification is implemented as a service of the underlying operating system. In this paper we describe this method in general, explain some design and implementation decisions and provide experimental results.

Keywords: On-line model checking, Verification service, Real-time operating system.

1 Introduction

Real-time applications are safety critical in many cases. A careful quality assurance process therefore is mandatory. This process includes more and more formal verification techniques like model checking. Model checking has the advantage of being fully automated and inherently includes means for diagnosis in case of errors. On the other hand, model checking is substantially confronted with the so called state explosion problem. This means that the state space to be explored grows very quickly to an unmanageable size whenever problems of practical relevance are to be handled. Numerous approaches to overcome this deficiency have been developed, like partial order reduction [1], compositional reasoning [2], and other simplification and abstraction techniques, which aim to reduce the state space to be explored by *over-approximation* [3] or *under-approximation*

* 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).

S. Lee and P. Narasimhan (Eds.): SEUS 2009, LNCS 5860, pp. 131–143, 2009.
© IFIP International Federation for Information Processing 2009

[4] techniques. Over-approximation techniques generate an abstract model by adding redundant behaviors to the original one (weaken constraints) such that the correctness at the abstract level implies the correctness of the original model. Under-approximation techniques generate an abstract model by removing irrelevant behaviors from the original one (strengthen constraints) so that the falseness at the abstract level implies the falseness of the original model. Applying these techniques can relieve the state explosion problem to some degree, but can not resolve it totally. That is, the correctness of a complex system with respect to some properties could not always be verified completely.

In this paper we propose a complementary technique, namely on-line model checking (or model-based runtime verification) [5,6]. Deferring formal verification to the execution phase of a real time application seems to be a strange idea, especially in real-time computing where one prefers to execute off-line as many activities related to a task as possible. However, we are looking at real-time applications with a highly dynamic task set. For a software system with self-adaptive capability, the task set consists of instances that are activated under various profiles. It is based on the actual environmental conditions to decide which profile to be used, i.e, which tasks to be activated. In such real-time applications with dynamic task sets an acceptance test concerning schedulability has to be executed whenever a new task is added to the task set. It seems to be natural to extend this acceptance test by a logical safety test, which may be implemented by means of model checking. But we would be confronted with the state explosion problem again, now even under real-time constraints.

To make on-line model checking feasible, we suppose that the real-time application is developed by means of Model-Driven Engineering (MDE) [7], which is an efficient software engineering approach to complex systems development. According to MDE, we can follow three steps to develop a software system:

1. model the system according to the system specification,
2. verify the system model against the system specification, and
3. synthesize the system implementation (source code) from the system model.

Theoretically speaking, the following assertions are supposed to be *true*:

- The system model is consistent with the system specification.
- The system implementation is consistent with the system model.

However, are they really true under any specific running environment? We try to answer this question by doing model checking at runtime. The basic idea (as shown in Fig. 1) is to check on-line whether the monitored execution trace of the system

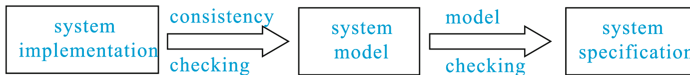

Fig. 1. On-line model checking framework

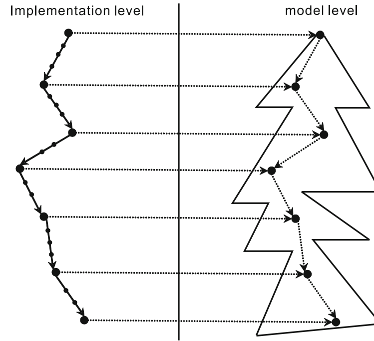

Fig. 2. Partial system model to be explored

conforms to the system model on the one hand and if a partial system model that covers the execution trace satisfies the system properties on the other hand.

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. 2. Intuitively, if this partial system model is checked *safe* 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.

Of course, sophisticated techniques have to be used to let it really fly, which will be detailed in the sequel. In this way, we obtain a natural solution to the state explosion problem. Instead of looking at the entire state space, we pay our attention only to a partial state space covering the execution trace. As a result, we do not need to simplify or abstract value domains of system variables at all. It is worth mentioning that off-line model checking is usually valid under the assumption that the platform, on which the real-time application runs, should behave correctly. This assumption is no longer needed for on-line model checking.

