A Monitoring and Checking Framework for Run-time Correctness Assurance

Insup Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, M. Viswanathan
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

ABSTRACT

Computer systems are often monitored for performance evaluation and enhancement, debugging and testing, control or to check for the correctness of the system. Recently, the problem of designing monitors to check for the correctness of system implementation has received increased attention from the research community. Traditionally, verification has been used to increase the confidence that a system will be correct by making sure that a design specification is correct. However, even if a design has been formally verified, it still does not ensure the correctness of an implementation of the design. This is because the implementation often is much more detailed, and may not strictly follow the formal design. So, there is possibility for introduction of errors into an implementation of a design that has been verified. One way that people have traditionally tried to overcome this gap between the design and the implementation has been to test the implementation's behavior on a pre-determined set of input sequences. This approach, however, fails to provide guarantees about the correctness of the implementation on all possible input sequences. Consequently, when a system is running, it is hard to guarantee whether the current execution of the system is correct or not using the two traditional methods. Therefore, the approach of continuously monitoring a running system has received much attention, as it attempts to overcome the difficulties encountered by the two traditional methods for checking the correctness of the current execution of the system.

We describe a framework that provides assurance on the correctness of program execution at run-time. This approach is based on the Monitoring and Checking (MaC) architecture, and complements the two traditional approaches for ensuring that a system is correct, namely static analysis and testing. Unlike these approaches which try to ensure that all possible executions of the system are correct, this approach ensures only that the current execution of the system is correct. The MaC architecture consists of three components: filter, event recognizer, and run-time checker. The filter extracts low-level information, in the form of values of program variables, from the instrumented system code, and sends it to the event recognizer. From this low-level information, the event recognizer detects the occurrence of abstract events, and informs the run-time checker about these. The run-time checker then, based on the events, checks the conformance of the behavior of the system on the current execution, to the formal requirement specification for the system.

Acknowledgment. This research was supported in part by NSF CCR-9415346, NSF CCR-9619910, AFOSR F49620-95-1-0508, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, and ONR N00014-97-1-0505 (MURI). The current address of Ben-Abdallah is at Department of d'Informatique, Universite de Sfax, Tunisia.

INTRODUCTION

Computer systems are often monitored for performance evaluation and enhancement, debugging and testing, control or check of system correctness [Sch95]. Recently, the problem of designing monitors to check for the correctness of system implementation has received increased attention from the research community [CG92,SM93,ML97,Sch98]. Such monitors can be used to detect violations of timing [ML97], a logical property of a program [CG92], a constraint on a language construct [SM93], and so on.

The reason for increased interest in correctness monitors is that it is becoming more difficult to test or verify software because software size is increasing and its functionality is becoming more complicated. With increasing size of software, it is infeasible to completely test the entire system because there are too many possibilities. Also, as the functionality and structure of software becomes complex in order to satisfy a broad range of needs, testing needs to be sophisticated enough to check the program according to diverse criteria. For example, for testing a numerical computation, it is enough to check output with given input. However, when we test a real-time application like traffic control system, we also have to check the timing behavior.

Traditionally, verification has been used to increase the confidence that a system will be correct by making sure that a design specification is correct. However, even if a design has been formally verified, it still does not ensure the correctness of an implementation of the design. This is because the implementation often is much more detailed, and may not strictly follow the formal design. So, there are possibilities for introduction of errors into an implementation of a design that has been verified. One way that people have traditionally tried to overcome this gap between the design and the implementation has been to “test” the implementation's behavior on a pre-determined set of input sequences. This approach, however, fails to provide guarantees about the correctness of the implementation on *all* possible input sequences. Consequently, when a system is running, it is hard to guarantee whether the current execution of the system is correct or not using the two traditional methods. Therefore, the approach of continuously monitoring a running system has received much attention.

In this paper, we describe a framework of monitoring and checking a running system with the aim of ensuring that it is running correctly. The salient aspect of our approach is the use of formal requirements specification to decide what properties to assure. Since our goal is to check an implementation against requirements specification at run-time, we assume that we are given both requirement specifications and an implementation. There are many issues in monitoring. This paper concentrates on the following two issues: (1) How to map high-level abstract events that are used in requirement specification to low-level activities of a running system. (2) How to instrument code to extract and detect necessary low-level activities.

