RV'03, Electronic Notes in Theoretical Computer Science 89(2), pp 108 - 127. 2003

RV'03, ENTCS 89, issue 2, pp 108 - 127. 2003.

Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation

Feng Chen and Grigore Roşu [1,2]

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana IL, USA

Abstract

With the explosion of software size, checking conformance of implementation to specification becomes an increasingly important but also hard problem. Current practice based on ad-hoc testing does not provide correctness guarantees, while highly confident traditional formal methods like model checking and theorem proving are still too expensive to become common practice. In this paper we present a paradigm for combining formal specification with implementation, called *monitoring-oriented programming* (MoP), providing a light-weighted formal method to check conformance of implementation to specification at *runtime*. System requirements are expressed using formal specifications given as annotations inserted at various user selected places in programs. Efficient monitoring code using the same target language as the implementation is then automatically generated during a pre-compilation stage. The generated code has the same effect as a logical checking of requirements and can be used in any context, in particular to trigger user defined actions, when requirements are violated. Our proposal is language- and logic-independent, and we argue that it smoothly integrates other interesting system development paradigms, such as design by contract and aspect oriented programming. A prototype has been implemented for Java, which currently supports requirements expressed using past time and future time linear temporal logics, as well as extended regular expressions.

1 Introduction

Most engineering disciplines nowadays consider and accept monitoring as a major design principle to increase safety, reliability and dependability of their products. This is not only true in the context of classical engineering branches, such as building or bridge construction, but also in more recent ones. For

[1] This material is based upon work supported by the joint NSF/NASA grant CCR-0234524.
[2] Email: `(fengchen, grosu)@uiuc.edu`

example, almost all electronics today have one or more fuses, which will rapidly put the device in a safe state if ”something wrong happens”, to either protect the humans using the device or to protect other more expensive parts which could be damaged by cascading effects. In hardware engineering it is common practice to monitor a hardware device or part of it using another, typically much simpler device, such as a watchdog, which would reset or restart the main device whenever it seems to misbehave. In aircraft and spacecraft, automatic, rather fancy controllers are typically monitored to ensure that their predicted state stays within a ”stability envelope”, from where the system can be safely and timely controlled using better understood but slower techniques.

In this paper we advocate the idea that monitoring can and should also be a design principle in software engineering. Unlike in many other engineering branches, software engineers tend to not be responsible when their products, the software programs, fail. All of us got used with rebooting our Windows machines when ”something wrong happens”. The situation is actually much worse, because our lives in hospitals or aircraft as well as our privacy and security on the internet depend on huge software systems. Formal program verification [2] techniques, such as theorem proving and model checking, rigorously and systematically check that a software or hardware product satisfies its intended requirements specification. The main problem with these techniques is that they do not always scale up well, so they often are not usable in common software practice. In this paper we propose that specification and implementation should *together* form a system, and that they should have a dual role: specification is checked at runtime against the execution trace generated by the implementation. Moreover, they can and should *interact* to each other *by design*; for example, recovery code can be provided to be executed when a safety specification is violated.

The proposed paradigm is simply called *monitoring-oriented programming*, and is abbreviated *MoP*. The general paradigm is language and specification formalism independent. However, a prototype MoP environment has been implemented and is also presented. Practice has shown that there is *no universal logical framework* to express requirements. Certain requirements can be best expressed using a certain logic formalism, for example temporal logics, while others can be best expressed using others. On the other hand, programming languages are intended to be universal. For these reasons, we believe that any MoP environment should provide the ability to define monitoring logical frameworks on top of a target programming language. Based on experience with monitoring logics, we provide a formal abstraction of the informal notion of ”monitoring logic”. Essentially one can add a new logic to a MoP environment by providing a program which takes as input a formula and returns an output in a pre-defined format (See Section 4). One can regard MoP from at least three viewpoints:

- As a discipline by which one *increases the reliability and dependability of a system by monitoring* its requirements against its implementation.
- As an *extension of programming languages with logics*. One can add logical statements anywhere in the program, that can refer to past or future states of the program. These special statements are like any other programming language boolean expressions, so they give the user a lot of flexibility in how to use them: to terminate the program, guide its execution, add new functionality, throw exceptions, etc.
- As a *light-weighted formal method* that complements the more traditional formal methods such as theorem proving and model checking. The idea here is that MoP avoids verifying an implementation against its specification statically, but rather *does not let it go wrong* at runtime.

Section 2 discusses several factors that we find important in motivating and designing an MoP environment. Section 3 describes our MoP environment at a high level, Section 4 presents three monitoring logics that are currently supported, and then Section 5 presents our current prototype for Java, called Java-MoP. Section 6 relates MoP with other paradigms in system development and analysis, such as design by contract, aspect oriented programming and runtime verification, and then Section 7 concludes the paper.

2 Advantages and Desirable Characteristics of MoP

This section discusses advantages and necessary features of MoP.

2.1 Separation of Roles

An MoP environment facilitates separation of responsibilities, making the project development process more effective. It is natural in an MoP application to consider several phases and assign specific roles to specific individuals. These can include logic experts, domain experts, requirement analyzers, software designers and programmers. Logic experts and domain experts work together to find out the logics which best meet the application domain needs. Domain experts and requirement analyzers work together to formally specify the requirements of applications. Requirement analyzers help designers to design application's architecture, assuring the requirements are correctly obeyed. At this phase, formal specifications can be integrated into the design as additional information. Designers help the programmers understand and implement the design, including appropriate specifications as annotations into the code. Programmers do not necessarily need to understand these specifications; they only need to provide the basic ingredients in order for those specifications to be transformable into real monitors, namely the atomic predicates.