Commonly used services at run-time are usually provided by the underlying Operating System. This is exactly our approach. We provide on-line model checking as a service of the underlying Real-time Operating System (RTOS). The verification service is implemented as isolated task in user space. This isolates model checking from the task to be verified and makes sure that errors in the task cannot infect the verification service. To enhance efficiency the verification service runs in its own address space which is attached to the kernel address space. The address space of the application is mapped into this verification address space as "read only" partition. This avoids cache refilling in case of the context switching between verification service and the task to be checked and allows fast access to the task's state variables by the verification service.

The state-of-the-art runtime-verification is discussed in the literature (see section 4) since years. The basic idea is to monitor the execution of the source code and afterwards to check the so far observed execution trace against the given properties specified usually by LTL formulas. The checking progress always falls behind the system execution because the checking procedure can continue only after a new state has been observed. In contrast, our runtime verification is applied to the model level. The states observed from the execution trace are mainly used to reduce the state space to be explored at the model level. That is, the checking progress is not strictly bound to the progress of the system execution, i.e., our on-line model checking might run ahead or behind the execution of the source code. If the processing speed is fast enough, our runtime verification could keep looking certain time steps ahead of the system execution and then tell the real-time application how many time steps ahead are safe.

2 Problem Statement

Without loss of generality, let $M = \{M_1, M_2, \cdots, M_n\}$ model a real-time reconfigurable system which consists of n (> 0) components $M_1, M_2, \cdots, M_n$ running in parallel. M may reconfigure itself at runtime either by adding a new component M_i' to or by removing an existing component M_i from M. This also includes replacing one component with another one as shown in Fig. 3, which can be done by consecutively removing and adding operations.

The components in M can communicate with each other only through the underlying RTOS. This forms a dependency relationship between the components in M. Without doubt, system reconfiguration might more or less affect the behavior of the related components in the system. What's more, the impact of the RTOS on the inter-process communication also might affect the behaviors of the related components in the system. For instance, the component B might be affected most by replacing the component C with the component E in Fig. 3. Could these effects violate some safety conditions associated to the related components in the system?

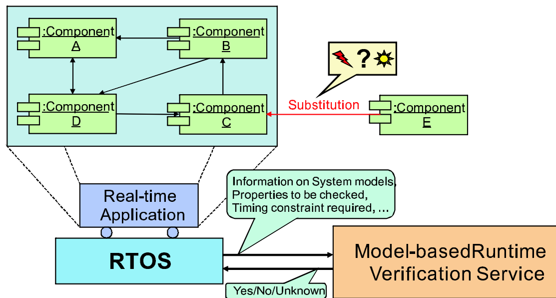

Fig. 3. A reconfiguration example

Since the reconfiguration might occur according to the actual running environment, it is hard to answer this question only by off-line verification techniques due to the unpredictable indefinite factors. Therefore, it is necessary to on-line check at model level if the most affected component in M still maintains safety after the system reconfiguration.

In doing so, we suppose that whenever the real-time application needs to do reconfiguration, the RTOS is informed about this in advance. As any modification of the task set can happen only under the control of the RTOS, this requirement is rational. With the information given by the real-time application, the RTOS will trigger the verification service as isolated task in user space and then schedule the verification service as earlier than the component to be checked as possible without violating the real-time deadlines.

To achieve this, we follow a deterministic approach by reserving a fixed time slot at the beginning of each scheduling cycle of the RTOS. This time slot is mainly reserved for the verification service. In case of no active on-line model checking task, the scheduler is allowed to allocate this slot to such preemptive low priority tasks that can be moved and replaced by on-line model checking at any time the verification service is triggered. In this way, if the checking process is efficient enough, we can always check at model level what might happen in the near future relative to the current state of the component's execution. In case that an error is detected or the checking progress falls behind the execution of the checked task, then the real-time application is informed in order to allow it undertaking appropriate counter means.

The properties to be checked are safety conditions that might be sensitive to the context of the related component. LTL (or ACTL) formulas are used to formally specify the safety properties, as the discrete time extensions to LTL (or ACTL) formulas are just shorthand notations to the usual LTL (or ACTL) formulas [8].