Figure 1 shows the framework, which consists of the three phases: the design phase, the implementation/instrumentation phase, and the run-time phase. During the design phase, the requirements on the system are specified. Optionally, a formal system specification may also be written down and in this case we assume that verification is done to ensure that the system specification satisfies the requirements. During the implementation phase the system is implemented.

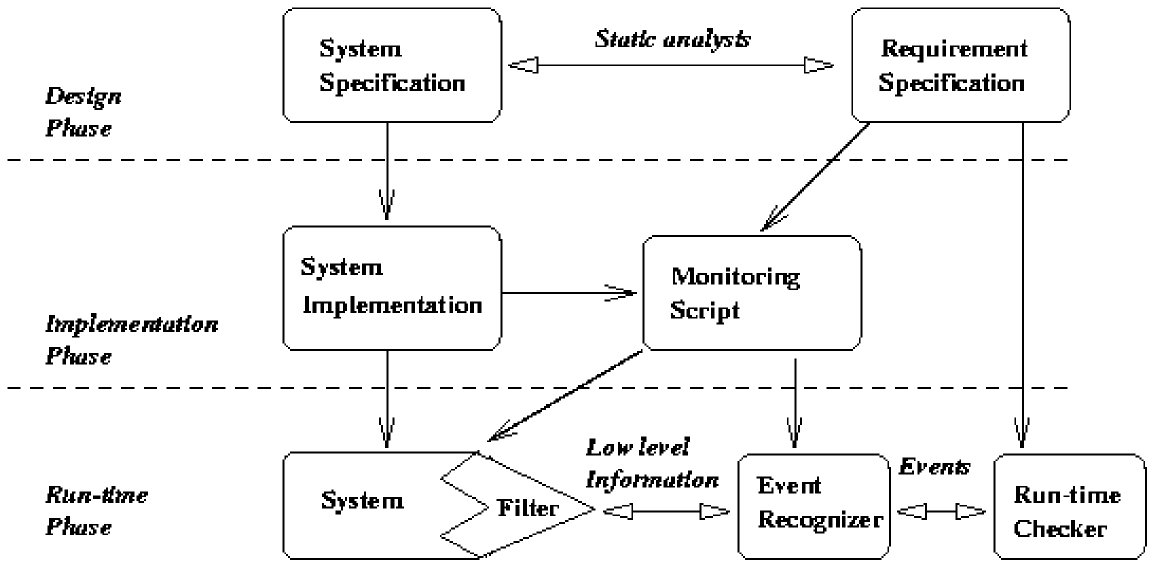

Figure 1. Overview of the MaC Framework

The other main task of this phase is to write down a script called the *monitoring script* that contains instructions for instrumenting the code so that low-level information about program state can be passed on to the monitor. In addition, it also contains information that can be used to produce an *event recognizer* that can transform this low-level information into high-level events.

Our run-time MaC architecture consists of three components: filter, event recognizer, and run-time checker. The *filter* extracts low-level information (namely, values of program variables and time when variables change their values) from the running code. In order to achieve this, we instrument the code of the system to be monitored. The filter sends this information to an *event recognizer*. It is the job of the event recognizer to map this low-level information about the running code to high-level information that the *run-time checker (RTC)* understands. As explained in the previous paragraph, instructions for instrumenting the program as well as for producing an event recognizer are contained in the monitoring script. Based on the values of the monitored variables it receives from the filter, the event recognizer detects the occurrence of events that are described in the requirement specification, and sends this to the run-time checker.

The run-time checker then checks the correctness of the system *thus far*, according to a requirement specification of the system, based on the information of events it receives from the event recognizer, and on the past history. It checks the correctness of a sequence of events seen by stepping through the requirements specification and the correctness of a numerical computation by executing a program checker [BK95] for that function. This is one of the main advantages that our approach offers over past approaches to monitoring: that is, it provides an integrated framework to check general requirements both for transfer of control during execution and for numerical computation.

The current prototype implementation of the MaC architecture monitors programs that are in JAVA bytecode and uses a new language called MEDL which is related to linear temporal logic to describe the formal requirements. Although MEDL is used for requirements specification, it is possible to use other formal languages like ACSR [BGGL93], temporal logic, extended state machines, and Petri net, by making minor modifications to the existing code. Furthermore, this framework can be used to monitor systems, that may or may not have been developed using some formal system specification.

The rest of the paper is organized as follows. The next section motivates and summarizes related research. The paper then gives an overview of the MaC framework and discusses pertinent issues as well as the current prototype under development. The last section outlines current and future work.