The major advantage here is that individuals do not need to know much beyond their own area of expertise. For instance, it is unnecessary for programmers to know about logics; all they need to know is what the atomic predicates they need to provide represent with respect to the state of the pro-

RV'03, ENTCS 89, issue 2, pp 108 - 127. 2003.

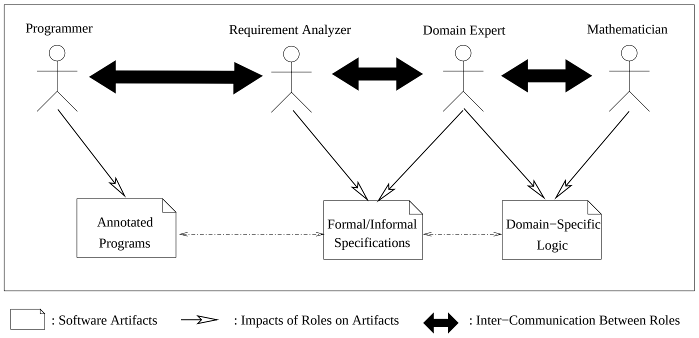


Fig. 1. Separation of Roles.

gram they implement. On the other hand, logic experts and domain experts do not need to know programming, not even what particular programming language is used. Figure 1 shows the needed interaction between individuals involved in an MoP application development process.

2.2 Language and Logic Independence

A major thought in our design is to *not* modify the host programming language, so the users can keep on their favorite development tools and compilers. The mechanism by which one can add formal specifications to the program is through *annotations*, a special kind of comments, which can be processed by MoP tools but ignored by other language tools. An MoP tool synthesizes host language code from these annotations.

Besides, one should keep the annotation language simple and generic. Our specifications are as simple as possible, they consist of only two entities, namely atomic predicates and formulae. The predicates usually have clear intuitive meaning for programmers, and formulae are defined using the predicates plus the syntax of the underlying logic of the specification under consideration. As discussed above, the formulae are extracted from requirements, as defined by domain experts. Programmers only need to understand the meaning of the atomic predicates, implement them in the host language, e.g., Java, and reasonably handle the possible violation.

2.3 Automation, Extensibility and Flexibility

One would like to add new monitoring logics to an MoP environment modularly, like plug-ins. Once a new logic is added, then one would like to simply allow formal specifications in those logics in a ”push-button” fashion. Monitoring logic plug-ins can be stored in WWW data-bases, so users do not have to implement already existing logics. A general architecture, which enables the use of logics as plug-ins, needs therefore to be designed. Within such an architecture, primary modules are clearly separated and communication protocols between modules need to be standardized. Tools will specialize this architecture to support specific programming languages and logics. However, although the tools are usually language-specific, an important observation is that the result of transformations of logic specifications can be abstract and language independent. By separating the logic transformation modules from the code generation, we can reuse the transformation modules (what we call code generators in Subsection 3) for different languages. Besides, the result of the logic engines is hard and unnecessary to be standardized, due to the distinct natures of logics. We thus do not restrict the protocol between the logic engines and the code generators, hereby giving the logic designers the maximum of flexibility.

2.4 Inline Monitoring versus Offline Monitoring

Depending on where the monitoring code is executed, one can distinguish between inline monitoring and offline monitoring. In inline monitoring, the monitoring code replaces the annotation containing the specification from which it was generated. That means that monitoring is being performed using the resources of the monitored program. In particular, an inline monitor cannot detect whether the program gets deadlocked or is stopped unexpectedly. However, it has the advantage that the real program's state is available, so communication overhead is significantly reduced.

On the other hand, in offline monitoring the monitoring code is executed within a different process, potentially on a different machine. The annotation is then replaced by instrumentation code which emits relevant events to the monitoring process. One advantage of offline monitoring is that it allows the centralized computation model, namely one monitor server can be used to monitor multiple programs. Due to advantages and disadvantages of inline versus offline monitoring, we believe that a good MoP environment should allow the users to also specify the desired type of the generated monitor.

2.5 A Light-Weighted Formal Method

Formal methods provide high confidence, but they are not always practical due to state explosion or invariant discovery problems. Testing is widely used in practice but often in an adhoc manner and it is unable to guarantee correctness. In MoP, one merges testing and formal specification in order to achieve the benefits of both approaches, while avoiding some of the pitfalls of adhoc testing and the complexity of full-blown theorem proving and model checking. Of course there is a price to pay in order to obtain a scalable technique: the loss of coverage. The suggested framework can only be used to examine single execution traces, and can therefore not be used to prove a system correct. MoP is based on the belief that software engineers are willing to trade coverage for scalability. With respect to error discovery, MoP can therefore be regarded as a framework implementing efficient algorithms to find *many* errors in programs, but admittedly not all of them.