3 On-Line Model Checking

3.1 Overview

As mentioned in Section 1, we suppose that the real-time application is developed following the MDE approach. In this way, we can model the system in UML with real-time extension[1] on the one hand and specify constraints in OCL with real-time extension on the other hand. From the real-time UML model, we can derive an FSM model and synthesize a source code respectively. Since the FSM model and the source code come from the same origin, there exists a mapping function σ from concrete states (derived from the source code) to abstract states (derived from the FSM model). From the real-time OCL constraints, we can derive the LTL (or ACTL) formulas and then transform them into *Büchi* automata. Having the concrete model (source code), the abstract model (FSM model) and the properties (*Büchi* automata) at hand, our on-line model checking aims to check

[1] http://wwwcs.uni-paderborn.de/cs/fujaba/

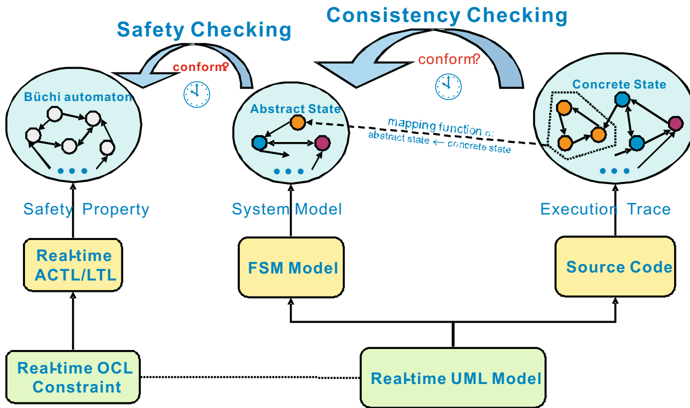

Fig. 4. Overview

if the execution trace conforms to the FSM model (consistency checking) and meanwhile if a partial state space of the FSM model conforms to the *Büchi* automaton (safety checking) as shown in Fig. 4. Here the partial state space reflects a near future relative to the current state observed from the execution trace of the system running.

3.2 Model Checking Paradigm

Recall that we have reserved a fixed time slot at the beginning of each scheduling cycle for on-line model checking. Without loss of generality, let the time slot be t_d time units. After verification service is triggered, in each scheduling cycle we have t_d time units to do on-line model checking from the current state of the task to be checked, which is obtained at the previous scheduling cycle as shown in Fig. 5. Of course, the current (concrete) state should be mapped to the corresponding abstract state at model level to be used by model checking. If the current state could not be mapped to an appropriate abstract state, it means that the execution trace no longer conforms to the behavioral model. In this case, the verification service will terminate the checking process and inform the RTOS to deal with this problem. Otherwise, the on-line model checking will continue until one of the following two cases happens:

Case *No*: if at some time point an error is detected, the verification service terminates with the answer *No* to the real-time application via RTOS.

Case *Yes*: if a sufficient partial state space that covers the execution trace of the task is successfully checked, the verification service reports definitely *Yes* to the real-time application via RTOS and then terminates the safety checking process (while the consistency checking can continue if necessary).

Notice that the "*No*" case only means that the detected errors might happen in the future, because we check at model level and thus do not know whether

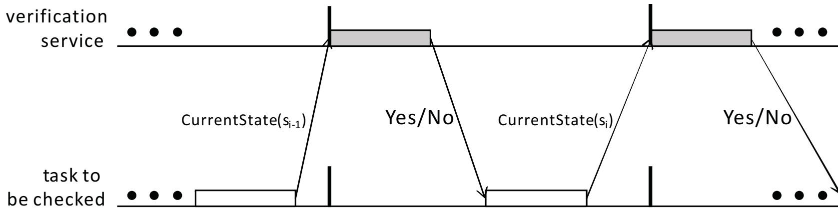

Fig. 5. Scheduling of verification service and task to be checked

the errors are spurious or not. To avoid the errors really to happen, we have to conservatively choose to 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 verification is conservative due to its being applied to the model level. However, the advantage of predicting and thus avoiding potential errors are gained just due to its being applied to the model level.

Experimental Results. A stand-alone prototype for on-line model checking invariants, LTL and ACTL properties is implemented. We have done some experiments for BEEM[2] benchmark set derived from mutual exclusion algorithms, communication protocols and so on in research or industry area. The benchmark set contains only FSM models, so we generate randomly the execution traces from the same FSM models. This can simplify the monitoring procedure of capturing the runtime information (current states) to be used by on-line model checking. In this way, we can estimate the performance of our verification service to some degree. Two experiments are done on a Pentium-IV 3.00Ghz processor with 1GB memory running Linux.