MOTIVATION AND RELATED RESEARCH

Traditionally, *verification* and *testing* have been two approaches to trying to ensure the correctness of a program. In formal verification, one describes both the requirements and the system in some formal specification language, and then attempts to prove that the formal specification of the system satisfies the requirements. This ``proof of satisfaction" can either be obtained using a model checker [CES86], which exhaustively checks for satisfaction of the requirement in all possible computation scenarios, or using a theorem prover [Gor88,OSR93], where one shows that the requirement is a logical consequence of the system specification. However, both model checking and theorem proving have limitations. The large size of the explicit representation of the state space of most systems severely limits the size of systems that can be model checked. This is a problem referred to as the *state space explosion* problem. Although a state reduction technique [Kur87] and a symbolic approach [BCD+90,McM93] have been proposed to overcome the state space explosion, fully describing and verifying a system is still hard. Theorem provers suffer from the fact that they cannot provide counter-examples which show that the design is incorrect, and that they are difficult to use, as they require extensive user interaction during proof construction. Most importantly, even if some design has been formally proved to be correct (using a model checker or a theorem prover), it does not mean that the implementation is correct because often the implementation has greater detail, and so is susceptible to errors not present in the design.

Testing attempts to overcome these problems by dealing with the implementation directly. Test cases are generated first, then the system is tested by checking sequences of inputs and outputs on test cases. However, it is hard to generate a test suite of small size that can detect most system faults with high probability. Furthermore, testing does not ensure the correctness of the system on all possible input sequences.

In order to overcome the limitations of both verification and testing, some researchers have tried to systematically derive the implementation of the system from a formal specification [BCG87,Tur96] or generate test cases and oracles using formal specification [DR96,JP97,ClLe97], these approaches have their own limitations. Implementing the system from a formal specification has problems of inefficiency and incompleteness in a sense that only skeleton code of the system can be generated and a human programmer has to be involved to complete implementation, which could lead to errors. In addition, most research on testing with a formal specification [JP97,SS97] focuses on black-box testing. However, the correctness of reactive programs depends not only on the input-output behavior, but also on on-going interactions between components of the system. Some researchers have, therefore, taken another approach, wherein the system code is instrumented and the program is monitored continuously at run-time. The sentry system [CG92] observes the execution of program and determines whether the program is behaving “correctly” with respect to a set of specified logical properties on program variables. Sankar and Mandel have developed a methodology to continuously monitor an executing Ada program for specification consistency [SM93]. The user annotates an Ada program with constructs from ANNA, a formal specification language. But they can check only constraints on Ada constructs such as a subtyping for Ada types. Mok and Liu [ML97] proposed an approach for monitoring the violation of timing constraints written in the specification language based on Real-time Logic as early as possible with low overhead. They derive a set of timing constraints such that for a given timing constraint specification any violation will be caught as soon as it happens by checking the constraint of such a set.

Our approach does not limit the scope of requirements to constraints on language construct nor focus on timing constraints. We monitor general requirements described in a formal specification. In the next section, we explain a run-time assurance monitoring architecture as a solution to the above limitations of prior research.

THE MONITORING AND CHECKING (MaC) PARADIGM

The MaC paradigm supports the run-time assurance monitoring of real-time systems. The paradigm consists of three phases for system development and deployment as shown in Figure 1: the design, implementation and run-time execution phases.

The Design Phase

In the *design* phase, we assume that the system requirements are formally specified. In this phase, a system design may also be formally specified and verified against its requirements specification. Several formal methods are available for this phase, e.g., extended finite state machines, process algebra, Petri nets, temporal logic. They differ in the level of formality, expressiveness, and automated analysis power. Run-time assurance monitoring in MaC can be applied both to a system with a formal specification and one without.

The requirements specification, however, must be given since it is the basis for run-time assurance. Although the system specification may help define events, by which the requirements specification is expressed, in terms of system specific information, the only essential part of the MaC paradigm is the requirements specification.

The Implementation Phase

In the *implementation* phase, we assume that code for the system is developed or derived from the design. Although there has been a progress in automating the derivation of an implementation from a design specification, currently the derivation for complex software produces at best skeleton code that must be manually augmented to complete an implementation. This gap between a design and an implementation in turn creates a gap between requirements and an implementation. The goal of the MaC paradigm is to narrow this gap.

