Runtime Verification of Timing and Probabilistic Properties using WMI and .NET

Jane Jayaputera, Iman Poernomo, Heinz Schmidt
CSSE, Monash University, Australia
{janej,ihp,hws}@csse.monash.edu.au

Reliability and availability are key issues to distributed service-oriented systems. In this paper, we present a methodology for run-time verification of reliability and availability properties for distributed architectures. Our approach generalizes the concept of design-by-contract to contracts involving time and probabilities. We define a language for contracts based on Probabilistic real time Computational Tree Logic (PCTL). We provide a formal semantics for this language based on possibile execution traces of a system. Then we describe a .NET-based system for monitoring contracts, built upon the Windows Management Instrumentation (WMI) framework.

1. Introduction

Availability is a primary concern in the design of modern service-oriented systems. A typical example is that of online banking systems, where 24-7, near 100% availability is essential for consumer requirements. Availability is an example of a nonfunctional property, as it expresses requirements that do not involve the input/output functionality of a system. This paper presents a process for designing, implementing and deploying service-oriented software in the Microsoft .NET framework, based on an extension of Meyer's design-by-contract approach to nonfunctional assertions involving time and probabilities. The idea is to enable availability contracts of a design to be verified in the implementation at run-time, providing a level of guarantee that the system will satisfy client requirements.

Meyer's general definition of a contract is a formal agreement between a system component and its clients, expressing each party's rights and obligations. Through specifying contracts and monitoring contract fulfillment, trust between a component clients is modeled and achieved. Traditionally, design-by-contract is employed at a fine-grained level, involving Boolean contract expressions and is used for expressing required functional properties (pre- and post-conditions) on individual methods of objects.

In this paper, we will be interested in nonfunctional design-by-contract architectures. We adapt the notion of a contract to coarse-grained, entire architectures and clients, and to modelling availability rights and obligations of a system. Availability is constraints often involve probability and timing requirements, and are therefore difficult to express using simple Boolean functions. We therefore express our contracts as formulae written in the language of Probabilistic Computational Tree Logic (PCTL). These contractual formulae can be used to specify properties such as "after a request for a service, there is at least 98 percent probability that the service will be carried out within 2 seconds".

The implementation of our method for contract checking involves the Windows Management Instrumentation (WMI) API for .NET. Using WMI, a system is monitored for method calls (corresponding to time steps in PCTL assertions) and for changes in state (over which atomic statements can be verified). Over repeated executions of a system, the probability of a certain assertion holding can be determined with increasing accuracy. When the probability falls below the average set by the assertion, the contract is taken to be violated.

Our approach for contractual specification and runtime verification builds on top of previous work [7]. The work developed a compositional approach to reliability models where probabilistic finite state machines (PFSMs) are associated to hierarchical component definitions and to connectors. Markov chain semantics permits hierarchical composition of these reliability models. The contracts of this paper are formally defined with respect to that semantics.

To the best of our knowledge, no existing approach has adapted PCTL to design-by-contract, and little similar work exists in probabilistic assertion checking.

This paper is organized as follows:

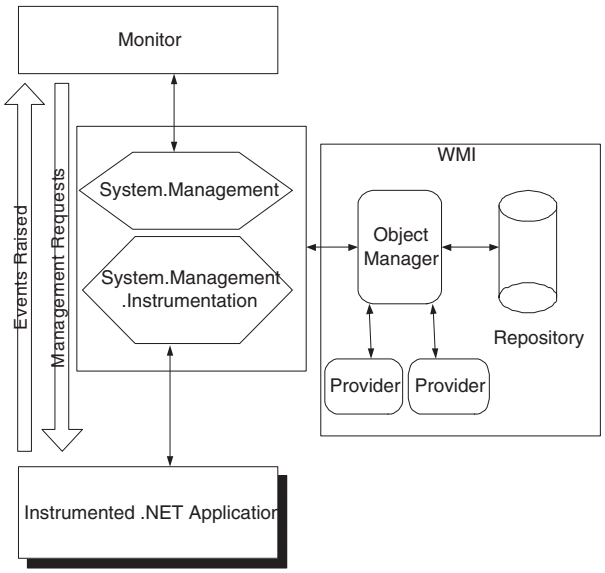
- Section 2. provides an overview of WMI and the .NET framework.

Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04)
1089-6503/04 $ 20.00 IEEE

- Section 3. describes our version of PCTL and provides a formal semantics with respect to possible execution traces of a system.
- Section 4. defines our new notion of design-by-contract, explaining how our PCTL formulae are used as assertions and verified using WMI.
- A simple illustration of our approach is provided in section 6.
- Conclusions and related work are discussed in section 7.

2. .NET and WMI

Microsoft Windows Management Instrumentation (WMI) is the core management-enabling technology built into the Windows 2000/XP/Server 2003 operating systems. WMI is an implementation of the Distributed Management Task Force's (DMTF) Web-Based Enterprise Management (WBEM) initiative, and the DMTF Common Information Model (CIM). The framework enables instrumentation and plumbing through which Windows resources can be accessed, configured, managed, and monitored.



The figure above depicts the three tiers of WMI architecture. A .NET monitoring application is accesses WMI instrumentation functionality using System.Management and System.Management.Instrumentation API classes. This enables subscription to management events and also retrieving collection of management objects from instrumented .NET applications.

WMI technology has 2 components: Management infrastructure and WMI Providers.

- WMI Management infrastructure consists of CIM Object Manager and CIM Repository, used primarily to store schema definitions and provider-binding information. Data would be supplied by the providers on demand, dynamically.
- WMI Providers handle requests from Monitor. The providers generate event notifications related to the underlying applications state changes, and also provide data from the applications to CIM Object Manager.

From the perspective of component-based software deployment, WMI provides a standards-based approach to instrumenting and monitoring interaction of components within an application. The framework provides instrumentation via events and objects.

- WMI events enable a loosely coupled, subscription-based approach to monitoring important changes in an application. Windows and .NET provide a large base set of important events that can be monitored: for example, component activation, method calls and exceptions are available for monitoring as WMI events without the need for manual instrumenting. In addition, the developer can extend the WMI event model to accommodate domain-specific events.
- WMI objects are .NET components that are visible to a monitoring WMI program. They provide a data-centric view of an application. A developer can instrument a program by collecting a range of important data views of the program within a WMI object. The object is instantiated and resides within the the same memory space as program. However, using the WMI API, a monitoring program can make inquiries about the WMI object, permitting instrumentation of the data.

WMI provides a convenient instrumentation framework in which to implement an architectural design-by-contract system. Data and events can be observed by a WMI monitor, and then checked against user-defined constraints. In the next section, we will describe a language for defining probabilistic contracts. Then, in section 4., we shall describe how WMI is used to check these contracts over .NET architectures.

3. PCTL

Probabilistic Computational Tree Logic (PCTL) was devised by Hansson and Jonsson [2] as a specification language for probabilistic model checking. It is used to specify properties hold within a given time and given likelihood. The language takes time to be discrete step units.

For instance, PCTL can specify a property such as: "there is a 99% probability within 20 time steps that a requested service will be carried out". It is decidable for systems whose behaviour can be modeled as probabilistic finite state machines with a truth valuation function associated with boolean

We do not use PCTL for model checking, but instead as a language for making probabilistic assertions about a system. These assertions are to be verified via a monitoring program, with probabilistic constraints compared against average behaviour.

3.1. Syntax PCTL formulae are built from atomic propositions, the usual logical connectives (implication, conjunction and negation) and special operators for expressing time and probabilities.

We use the following grammar (where $atom$ ranges over a set of atomic propositions, int is the integers and $float$ is any floating point number between 0 and 1):

$$\begin{aligned} F &:= atom \mid \text{not } F \mid F \text{ and } F \mid F \text{ or } F \mid \\ &\quad F \text{ until } F \text{ steps: } int \mid \\ &\quad F \text{ leadsto } F \text{ steps: } int \mid \\ Q &:= F \text{ until } F \text{ steps: } int \text{ prob: } float \mid \\ &\quad F \text{ leadsto } F \text{ steps: } int \text{ prob: } float \end{aligned}$$

Formulae of the form F are called nonprobabilistic formulae and Q are called probabilistic formulae.

The informal meaning of formulae as specifications is best understood with respect to the idea of a system that can execute a multiple number of runs. Each run is invoked by a call to the system from a client. Each execution run considered as a series of discrete states. The transition from one state to another is determined by some system activity. The transition from one state to another should be considered as a discrete time step.

The specification expert chooses the system activity that PCTL specifications are to involve, and the consequent demarcation of states and time step. The specification expert should also provide the set $atom$ of atomic propositions, to stand a vocabulary of important system properties that can be verified at each state. In this way, the scope and application of PCTL as a specification language can be fixed to suit a particular domain.