3 The Architecture of an MoP Environment

Figure 2 shows the general architecture and Figure 3 shows the workflow of the MoP environment that we have implemented so far, which is further instantiated to Java in Section 5. There are three levels of modules, namely

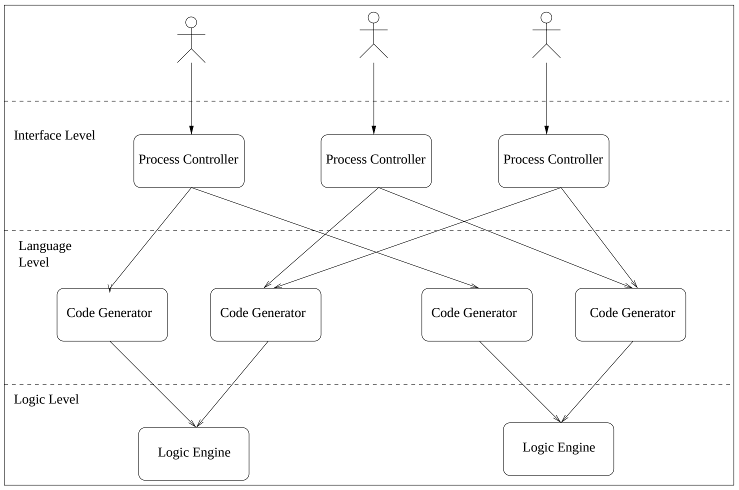


Fig. 2. Architecture of an MoP environment

the interface level, the language level and the logic level. Modules on a lower level can be reused by those on the adjacent upper level. The modules of the interface level are called *process controllers*, those on the language level are called *code generators*, and those on the logic level are called *logic engines*.

The process controllers have three major functions. The first is to extract the formal specifications from the annotations and dispatch them to corresponding code generators. The second is to collect the outputs of code generators and transform them into host language executable code. The third is to provide an interfaces to users, text- or graphic-based.

Code generators play an intermediate role between the process controllers and the logic engines. They take formal specifications from main modules, turn them into specific formats recognizable by the logic engines, and then send them to the corresponding logic engines. In the other direction, they read the results produced by logic engines and translate them into fragments of real programs which are sent to the process controllers to be transformed into real monitoring code. Both the process controllers and the code generators are host language specific.

The logic engines are those which give the power of the MoP approach. Each yields an efficient monitoring algorithm for a specific monitoring logic. They take formulae as inputs. The outputs of the engines consist of abstract pseudocode, which is host-language independent. Thus the logic engines can

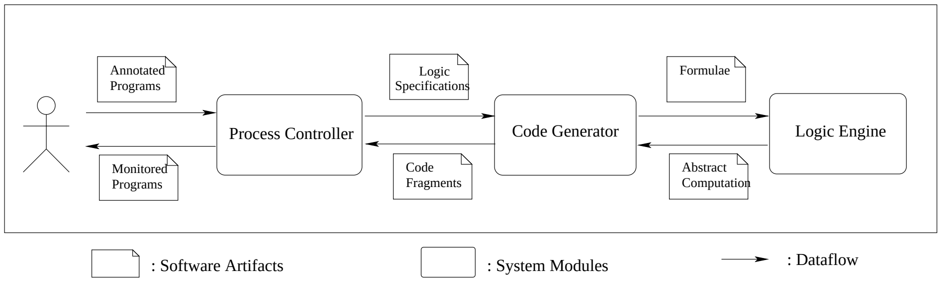


Fig. 3. Workflow in an MoP environment

be reused across different languages. More precisely, a code generator module will be implemented to wrap a logic engine for a specific language, while a logic engine can serve multiple code generators for different languages.

The communication protocols between modules should answer two questions, how they interact with each other and what is the format of the information transferred between them. To achieve the maximum of flexibility, the modules are implemented as individual programs whose inputs and outputs consist of just ASCII text. The formats of modules inputs/outputs are standardized to make the MoP environment easily extensible. The input of code generators is directly extracted from code annotations, containing the language-specific definition of the predicates and the monitoring formulae defined using predicates. The output may include several parts according to the requirements of the underlying logics. Section 4 discusses the output format in more detail. We do not restrict the formats of the logical engines' inputs/outputs, since there is no uniform, effective solution to all logics. For example, DFAs are enough for some logics, such as future time linear temporal logic, but other logics need more complex procedures, e.g., dynamic programming for past time linear temporal logic (see Section 4).

Each code annotation is divided into three parts: the *annotation head* contains a keyword identifying uniquely a logic; the *logic specification* contains the definitions of the predicates and the formula to monitor; the *failure handler* contains the user defined code which will be triggered when the formula is violated. Subsection 5.3 gives a concrete example of using the Past Time Linear Temporal Logic in Java.

4 Logical Frameworks for Monitoring

Different requirements can be best expressed using different underlying logics, so one would ideally like to attach new logical frameworks for expressing requirements in a modular way to an MoP environment. Since formal requirements need to be translated into executable code, logical frameworks should provide a standardized interface in order to facilitate extensibility of MoP. The input of a logic module is clearly a logical formula, but the format of its output is less obvious.

We propose five dimensions that the output of any monitoring logic module should consider in order to be attachable to an MoP environment. These are listed below and will be instantiated in the sequel for the three logics discussed:

Declarations. This part lists as variables those program states which should be maintained for the next step of monitoring. These variables need to be inserted by the MoP environment at appropriate places, which depend upon the target programming language.

Initialization. The initialization phase prepares the variables for starting the monitoring and is executed only once, the first time the monitoring

RV'03, ENTCS 89, issue 2, pp 108 - 127. 2003.

breakpoint is encountered during the execution of the program.