One source of the gap is that requirements are described often in terms of “high-level events,” while code uses only “low-level state” information about execution. In the rest of the paper, we use ”event” to denote “high-level event” and “state” for “low-level” state. In order to monitor and check required properties, the two types of information, i.e., events and states, need to be related and their relations must be specified explicitly in a *monitoring script*. The monitoring script describes how events at the requirements level are defined on the monitored states of an implementation. For example, in a gate controller of a rail-road crossing system, the requirements may be expressed in terms of the event `train_in_crossing`; the implementation, on the other hand, only represents the train's position with respect to the crossing in terms of a metric coordinate `train_position`. The monitoring script in this case can define the requirements event in terms of the value of the (implementation) variable `train_position`, for example, as the time instant when `train_position < 800`.

Ideally, the monitoring script should also be used to generate a *filter* and an *event recognizer* automatically. A filter is a collection of code fragments that is used to instrument an implementation to monitor necessary state information at run-time and an *event recognizer* determines the occurrences of events using the information provided by the filter. We note that the monitoring script language depends on both the requirements and the implementation languages used.

The major part of a monitoring script language is to define events. There is a tradeoff between the expressiveness of event definitions and the run-time cost of event detection. The language can be designed to allow very expressive event definitions so that violation of any requirements property is itself an event. However, the drawback of an expressive event definition language is the cost of event detection at run-time. In general, as the language becomes more expressive, the granularity of detection needs to be finer, which incurs more overheads in both time and space. However, if the language has limited expressiveness, it cannot define some useful events. For example, if an event definition language cannot express *i*th occurrence of event, we cannot define 5th `InCrossing` of the train. We later show how our current prototype employs a two-level approach, with PEDL and MEDL. PEDL ensures that it is possible to have an efficient filter and event recognizer, whereas MEDL provides expressiveness. Furthermore, the implementation language specifics are restricted to PEDL.

The Run-time Phase

At run-time, the instrumented implementation is executed while being monitored and checked against the requirements specification. As Figure 1 shows, while the instrumented implementation is executed, the filter sends relevant state information to the *event recognizer* that determines the occurrence of high-level events. These events are then relayed to the *run-time checker* to check adherence to the requirements. If and when it is determined that the system is in an unsafe state, there should be a way to provide feedback to steer the system into a desirable state.

Filter. A *filter* is a set of program fragments that will be inserted into the implementation to instrument the system. The essential functionality of a filter is to keep track of monitored (implementation) variables and send pertinent state information to an event recongnizer as described in the monitoring script. This part of the MaC framework depends on the implementation language used. In addition, it must resolve the following four issues.

- **When to instrument.** The filter can be added to the implementation either statically or dynamically. Static instrumentation is to insert the filter in the implementation before the system is executed, whereas dynamic instrumentation is insert the filter in the implementation at run-time. The advantage of dynamic instrumentation is flexibility, for example, where and what to insert can be determined at run-time from the intermediate result of the execution (e.g., [MC95]). However, it may incur extra overhead at run-time to determine when it is safe to insert and remove filters. Also, it requires a complex instrumentation mechanism. Our current prototype uses static instrumentation.

- **What to instrument.** The filter can be inserted at the source-code level or the excutable-code (e.g., bytecode) level. Compared to the source-code level instrumentation, the executable-code level instrumentation is complicated since the source-level information useful for understanding the program. In addition, modifying a system at executable-code level requires modification not directly related to monitoring, but necessary to keep the format of executable code consistent. The major advantage, however, is that this executable-code level of instrumentation can be applied to a broad range of target systems because executable code is always available for running a system. Assuming we have some limited source-code level information on the system such as meaning and names of monitored variables and functions, we can instrument executable code of systems without referring to the source code. Furthermore, low level operations on executable code can provide a fine granularity and an easy mechanism for detecting change of variables. For the above reasons, our prototype instruments the bytecode (i.e., executable code) of the system.
- **How to instrument.** Instrumentation can be done automatically or manually. In a manual instrumentation, which is more common (e.g. [ES94]), a user reads a source code of the program, then inserts filters (or probes) into the system. This can be done efficiently because it uses a heuristic and domain knowledge to pinpoint where to instrument and what state information to extract. However, its heuristic character may result in incompleteness. For assurance monitoring, instrumentation should be *complete* in the sense that it should capture all interesting information. Missing information could lead to false or missed detection of faults. Automatic instrumentation determines what filters/probes should be inserted where in a mechanical way based on the definition of the event and the structure of the program. The weak point of mechanical instrumentation is that it is not easy to define an event of high level behavior based on low level state information. However, once the event definition is provided in terms of low-level monitored information (as it is in the case of our current prototype), it provides a systematic way of instrumentation which automates the instrumentation. Therefore, our prototype system instruments a system automatically.
- **When to report.** The filter is a part of the instrumented program. The filter sends the values of the monitored variables to the monitor. When should it do so? The filter sends these values whenever some conditions are satisfied and the conditions depend on the definition of events.