The truth of a formula is determined according to the state the system is in, and, in the case of probabilistic formulae, the number of runs a system has had.

Nonprobabilistic formulae consist of two kinds of formulae:

- Ordinary propositional formulae built from a set of atomic propositions. We assume that atomic propositions correspond to statements about the state of a system, and can be verified to be true or false at any point in an execution. By Boolean semantics, it is then easy to verify if compound propositional formulae are true for a state. Consequently, all propositional formulae can be viewed as assertions about a system's state.

- Formulae with timing. These formulae make statements about the the way a run may evolve, given certain assumptions hold at a state.

  The informal meaning of the until statement is as follows. The statement A until B steps: sp is *false* at a state when B becomes true within s time steps, but where A became false at a step before s. This is equivalent to saying the statement A until B steps: s prob: p is *true* when, with a probability of p, for some $q \leq s$, assuming B is true at q, then A is true at each time step $k < q$.

  The informal meaning of the leadsto statement is as follows. The statement A leadsto B steps: s holds when, with a probability of p, assuming A is true at a state, B will become true within s steps.

Probabilistic formulae are understood in terms of corresponding nonprobabilistic formulae with truth averaged over a number of runs. For instance, A until B steps: s prob: p is true for a number of runs, if the corresponding nonprobabalistic formula A until B steps: s prob: is true over these runs with a probability of p. The case is similar for A leadsto B steps: s prob: p.

We will use PCTL formulae to specify required properties of a system, *immediately after a client call to the system*. That is, we make our specifications with respect to the initial state of system execution runs. However, the ability to make timing constraints enables us to specify requirements over later states in a system.

Example Consider a system in which PCTL time steps are taken to correspond to method calls to components within the system. The important system properties are "the system is in a failed state" and "the system is in a healthy state". We denote these two properties by the propositions *Failed* and *Healthy*, respectively.

Then a fault-tolerance constraint can be specified as

$$\textit{Failed} \text{ leadsto } \textit{Healthy} \text{ steps: } 2 \text{ prob: } .999$$

This states that, with a probability of .999, if the system is in a failed state after a client call, it will take two time steps to become healthy again.

3.2. Formal semantics of contracts We can formally define the semantics of contract verification according to an abstract machine model of a system's execution.

The semantics of a system is given as a series of completed executions. Each execution is modelled as set of states, that determine if an atomic proposition is true or false.

Definition 3..1 (Machines, runs, states) *A* system *consists of a series of runs. A* run *is a finite series of states. A* state *is a truth valuation function of atomic formulae, $atom \rightarrow boolean$. The ith run $exec_i$ of a machine is represented by a function from integers to states: $int \rightarrow atom$. The integer is the number of time steps taken to arrive at a particular state. For instance, $exec_i(0)$ is the initial state of the run, where no steps have been taken yet, $exec_i(10)$ is the state of the run after 10 steps, and so on.*

We can formally define how a PCTL formula is true of a system using this semantics.

Atomic formulae are known to be true or false at any state of a run, by applying the state as a truth valuation function. Compound boolean statements can also be computed for states in the obvious way. Timing formulae without probabilities are determined to be true with respect a run, with initial requirements computed against the initial state of the run and subsequent requirements computed against following states within the timing bounds. Probabilistic formulae are computed by taking average non-probabilistic truth values over the runs for a system.

Definition 3..2 *We have a recursive definition of truth, given with respect to a number of runs r.*

For a nonprobabilistic formula F, define $IsTrue(F, r, t)$ is whether F is true at run r after t time steps.

- *Assume F is atomic. Then $IsTrue(F, r, t)$ holds if $exec_r(t)(F) = True$.*
- *Assume $F \equiv A$ or B. Then $IsTrue(F, r, t)$ holds if $exec_r(t)(A) = True$ or $exec_r(t)(B) = True$.*
- *Assume $F \equiv A$ and B. Then $IsTrue(F, r, t)$ holds if $exec_r(t)(A) = True$ and $exec_r(t)(B) = True$.*
- *Assume $F \equiv$ not A. Then $IsTrue(F, r, t)$ holds if $exec_r(t)(A) = False$.*
- *Assume $F \equiv A$ leadsto B steps: s. Then $IsTrue(F, r, t)$ holds when*

$$IsTrue(A, r, t) \Rightarrow IsTrue(B, r, j)$$
$$\text{for some } j \leq t + s$$

- *Assume $F \equiv A$ until B steps: s.*

$$\text{there is a } j \leq t + s \text{ such that, for all } t \leq i \leq j,$$
$$IsTrue(B, r, j) \Rightarrow IsTrue(A, r, i)$$

We then define truth for any formula F after a system after a client call after r runs, $IsTrue(F, r)$ as follows:

- *If F is nonprobabilistic,*

$$IsTrue(F, r) = IsTrue(F, r, 0)$$

the truth value of F at the initial state of the system (the state immediately after a client has called the system).