Monitoring body. The monitoring body is the main part of the monitor, which is being executed any time the monitor breakpoint is reached, except the first time. Depending on the type of monitoring desired for the requirement in question, such as inline versus offline, or synchronous versus asynchronous, the monitoring body is executed before the program is allowed to continue, or is executed in parallel via forking, or even as a process on a different machine.

Success condition. This gives the condition stating that the monitoring requirement has been fulfilled, so there is no reason to monitor it anymore. In the context of future time temporal logic, as an example, a successful condition becomes true when a requirement of the form "eventually F" is being monitored and F has been observed to hold. In the context of an execution environment supporting self-modifying code, the monitor can be entirely removed when the success condition becomes true, to eliminate entirely the runtime overhead.

Failure condition. This gives the condition that shows when the trace violates the requirements. When this condition becomes true, user provided recovery code will be executed. "Recovery" should be taken with a grain of salt here, because such code can not only throw an exception or put the system in a safe state, but also attach new functionality to the program.

We have implemented these for past time and future time linear temporal logics (ptLTL) [16,13], as well as for extended regular expressions (ERE), so these three logics are supported by our current MoP prototype. We briefly present our implementations of these three logics next, mentioning that we are currently designing implementing modules for real-time temporal logic (RTL) [17] and for metric temporal logic (MTL) [18] following the same structure.

4.1 Past Time Linear Temporal Logic

Past time linear temporal logic (ptLTL) formulae are routinely used to express safety requirements. In this section we show how the dynamic programming based monitor generator presented in [7] can be organized as a logic engine in our MoP environment.

4.1.1 Syntax and Semantics

We allow the following constructors for ptLTL formulae, where A is a set of "atomic propositions":

$$\begin{array}{llr} F ::= & \mathit{true} \mid \mathit{false} \mid A \mid \neg F \mid F\ \mathit{op}\ F & \text{Propositional operators} \\ & \odot F \mid \diamond\!\!\cdot F \mid \boxdot F \mid F\ \mathcal{S}_s\ F \mid F\ \mathcal{S}_w\ F & \text{Standard past time operators} \\ & \uparrow F \mid \downarrow F \mid [F, F)_s \mid [F, F)_w & \text{Monitoring operators} \end{array}$$

The propositional binary operators, *op*, are the standard ones, that is, disjunction, conjunction, implication, equivalence, and exclusive disjunction. The standard past time and the monitoring operators are called "temporal operators", because they refer to other (past) moments in time. The operator $\odot\ F$ should be read "previously F"; its intuition is that F held at the immediately previous step of execution. $\Diamond F$ should be read "eventually in the past F", with the intuition that there is some past moment in time when F was true. $\boxdot F$ should be read "always in the past F", with the obvious meaning. The operator $F_1\ Ss\ F_2$, which should be read "F_1 strong since F_2", reflects the intuition that F_2 held at some moment in the past and, since then, F_1 held all the time. $F_1\ Sw\ F_2$ is a weak version of "since", read "F_1 weak since F_2", saying that either F_1 was true all the time or otherwise $F_1\ Ss\ F_2$.

The monitoring operators $\uparrow$, $\downarrow$, $[\_,\_)_s$, and $[\_,\_)_w$ were inspired by work in runtime verification in [10]. We found these operators often more intuitive and compact than the usual past time operators in specifying runtime requirements, despite the fact that they have the same expressive power as the standard ones. The operator $\uparrow\ F$ should be read "start F"; it says that the formula F just started to be true, that is, it was false previously but it is true now. Dually, the operator $\downarrow\ F$ which is read "end F", carries the intuition that F ends to be true, that is, it was previously true but it is false now. The operators $[F_1, F_2)_s$ and $[F_1, F_2)_w$ are read "strong/weak interval F_1, F_2" and they carry the intuition that F_1 was true at some point in the past but F_2 has not been seen to be true since then, including that moment. For example, in the phone system application which is presented later in Subsection 5.3, the formula, $\boxdot(\uparrow (dialing) \rightarrow \neg \odot(busyTone \vee connected))$, states that one cannot dial when the phone is busy or connected.

4.1.2 Algorithm

An observation of crucial importance is that the semantics of ptLTL can be defined recursively, in such a way that the satisfaction relation for a formula and a trace can be calculated along the execution trace looking only one step backwards. For example, according to the formal, nonrecursive, semantics, a trace $t = s_1 s_2 ... s_n$ satisfies the formula $[\_,\_)_w$ if and only if either F_2 was false all the time in the past or otherwise F_1 was true at some point and since then F_2 was always false, including that moment. Therefore, in the case of a trace of size 1, i.e., when $n = 1$, it follows immediately that $t \models [F_1, F_2)_w$ if and only if *not* $t \models F_2$. Otherwise, if the trace has more than one event then first of all *not* $t \models F_2$, and then either $t \models F_1$ or else the prefix trace satisfies the interval formula, that is, $t_{n-1} \models [F_1, F_2)_w$. Similar reasoning applies to the other recurrences.