One experiment is on-line invariant checking. This experiment can help find out the influence of the out-degrees of the states on the look-ahead performance, i.e., how far away the model checking can look ahead from each state of the given model within a predefined time interval. 16 typical models are selected from the BEEM benchmark set to perform on-line invariant checking. The features of these models are given in the number of states, the number of transitions, the average degrees of states, the height of BFS, and the maximal stack of DFS as well as the number of *Boolean* (state) variables. In this experiment, each transition in the models is set to represent 1 millisecond, i.e., it takes 1 millisecond from one state to next state. We also say one transition being one time step. For each model, this

[2] http://anna.fi.muni.cz/models/

Model	Type	State	Transition	Average Degree	Maximal Out-degree	BFS Height	Max. Stack	Boolean Variables	Transition timeunit (ms)	Minimal Look-ahead	Maximal Look-ahead	Average Look-ahead
sorter_1	Controller	20544	30697	1,5	5	198	617	36	1	40	299	103
collision_1	Communications protocol	5593	10792	1,9	5	57	617	25	1	26	81	48,7
synapse_2	Protocol	61048	125334	2,1	18	41	2349	46	1	7	28	21,5
driving_phils_2	Mutualexclusion algorithm	33173	81854	2,5	9	150	3702	27	1	31	97	65,7
blocks_1	Planningand Scheduling	7057	18552	2,6	6	19	4263	23	1	8	21	14
peterson_1	MutualExclusion Algorithm	12498	33369	2,7	5	54	1862	30	1	13	39	31,7
szymanski_1	MutualExclusion Algorithm	20264	56701	2,8	3	72	2064	27	1	13	90	49,7
hanoi_1	Puzzle	6561	19680	3	3	256	4376	36	1	56	103	75,9
iprotocol_2	Communications protocol	29994	100489	3,4	7	91	443	39	1	18	451	50
phils_3	MutualExclusion Algorithm	729	2916	4	6	17	518	18	1	156	357	265
cyclic_scheduler_1	Protocol	4606	20480	4,4	8	55	1819	40	1	23	437	278
rushhour_1	Puzzle	1048	5446	5,2	9	73	535	28	1	66	248	150,7
rushhour_2	Puzzle	2242	12603	5,6	10	80	906	32	1	36	408	116,4
pouring_1	Puzzle	503	4481	8,9	9	13	348	16	1	42	101	71,9
reader_writer_2	Protocol	4104	49190	12	19	13	4097	25	1	4	16	9,9
pouring_2	Puzzle	51624	1232712	23,9	25	15	44509	18	1	1	4	2

Fig. 6. Experimental result of on-line invariant checking

experiment is designed to compute how many time steps model checking could look ahead from each state in the model within one time step (i.e. 1ms). So the invariant to be checked is a *Boolean* formula derived from the set of the states in each model. The experimental results in Fig. 6 show the minimal, the maximal and the average look-ahead from the states of each model. It is easy to see that the maximal out-degree of a model has a larger influence on look-ahead performance than the average degree of the model.

The other experiment is on-line LTL model checking. The model driving_phils_2 is derived from a mutual exclusion algorithm of processes accessing several resources, motivated by "The Driving Philosophers" in [9]. The property to be checked is $G(ac_0 \rightarrow Fgr_0)$, where the proposition ac_0 denotes that process 0 requests a resource and the proposition gr_0 denotes that the resource is granted to process 0. In other words, if process 0 requests a resource, it will be granted to him eventually. The experimental result in Fig. 7 is obtained by setting $t_d = 5ms$ and running 2000 scheduling cycles. That is, at each scheduling cycle the verification service is allocated 5ms to perform on-line model checking. The property is not violated at least up to this 2000 checking rounds. Fortunately, the verification service can always run enough time steps earlier than the simulated execution of this model. The minimal look-ahead is 23 time steps, the maximal look-ahead is 74 time steps and the average look-head is 57.2 time steps relative to the corresponding current states monitored from the randomly generated execution trace.