Event Recognizer. The event recognizer detects an event from values of monitored variables received from the filter and as defined in the monitoring script (see Figure~\ref{fig:mac}). Each time it recognizes an event according to the event definitions in the monitoring script, the event recognizer sends it to the run-time checker. In addition to sending events, the event recognizer may also forward variables' values to the run-time checker, which uses them to check certain types of requirements, e.g., correctness of a function computation.

While it is conceivable to merge the event recognizer with the filter, separating the two modules is more advantageous: it shields the system execution from the overhead of abstracting out events from low-level information. In other words, it minimizes interference with the monitored system's execution. On the other hand, this architecture contains communication overhead, as the filter must send monitored variable changes to the event recognizer.

Another related issue is the location of the event recognizer. The event recognizer can be on the side of the system or the side of the run-time checker. In former case, the system sends only high level events, which reduce communication overhead between the system and the monitor. The disadvantage of this approach is an overhead in the system due to the event detection. In the latter, the communication overhead might be big because the filter sends values of the monitored variable to the event recognizer. However, we can reduce communication overhead by sending only necessary change to values of monitored variables. Furthermore, it does not cause extra overhead to the system and it is conceptually clean in a sense that we separate the monitor from the system clearly. Therefore, we put the event recognizer on the side of run-time checker. A further potential advantage of this choice is when we extend the MaC paradigm to distributed systems. In this case it will be possible to distribute filters, which communicate with a central event recognizer.

Run-time Checker. The run-time checker essentially checks membership of the execution thus far in some set that contains all ``acceptable behaviors". The nature of this set determines the kind of assurance we may have about the correctness of the system. For example, if the set of acceptable behaviors is the set of traces, then all we can say about an execution that is ``passed" by the run-time checker is that the sequence of events seen is correct.[1] Such a problem is called the trace validity problem, which is a membership-checking problem to determine if a given trace is in the set of valid traces. For sufficiently expressive requirements specification languages (such as a process algebra like ACSR) this problem turns out to be NP-complete. Thus care should be taken in defining this language to make it expressive enough while still leading to a tractable trace validity problem.

What kinds of formal guarantees does the monitor provide? If all the checking that is required is the validity of a trace in MEDL then the monitor absolutely guarantees that the system behavior so far is correct. If the program uses numerical functions, the correctness of whose outputs is checked using the program checking paradigm, then the guarantees on system behavior so far will be probabilistic. However, the probability that the guarantee is incorrect is under our control and we can make it so small that it essentially as good as a deterministic guarantee.

There is a possibility that monitoring of a running system's behavior for sufficiently long will allow us to provide statistical guarantees on the system itself and not just on its behavior. This is a direction that we will explore in the future.

Another issue is *unexpected events*. Unexpected events are events that are not described in the requirements specification and thus not detected by the event recognizer. If an unexpected event happens at run-time, the monitor may not see this event and consequently can make a wrong conclusion on the current execution of the system. For example, let us assume that a requirements specification does not specify an event caused by arithmetic exception such as division by zero. When arithmetic exception happens at run-time, which may lead to the system

[1] Another possibility is to have the set of ``acceptable behaviors" be a set of timed traces. In this case, it will be possible to ensure not only the correctness of the system, but also its timing properties.

crash, the monitor might not detect that the current execution is incorrect because it does not receive an event of arithmetic exception which is not defined in the specification. Currently, we do not handle unexpected events, i.e., we assume that unexpected events do not happen or these events do not influence the correct execution of the system. However, these have to be considered for providing complete guarantees.