Based on the recursive semantics of ptLTL, efficient monitors can be generated from ptLTL formulae. For example, let $\uparrow p \rightarrow [q, \downarrow (r \vee s))_s$ be a ptLTL-formula that we want to generate code for. The formula states: "whenever p becomes true, then q has been true in the past, and since then we have not yet seen the end of r or s". The code translation depends on an enumeration of its subformulae that satisfies the *enumeration invariant*: any formula has an enumeration number smaller than the numbers of all its subformulae. Let $\varphi_0, \varphi_1, ..., \varphi_8$ be such an enumeration:

$\varphi_0 = \uparrow p \rightarrow [q, \downarrow (r \vee s))_s$,
$\varphi_1 = \uparrow p$,
$\varphi_2 = p$,
$\varphi_3 = [q, \downarrow (r \vee s))_s$,
$\varphi_4 = q$,
$\varphi_5 = \downarrow (r \vee s)$,
$\varphi_6 = r \vee s$,
$\varphi_7 = r$,
$\varphi_8 = s$.

The input to the generated program will be a finite trace $t = s_1 s_2 ... s_n$ of n events. The generated program will maintain a state via a function $update : \mathbf{State} \times Event \rightarrow \mathbf{State}$, which updates the state with a given event. According to the recursive semantics, one can use two boolean arrays, $now[8]$ and $pre[8]$, to record the current and previous states respectively, with the meaning that $now[i]$ is true if and only if the current φ_i is true. The algorithm in [7] implemented as a logical engine generates the following output for the formula above:

Declarations. *boolean* $now[8]$, $pre[8]$

Initialization. The following list of assignments:

```
    $state \leftarrow update(state, s_1)$
    $pre[8] \leftarrow s(state)$; $pre[7] \leftarrow r(state)$; $pre[6] \leftarrow pre[7]$ or $pre[8]$;
    $pre[5] \leftarrow$ false; $pre[4] \leftarrow q(state)$; $pre[3] \leftarrow pre[4]$ and not $pre[5]$
    $pre[2] \leftarrow p(state)$; $pre[1] \leftarrow$ false; $pre[0] \leftarrow$ not $pre[1]$ or $pre[3]$
```

Monitoring Body. The following list of assignments:

```
    $state \leftarrow update(state, s_i)$
    $now[8] \leftarrow s(state)$; $now[7] \leftarrow r(state)$
    $now[6] \leftarrow now[7]$ or $now[8]$; $now[5] \leftarrow$ not $now[6]$ and $pre[6]$
    $now[4] \leftarrow q(state)$; $now[3] \leftarrow (pre[3]$ or $now[4])$ and not $now[5]$
    $now[2] \leftarrow p(state)$; $now[1] \leftarrow now[2]$ and not $pre[2]$
    $now[0] \leftarrow$ not $now[1]$ or $now[3]$; $pre \leftarrow now$
```

Failure Condition. $now[0] = false$

ptLTL formulae are meant to be always satisfied, so there is no success condition for ptLTL. The above program can be further optimized, considering that there are only three formulae need to be remembered to calculate the next states, namely $pre[6]$, $pre[3]$ and $pre[2]$. Then we can get the following optimized monitor body, which is the real output of our logic module:

```
  $state \leftarrow update(state, s_j)$
  $now[3] \leftarrow r(state)$ or $s(state)$
  $now[2] \leftarrow (pre[2]$ or $q(state))$ and $(now[3]$ or not $pre[3])$
  $now[1] \leftarrow p(state)$
  if $(now[1]$ and not $pre[1]$ and not $now[2])$
```

```
        then output(``property violated'')
```

Given a fixed ptLTL formula, the analysis of this algorithm is straightforward. Each time the monitoring body is executed, it takes time $\Theta(m)$, where m is the number of temporal operators in the ptLTL formula. The space required is also very reduced, $2m$ bits.

4.2 Future Time Linear Temporal Logic

Future time linear temporal logic (ftLTL) is sometimes more convenient to express safety properties than ptLTL. We next show how the ftLTL monitors based on binary transition tree finite state machines (BTT-FSM) [5] are generated by an appropriate logic engine in our MoP environment.

4.2.1 Syntax and Semantics

ftLTL has the following constructors:

$$F ::= \mathit{true} \mid \mathit{false} \mid A \mid \neg F \mid F \; op \; F \qquad \text{Propositional operators}$$

$$\Box F \mid \Diamond F \mid F \, \mathcal{U} \, F \mid \circ F \qquad \text{Future time operators}$$

ftLTL provides in addition to the propositional logic operators the temporal operators $\Box$(always), $\Diamond$(eventually), $\mathcal{U}$ (until), and $\circ$(next). An ftLTL standard model is a function $t : Nat^+ \to 2^{\mathcal{P}}$ for some set of atomic propositions $\mathcal{P}$, i.e., an infinite trace over the alphabet $2^{\mathcal{P}}$, which maps each time point (a natural number) into the set of propositions that hold at that point. The operators have the following interpretation on such an infinite trace. Assume formulae X and Y. The formula $\Box X$ holds if X holds in all time points, while $\Diamond X$ holds if X holds in some future time point. The formula $X \, \mathcal{U} \, Y$ (X until Y) holds if Y holds in some future time point, and until then X holds (so we consider strict until). Finally, $\circ X$ holds for a trace if X holds in the suffix trace starting in the next (the second) time point. The propositional operators have their obvious meaning. As an example illustrating the semantics, the formula $\Box(X \to \Diamond Y)$ is true if for any time point ($\Box$) it holds that if X is true then eventually ($\Diamond$) Y is true.