Compared to the usual (off-line) model checking, our on-line model checking 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 on-line model checking is less than that of the traditional model checking. Compared to the usual runtime verification, our runtime verification checks the

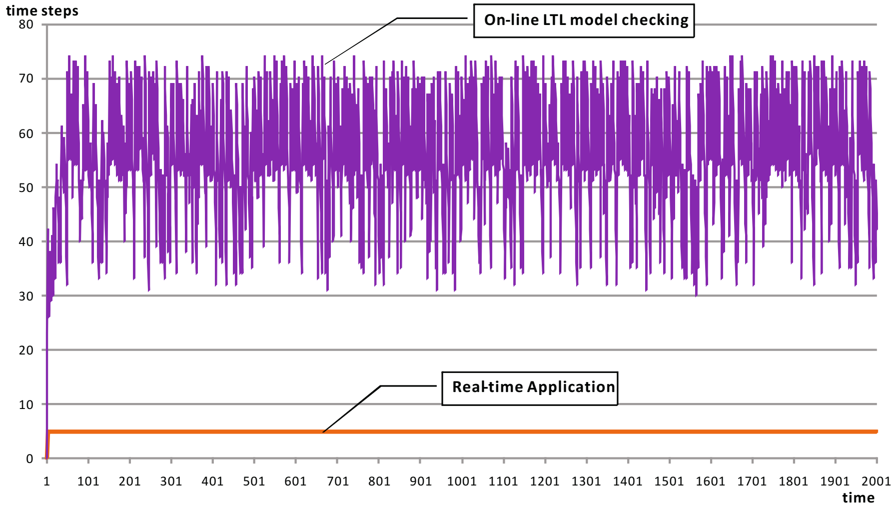

Fig. 7. Experimental result of on-line LTL checking

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 [10] shows that the property patterns to be checked in practice are usually not very complex.

3.3 Pre-checking and Post-checking

Ideally, we wish that on-line model checking could always run enough (time) steps ahead the execution of the task to be verified. This depends on the complexity of the behavioral model of the task as well as the underlying hardware architecture. Therefore, we have to face the reality that the verification service might fall behind the execution of the task to be checked. As a result, we introduce two checking modes: *pre-checking* and *post-checking*. We say that the verification service is in pre-checking mode, if it runs ahead of the execution of the task to be checked; otherwise, it is in post-checking mode as shown in Fig. 8.

In pre-checking mode, the verification service can naturally predict violations before they really happen. In post-checking mode, it seems that the violations could only be detected after they have already happened. Fortunately, it is still possible to "predict" violations even in post-checking mode because our on-line verification works at the model level. In case that an error is found at some place

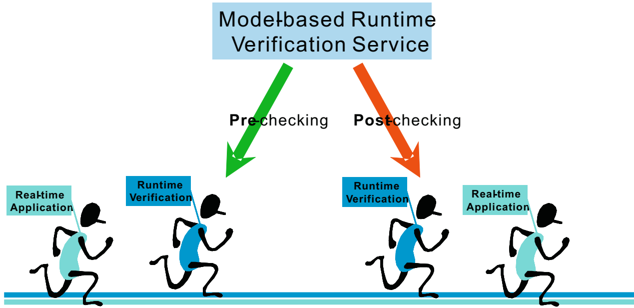

Fig. 8. Pre-checking and Post-checking

other than the monitored execution trace in the partial state space being checked, then we can "predict" that there might be an error in the model which has not happened yet. In this sense, both checking modes are useful for safety-critical systems.

Notice that our on-line model checking can observe the actual execution trace of the task being checked once it falls behind. This means that only a rather small state space needs to be explored in post-checking mode. Thus, there still exists chance for the verification service to pass over the task being checked again. On this view, it seems as if the verification service and the task are involved into a two-player game. In the course of the game, we say that the verification service *win*s against the task being checked, if the verification service takes the leading position for a longer time than the task does.

Without doubt, we need to find an improved strategy to make the verification service have more chance or higher probability to win against the task to be checked. Recall that the source code of the system implementation is usually validated by simulation and testing. Therefore, in the future we are going to learn some heuristic knowledge at the system testing phase so that the system model can be enriched with more useful information. The heuristic information can thus guide on-line model checking to reduce the state space to be explored whenever necessary.

4 Related Work

Unlike our on-line model checking, the state-of-the-art runtime verification takes the system implementation and the system specification into account. The basic idea is to monitor the execution of the source code and afterwards to check the so far observed execution trace against the system properties specified usually by LTL formulas. 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 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 a property is 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 property 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 property automaton to be checked.