- *Assume $F \equiv A$ until B steps: s prob: p. Then $IsTrue(F, r, t)$ holds when*

$$\frac{\sum_{i=1}^{r} IsTrue(A \text{ until } B \text{ steps: } s, r)}{r} \leq p$$

- *Assume $F \equiv A$ leadsto B steps: s prob: p. Then $IsTrue(F, r, t)$ holds when*

$$\frac{\sum_{i=1}^{r} IsTrue(A \text{ leadsto } B \text{ steps: } s, r, t)}{r} \leq p$$

4. Design-by-contract

Based on the formal semantics, we can define a sense in which our formulae can be checked at runtime over the execution of a system.

Design-by-contract traditionally uses Boolean functions to specify the contracts between two parties. The idea is that Boolean assertions specify constraints over an aspect of a system. Runtime monitors check for violations of assertion contracts and notify the system administrator in the event of violation. This leads to further improvements in the system.

We adapt this approach to formulae written in the language of PCTL. Essentially our approach is to use WMI to monitor and build a semantic model of system runs. At the end of each run, PCTL assertion formulae are checked against the total set of runs so far obtained. When the average behaviour falls below required probabilities, the assertion is violated and the administrator is notified.

5. Deployment

Our assertion monitoring framework is generic over several properties.

- In our implementation, time steps are generic over some chosen event that occurs regularly and can be monitored by the WMI. Possible WMI events could be method calls, ping heartbeat protocols, specific method call executions, or user-defined WMI time step events.
- PCTL formulae can be checked against a .NET system according to a semantic mapping that must be defined between atomic propositions and .NET boolean valued functions about system data gathered by WMI components at each state of the execution. This provides the ability for the user to define the semantics of their atomic PCTL assertions.
- The point at which probabilistic assertions begin to be checked can be set by the specification expert. This is defined as an adequately sized number of sample runs against which probabilistic formulae can be checked.

In general, we envisage WMI libraries to be defined over particular domains, with a deployment package to be reusable for many applications in that domain. A given system deployment in our framework involves associations between WMI events and monitored data views and elements of a PCTL assertion. In particular, a monitored system involves

- a library of WMI events and components. At a minimum, this consists of the standard WMI events that can monitor NET applications.
- a library of .NET boolean functions.
- a vocabulary of atomic boolean propositions (to be used in a PCTL grammar for assertions about the system).

These libraries and semantic mappings between them are defined using an administration interface, using reflection to obtain information about domain specific WMI libraries. The user can specify contracts programmatically, as custom attributes of .NET classes, and also externally, using the administration interface.

5.1. Construction of a semantics Verification of formulae can be done according to the formal semantics, given a set of runs for the system. To construct such a set, we monitor property values of WMI component views of a system.

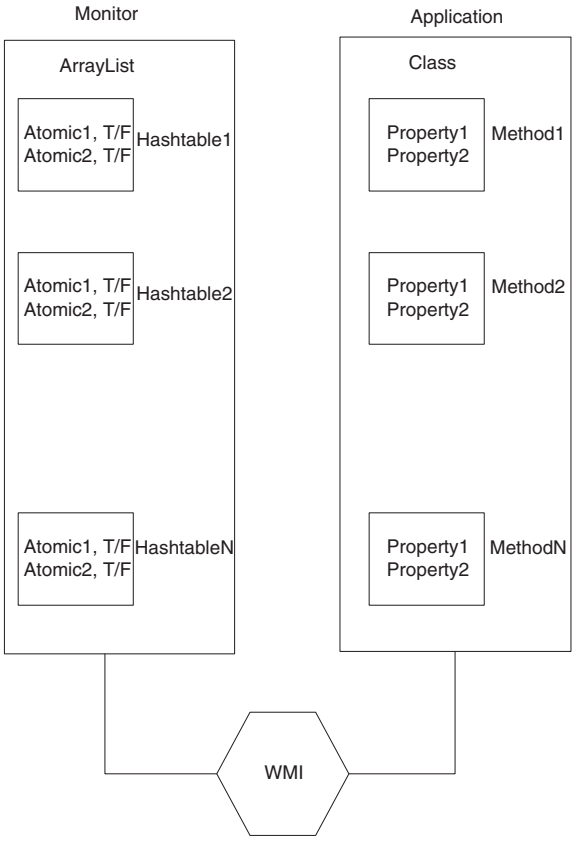
The monitoring process proceeds as follows for a single run:

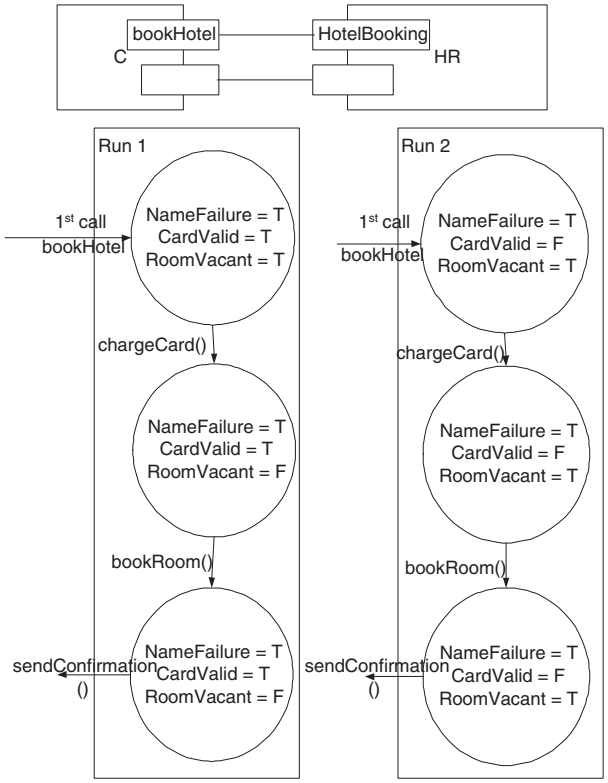
1. We take an empty list, which will represent the run's states.
2. The monitor waits for an event to arrive as this corresponds to a time step.
3. Upon arrival of the event, the monitor should check truth values of each atomic proposition using the semantic map. Mappings between atomic propositions and truth values are stored in a hashtable, which is added as a node in the run's list.
4. Steps 2 and 3 are repeated until program execution finished and the list for the run is completed.

We can check compound nonprobabilistic formulae with respect to a run, in the sense determined by the semantics.

5.2. Assertion checking We begin to check probabilistic formulae after a certain number of runs. This number is configured by the administrator. When this number has been reached, a probabilistic formula is checked against the average truth value of its corresponding nonprobabilistic subformulae for the runs. This is then checked again after each new run has been built up. The monitor notifies the system administrator in the case that an assertion is false.

We give an illustration of our approach in the following figure. It shows how Monitor listens for an event (MethodEvent) fired by the Application every time a method finished its computation. For the sake of simplicity, we assume one method call as a time step in PCTL. Upon receipt of an event, the Monitor adds a Hashtable representing an Application's method to an ArrayList (dynamically grows and shrinks). Then, the Monitor evaluates the properties value published by the Application and adds a true/false value to a corresponding atomic proposition for that particular property. Let say, name is a property in the Application code, and if it is an error, the Monitor will put a *NameFailure* = *true* in the corresponding Hashtable for that method.





The hotel reservation system above works as follows.

1. When the `bookHotel` method is called on `C`, then `C` calls the `HotelBooking` method of `HR`. This will result in `chargeCard()`, as the first internal method of `HotelBooking`, validating credit card information, to obtain truth value of `NameFailure` and `CardValid`. If the name of the card holder is correct, `NameFailure` is `false`. Otherwise, it is `true`. The same holds for the credit card information: if it is valid, `CardValid` is `true`. Otherwise, it is `false`. `RoomVacant` propositions would not changed. Then the event will be fired and resulting in `bookRoom()` is called.

2. When the `bookRoom()` is called, then a vacant room will be booked so we have `RoomVacant` changed to `false`. Otherwise, it is `true`. Then the event will be fired and resulting in `sendConfirmation()` is called. The internal method calls completed and the call is returned back to the caller.