4.2.2 Algorithm

It is shown in [5,6] how an efficient data-structure, called *binary transition tree finite state machine (BTT-FSM)*, can be generated from a future time LTL formula. A BTT-FSM is a finite state machine in which transitions are enabled by executing a series of conditionals organized as *binary transition trees*, whose role is to minimize the amount of computation needed in order to make a transition; BTT-FSMs are specifically tuned for monitoring purposes, to reduce the monitoring overhead. BTT-FSMs generated from ftLTL formulae have initial states and two special states, called *true* and *false*, with special meanings: *true* means that the sequence of states observed so far validates the ftLTL formula, in the sense that any subsequent sequence of states would form a model satisfying the formula; *false* means the opposite, that is, that there is no possible continuation of the observed sequence of states which would form a model satisfying the formula.

RV'03, Electronic Notes in Theoretical Computer Science 89(2), pp 108 - 127. 2003

RV'03, ENTCS 89, issue 2, pp 108 - 127. 2003.

Briefly, optimal BTT-FSMs are generated in two steps. A finite state machine whose transitions are activated by boolean propositions is generated first, and then the transitions out of each state are organized in a BTT. The latter can be done by generating all possible BTTs correctly implementing the transitions from each state, and then selecting the minimal one. The former is rather technical and we do not discuss it here; the interested reader is refered to [5] for more details. An example of a BTT-FSM for a traffic light controller requirements formula $\Box(green \rightarrow \neg red\, \mathcal{U}\ yellow)$, saying that "after green yellow comes" can be seen in Figure 4. The logic module implemented in our

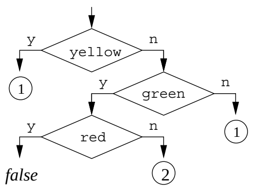
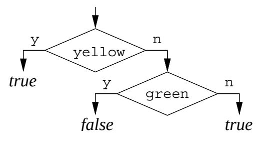
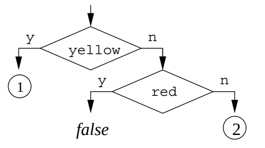
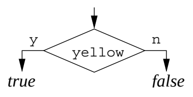
State	BTT for non-terminal events	BTT for terminal events
1	 y yellow n 1 y green n y red n 1 false 2	 y yellow n true y green n false true
2		

Fig. 4. A BTT-FSM for the formula $\Box(green \rightarrow \neg red\, \mathcal{U}\ yellow)$

current MoP environment generates the following output for this formula:

Declarations. *integer state*

Initialization. $state = 1$

Monitoring Body. The following case statement:

$case(state)$

$1\ :\ state = yellow\ ?\ 1\ :\ green\ ?\ (red\ ?\ -1\ :\ 2)\ :\ 1;$

$2\ :\ state = yellow\ ?\ 1\ :\ red\ ?\ -1\ :\ 2;$

Failure Condition. $state = -1$

There is no success condition for this formula, but that may exist for some formulae, such as $\diamond F$. The size of these monitors can be exponential (as a function of the size of the ftLTL formula) but they only need to evaluate *at most* all the atomic state predicates in order to proceed to the next state when a new event is received, so the runtime overhead is actually linear at worst. The size of these monitors can become a problem when storage is a scarce resource, so we pay special attention to generating *optimal* BTT-FSMs. Interestingly, the number of state predicates to be evaluated tends to decrease with the number of states, so the overall monitoring overhead is also reduced.

4.3 Extended Regular Expression

Ordinary software engineers and programmers can understand easily regular patterns, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must *not* occur during an execution. Complementation gives one the power to express patterns on strings non-elementarily more compactly.

4.3.1 Syntax and Semantics

The EREs have the following constructors:

$$R ::= R + R \mid R \cdot R \mid R \cap R \mid R^{\star} \mid \neg R \mid a \mid \epsilon \mid \emptyset.$$

The language defined by a R, denoted by $\mathcal{L}(R)$, is defined inductively as

$$\mathcal{L}(\emptyset) = \emptyset, \mathcal{L}(\epsilon) = \{\epsilon\}, \mathcal{L}(A) = \{A\}, \mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2),$$

$$\mathcal{L}(R_1 \cdot R_2) = \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\},$$

$$\mathcal{L}(R^{\star}) = (\mathcal{L}(R))^{\star}, \mathcal{L}(R_1 \cap R_2) = \mathcal{L}(R_1) \cap \mathcal{L}(R_2), \mathcal{L}(\neg R) = \Sigma^{\star} \setminus \mathcal{L}(R).$$

Given an ERE, as defined above using union, concatenation, Kleene Closure, intersection and complementation, one can translate it into an equivalent expression that does not have any intersection operation, by applying De Morgan's Law: $R_1 \cap R_2 = \neg(\neg R_1 + \neg R_2)$. The translation only results in a linear blowup in size.

4.3.2 Algorithm

A simple, straightforward, and practical approach to monitor EREs is to generate optimal deterministic finite automata (DFA) from EREs [9]. The typical procedure involves the conversion of each negative sub-component of the ERE to a non-deterministic finite automaton (NFA), determinization of the NFA into a DFA, complementation of the DFA, and then its minimization. The algorithm runs in a bottom up fashion starting from the innermost negative ERE sub components. This method, although generates the minimal automata, is too complex and cumbersome in practice. Its space requirements can be non-elementarily larger than the initial regular ERE, because negation involves an NFA-to-DFA translation, which implies an exponential blow-up; since negations can be nested, the size of such NFAs or DFAs could be highly exponential.