Typically, [11] 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; [12] 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; [13] 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; [14] 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 [15] 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) [16] 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, [17] 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.

Our on-line model checking can explore 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 source code, i.e., it may run before or after the system execution. If the processing speed is fast enough, our runtime verification could keep running certain time steps before the system execution and then tell the system how many time steps ahead are safe. Also, our runtime verification can check more general properties specified by ACTL and/or LTL formulas.

5 Conclusion

On-line model checking has the potential to serve as a powerful complementary verification technique for real-time applications with dynamic task sets. It is complementary in the sense that we have to assume that the newly accepted task has been verified off-line under the assumptions necessary for such a verification. The on-line model checking then can be restricted to verify whether the actual execution trace is correct under the real environmental conditions. As our on-line model checking reduces dramatically the state space to be verified, a much finer granularity concerning value domains of system variables can be handled. By this and due to the fact that *a priori* unknown run-time conditions can be considered as well, our run-time verification establishes an additional safety level. This method can also be seen as a complementary attempt to overcome the well known state explosion problem of model checking. Whenever the state space is reduced, it is essential to reduce it to the states that are relevant. Our method automatically and dynamically reduces the state space to exactly those states that are relevant for the actual execution trace. The resulting verification method can be implemented as an operating system service comparable to the schedulability acceptance test which is part of any RTOS being able to handle dynamic task sets. This service is triggered whenever some reconfiguration of the task set to be handled takes place. In contrary to traditional *a posteriori* runtime verification methods published so far, our approach can look into the future, i.e., a partial state space at model level relative to the current state of the execution trace. Experimental results show that run-time model checking is possible when the approach as outlined in this paper is followed. Although these experiments have been carried out based on simulations up to now, there is a strong indication that also systems of practical relevance can be handled.

References

1. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996); Foreword By-Wolper, Pierre
2. Berezin, S., Campos, S.V.A., Clarke, E.M.: Compositional reasoning in model checking. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 81–102. Springer, Heidelberg (1998)
3. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16(5), 1512–1542 (1994)
4. Lee, W., Pardo, A., Jang, J.Y., Hachtel, G., Somenzi, F.: Tearing based automatic abstraction for ctl model checking. In: ICCAD 1996: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, pp. 76–81. IEEE Computer Society, Los Alamitos (1996)
5. Zhao, Y., Oberthür, S., Kardos, M., Rammig, F.J.: Model-based runtime verification framework for self-optimizing systems. Electr. Notes Theor. Comput. Sci. 144(4), 125–145 (2006)
6. Zhao, Y., Rammig, F.J.: Model-based runtime verification framework. In: Proceedings of the Formal Engineering Approaches to Software Components and Architectures (FESCA 2009), New York, UK (March 2009)

7. Kent, S.: Model driven engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
8. Clark, E.M., Grumberg Jr., O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
9. Baehni, S., Baldoni, R., Guerraoui, R., Pochon, B.: The driving philosophers. Technical report. In: Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science (TCS 2004) (2004)
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE 1999: Proceedings of the 21st international conference on Software engineering, pp. 411–420. IEEE Computer Society Press, Los Alamitos (1999)
11. Barnett, M., Schulte, W.: Spying on components: A runtime verification technique. In: Leavens, G.T., Sitaraman, M., Giannakopoulou, D. (eds.) Workshop on Specification and Verification of Component-Based Systems (October 2001)
12. Arkoudas, K., Rinard, M.: Deductive Runtime Certification. In: Proceedings of the 2004 Workshop on Runtime Verification (RV 2004), Barcelona, Spain (April 2004)
13. Tasiran, S., Qadeer, S.: Runtime Refinement Checking of Concurrent Data Structures. In: Proceedings of the 2004 Workshop on Runtime Verification (RV 2004), Barcelona, Spain (April 2004)
14. Chen, F., Rosu, G.: 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)
15. Drusinsky, D.: The Temporal Rover and the ATG Rover. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 323–330. Springer, Heidelberg (2000)
16. Havelund, K., Rosu, G.: Java PathExplorer — a runtime verification tool. In: Proceedings 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space (ISAIRAS 2001), Montreal, Canada (June 2001)
17. Easwaran, A., Kannan, S., Sokolsky, O.: Steering of discrete event systems: Control theory approach. Electr. Notes Theor. Comput. Sci. 144(4), 21–39 (2006)