As the Hotel Reservation system designer, we only want to accept a reservation request if there is a room vacant. After validating credit card and card holder information, we want those two properties to hold, so that a room can be booked in.

Provided with the above information on Hotel Reservation example, we would be able to formulate the fol-

6. Example

To illustrate our approach, we present a simplified hotel reservation system. We focus with two use-cases for the system: credit card validations and vacant room bookings. These use cases would be represented by two methods: `chargeCard()` and `bookRoom()`, each of which will fire an event to notify the monitoring application once it has finished the computation. The monitor will query managed object for the corresponding properties that should be verified. The properties being monitored will be atomic propositions that we are interested in. We deal with three atomic propositions: `NameFailure`, `CardValid`, and `RoomVacant`, which are initialized to `true`. The truth value of these three properties would be seen by WMI once it is published. Therefore, it is up to the designer to check the properties value anytime.

We identify the above hotel reservation system as a business logic component, `HR`, used by the client to handle required connections to a database. `HR` would be used by a client web-service component, `C`, to create hotel reservations.

lowing.

RoomVacant leadsto
(CardValid and not NameFailure) steps: 2

On the first run in the previous diagram, the truth value of the above formula is `true`, while in the second run, the formula is `false`.

7. Related Work and Conclusions

Most work involving PCTL for specification of systems involves static verification of state transition designs. For example, in our previous work [3], we used a model checking tool called PRISM [5] to check PCTL specifications against our fault tolerant architectures.

An active field of research in software engineering is the specification and prediction of nonfunctional properties for systems. For example, [7] predicted software reliability using Markov Chain analysis. However, little work has been done before in combining nonfunctional specification with design-by-contract principles.

Runtime verification and monitoring work in [1] focused on specification based monitoring and on predictive analysis of systems, specific to Java. Based on annotations, Monitoring Oriented Programming (MoP) tried to combine together the system specification and the implementation by extending programming languages with formal specification languages, such as Extended Regular Expression (ERE) and Linear Temporal Logic (LTL). The work makes it possible to separate job roles into programmers, domain experts, and logicians. While it is not easy to change the configuration code based on numerous experiences, the work has not incorporated design-by-contract.

An approach in developing software is by means of guards instead of contracts. Meyer [6] gave extensive comparisons between tolerant approach (using guards) with design-by-contract approach. The code became unnecessarily more complex if we use guards instead of pre-/post- conditions for software communications. This happens in case we have to check each of the guards for assertions. However, tolerant approach remains useful for software elements that deal not with other software elements but with data coming from the outside world, such as user input or sensor data, and therefore are suitable for the presence of nonfunctional aspects of system specification.

We adopt a similar approach to [4] in enforcing correct usage of components formally. The usage policy has to be specified, automatically generating the code at runtime enabling it to be statically verified or dynamically enforced. The usage policy consisted of activation and interaction policies of the components, whereas we use time and probabilities for formal specification at runtime.

It is envisaged that approaches similar to ours will become increasingly useful to coarse-grained system design, implementation and maintenance. Large-scale service-oriented software often has cost tarifs associated with failure to meet availability constraints. Accurately specifying and monitoring such constraints is therefore an important issue. By adapting PCTL to design-by-contract, it becomes easier to systematically test system deployments against timing and probability constraints and to improve implementation performance. Also, by virtue of our framework's WMI base, continual observation of system performance becomes an simpler maintenance and administration task in .NET.

References

[1] Feng Chen and Grigore Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proc. of 3rd International Workshop on Runtime Verification (RV'03)*, volume 89(2). Electronic Notes on Theoretical Computer Science, Elsevier Science B.V., 2003.

[2] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[3] J. Jayaputera, I. Poernomo, and H.Schmidt. Timed probabilistic reasoning on uml specialization for fault tolerant component based architectures. In *Proc. of SAVCBS Workshop at European Software Engineering Conference, Helsinki, Finland*, 2003.

[4] W. DePrince Jr. and C. Hofmeister. Enforcing a lips usage policy for corba components. In *Proc. of 29th EUROMICRO Conference, New Waves in System Architecture*, pages 53–60. Belek-Antalya, Turkey, 2003.

[5] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. In *PAPM/PROBMIV'01 Tools Session*, 2001.

[6] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, Englewood Cliffs, 2nd edition, 1997.

[7] R. Reussner, H. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 66(3):241–252, 2003.