Our approach is to generate a minimal BTT-FSM instead (see the previous subsection) from an ERE using coinductive techniques. Briefly, in our approach we use the concept of derivatives of a regular expression which is based on the idea of "event consumption", in the sense that an extended regu-

RV'03, ENTCS 89, issue 2, pp 108 - 127. 2003.

lar expression R and an event a produce another extended regular expression, denoted $R\{a\}$, with the property that for any trace w, $aw \in R$ if and only if $w \in R\{a\}$. Let's consider an operation $\_\{\_\}$ which takes an ERE and an event, then we give several equations which define its operational semantics recursively, on the structure of regular expressions:

$$(R_1 + R_2)\{a\} = R_1\{a\} + R_2\{a\} \tag{1}$$

$$(R_1 \cdot R_2)\{a\} = (R_1\{a\}) \cdot R_2 + \texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi} \tag{2}$$

$$(R^\star)\{a\} = (R\{a\}) \cdot R^\star \tag{3}$$

$$(\neg R)\{a\} = \neg(R\{a\}) \tag{4}$$

$$b\{a\} = \texttt{if } (b == a) \texttt{ then } \epsilon \texttt{ else } \emptyset \texttt{ fi} \tag{5}$$

$$\epsilon\{a\} = \emptyset \tag{6}$$

$$\emptyset\{a\} = \emptyset \tag{7}$$

For a given ERE one generates all possible derivatives that the ERE can generate for all possible sequences of events. This set of derivatives is finite and its size depends on the size of the initial ERE. However a number of these derivative EREs can be equivalent to each other. We check the equivalence of EREs using an automatic procedure based on coinduction, getting a set of equivalence classes of derivatives. These equivalence classes form distinct states in the optimal BTT-FSM. The BTTs are then generated in a similar fashion to those for ftLTL described in the previous section.

Experiments with this logic engine are very encouraging. Our implementation, which is also available graphically on the internet via a CGI server reachable from `http://fsl.cs.uiuc.edu/rv`, rarely took longer than one second to generate a BTT-FSM. For example, the optimal BTT-FSM for the ERE ¬ ((¬ `empty`) (`green red`) (¬ `empty`)) stating a similar traffic light controller safety policy to the one in the previous section using ERE notation is generated as in Figure 5. The output of our ERE logic engine is:

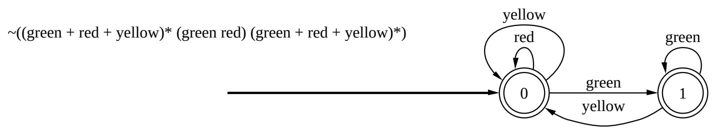

~((green + red + yellow)* (green red) (green + red + yellow)*)
yellow
red
green
0
green
yellow
1

Fig. 5. The optimal BTT-FSM for ¬ ((¬ `empty`) (`green red`) (¬ `empty`))

Declarations. *integer state*

Initialization. $state = 0$

Monitoring Body. The following case statement:

```
case(state)
    0 : state = (yellow ∨ red) ? 0 : green ? 1 : −1;
    1 : state = green ? 1 : yellow ? 0 : −1;
```

Failure Condition. $state = -1$

5 An MoP Prototype for Java

We have implemented an MoP environment prototype, together with the three logic engines presented in the previous section. The prototype works in command-line style, but also provides a graphic user interface (GUI). This prototype will be soon available to download at `http://fsl.cs.uiuc.edu`. Currently, all the logic engines are implemented in Maude. The command line version is implemented in Perl and the GUI version on top of the Eclipse platform using Java.