CURRENT AND FUTURE WORK

The paper proposes the MaC paradigm as a run-time assurance-monitoring framework. It's bridging a gap between the traditional two approaches, static verification and testing. We are currently developing a prototype system. We use Java as our implementation language and mechanically instrument the target system in bytecode level. Its prohibition of using pointers and strong typing system makes keeping track of access to program variables easier than other conventional languages like C/C++. In addition, as Java gains further popularity among industry and research area, more system will be implemented in Java. Choosing Java as the implementation language, we can apply our methodology to many target systems written in Java. For monitoring scripts, we have designed PEDL (Primitive Event Definition Language) and for requirements specification, we have designed MEDL (Meta Event Definition Language).

Our monitoring approach incorporates formalisms for concurrency and communication and program checker for numerical computation during the execution. To use run-time assurance monitoring for safety critical systems, we need to be able to provide formal guarantee of the MaC paradigm, which we have not fully investigated yet.

We are investigating a number of other issues that deals with extension of the MaC paradigm to distributed monitoring. For example, in distributed system environment, how can we capture the global state by each local monitor that knows only local information. We think instrumentation of a Java virtual machine can help distributed monitoring by getting timing information about threads and communications among the system. We performed some experiment on multiple reader-writer examples and have some theoretical result on distributability of monitor.

Another interesting question is about gap between the computation model in a specification language and the real program. For example, ACSR [BGGL93] uses both synchronous timed events and instaneous events. But in a real situation, every computation takes time and it does not completely fit into model of ACSR. Most of a formal specification language has an ideal model, which might not fit into the real program. The question here is how to compare the specification and the system of different models and how to interpret error due to this kind of difference.

REFERENCES

[BCLS97] H. Ben-Abdallah, D. Clarke, I. Lee, and O. Sokolsky. PARAGON: A Paradigm for the Specification, Verification, and Testing of Real-Time Systems. *In IEEE Aerospace Conference*, Vol. 2, pages 469-488, February 1-8, 1997.

[BCD+90] J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, 1990.

[BCG87] G. Berry, P. Couronné, and G. Gonthier. Synchronous Programming of Reactive Systems: an Introduction to ESTEREL. Technical Report 647, INRIA, 1987.

[BGGL93] P. Brémond-Grégoire, R. Gerber, and I. Lee. ACSR: An Algebra of Communicating Shared Resources with Dense Time and Priorities. In *Proceedings of CONCUR '93: International Conference on Concurrency Theory*, August 1993.

[BK95] M. Blum and S. Kannan. Designing programs that check their work. In *JACM* 42(1), pages 269-291, January 1995.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specification. *ACM Transactions on Programming Languages and Systems*}, 8(2), April 1986.

[CG92] S. E. Chodrow and M. G. Gouda. The Sentry System. In *SRDS11*, October 1992.

[CL97] D. Clarke and I. Lee. Automatic Specification-Based Testing of Real-Time Properties. Submitted to *IEEE Trans. on Software Engineering*, Oct 1997.

[ES94] G. Eisenhauser and K. Schwan. Falcon - toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *HPDC*, 1994.

[Sch98] F. B. Schneider. Enforceable Security Policies. Cornell University Technical Report TR98-1664. January 1998.

[Gor88] M. J. C. Gordon. HOL: a proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthsis*, pages 73-128. Kluwer, 1988.

[JP97] L. Jagadeesan and A. Porter. Specification-based testing of reactive software: tools and experiments. *In 19th International Conference on Software Engineering*, May 1997.

[Kur87] R. P. Kurshan. Reducibility in analysis of coordination. In *LNCS*, volume 103, pages 19-39, 1987.

[LKVK98] I. Lee, S. Kannan, M. Viswanathan, and M. Kim. Event model, language and detection mechanism the MaC architecture. Internal document. September 1998.

[DR96] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96), Software Engineering Notes*, pages 106-117, 1996.

[MC95] B. P. Miller and M. D. Callaghan. The PARADYN parallel performance measurement tools. In *IEEE Computer*, volume 28(11), November 1995.

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

[ML97] A. K. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, 1997.

[OSR93] S. Owre, N. Shankar, and J. M. Rushby. The PVS specification language. In ???, February 1993.

[Sch95] B. A. Schroeder. On-line monitoring: A tutorial. In *IEEE Computer*, pages 72-78, June 1995.

[SM93] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. In *IEEE Computer*, pages 32-41, March 1993.

[SS97] T. Savor and R. E. Seviora. An approach to automatic detection of software failures in real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 136-146, 1997.

[Tur96] D. N. Turner. *The Polymorphic Pi-Calculus:Theory and Implementation*. PhD thesis, University of Edingburgh, 1996.