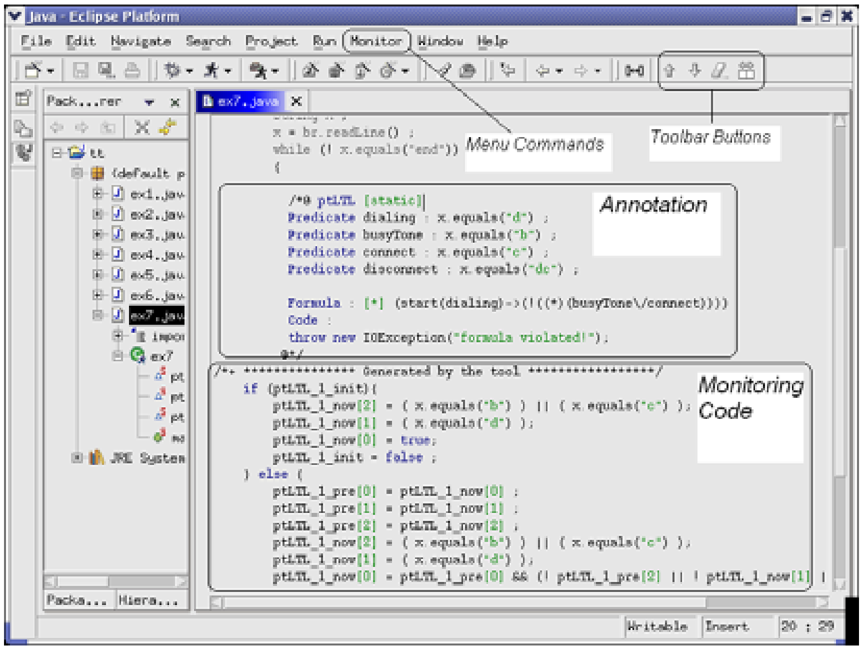
5.1 Overview

The main advantage of the command-line version is that it provides the ability to process the code generation in batch. Basically, the command-line tool is invoked simply by typing in "`generate [-l] [-r] pathname`", where the "`pathname`" is the path to a properly annotated java file or a list file or a directory. The simplest scenario is that the input is a Java file. The prototype processes the file and outputs the result program directly to the standard output. If the input path is a directory then all Java files in that directory are processed. The option *-r* tells the tool to recursively apply the monitor generation procedure to all subdirectories. The option *-l* gives the users a more controllable way to run the batch process, with a text file containing a list of Java files as the input file. The tool will read the list file and process all the listed Java files. When the tool carries out the batch processing, it will backup the source files and then modify those original files directly. The batch file option may be desirable when lots of monitors need to be generated or when they involve large logical formulae which need a long time to be processed.

On the other hand, the GUI tool gives the users a more friendly interface to the MoP environment. Our GUI tool is implemented as plug-ins to the Eclipse platform [15], which contains a powerful and extensible Java IDE. Having Eclipse and our plug-ins installed, the users can choose to open a Java file with our editor, namely the `Annotated Java Editor` (AJE), in the Eclipse workbench when working on a Java project. Figure 6 shows a snapshot of an AJE session. The annotations containing logic specifications, as well as the generated codes, are highlighted, while other parts of the source file are greyed. Users can navigate through annotations, generate monitoring code or remove generated codes by simple clicks or hot keys. Along with other features provided by Eclipse, users can integrate this MoP environment into their development work, interactively editing annotations, instantly checking and debugging the generated code, and so on.

5.2 Implementation

The architecture of our prototype, shown in Figure 7 is a specialization of the general architecture in Figure 2. The process controllers and code generators



```
    x = br.readLine() ;
    while (! x.equals("end"))
    {
      /*@ ptLTL [static]
      Predicate dialing : x.equals("d") ;
      Predicate busyTone : x.equals("b") ;
      Predicate connect : x.equals("c") ;
      Predicate disconnect : x.equals("dc") ;

      Formula : [*] (start(dialing)->(!((*)(busyTone\/connect))))
      Code :
      throw new IOException("formula violated!");
     @*/
/*+ *************** Generated by the tool *****************/
  if (ptLTL_1_init){
     ptLTL_1_now[2] = ( x.equals("b") ) || ( x.equals("c") );
     ptLTL_1_now[1] = ( x.equals("d") );
     ptLTL_1_now[0] = true;
     ptLTL_1_init = false ;
  } else {
     ptLTL_1_pre[0] = ptLTL_1_now[0] ;
     ptLTL_1_pre[1] = ptLTL_1_now[1] ;
     ptLTL_1_pre[2] = ptLTL_1_now[2] ;
     ptLTL_1_now[2] = ( x.equals("b") ) || ( x.equals("c") );
     ptLTL_1_now[1] = ( x.equals("d") );
     ptLTL_1_now[0] = ptLTL_1_pre[0] && (! ptLTL_1_pre[2] || ! ptLTL_1_now[1] |
```

Fig. 6. Snapshot of an AJE session.

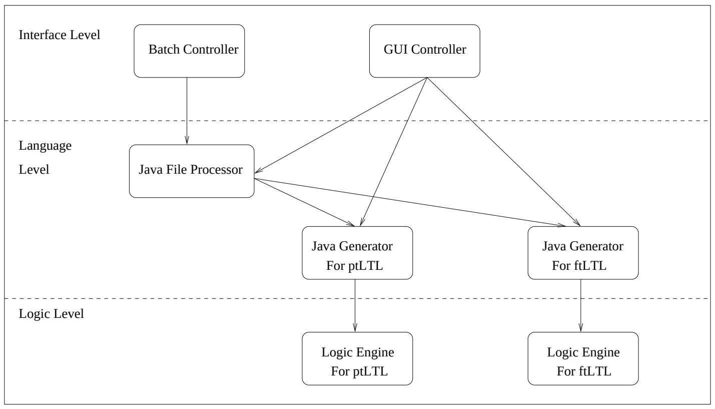


Fig. 7. The Architecture of the current MoP prototype

are specific to Java. Since their functions are mainly text-related, the batch processor, a file processor and the code generators are written in Perl. Since Maude is a very elegant and efficient meta-logic development environment, all the logic engines are implemented by rewriting in Maude. The main module of the GUI tool is implemented in Java.

A configuration mechanism is provided to facilitate extensibility. In this prototype, the Java property file is utilized for configuration, since its format is simple and easy to handle. Each property file is a multiple-line text file, each line containing a (name, value) pair of the form *name* = *value*. Every line in the configuration file associates an annotation keyword to an executable program; e.g., `ftLTL = /bin/ftLTL`. The users can change the code generators or add new ones by directly editing the configuration file with any text editor.

5.3 Example

In this section we further discuss our MoP prototype by means of a simple example. This example is part of a program which can control a phone call process. The annotation here describes a safety property, namely that one cannot dial when the phone is busy or connected.

```
static String x ;
...
x = phone.getStatus() ;
/*@ PTLTL [static]
Predicate dialing : x.equals("d") ;
Predicate busyTone : x.equals("b") ;
Predicate connected : x.equals("c") ;
Formula :
  [*] (start(dialing)->(!((*)(busyTone\/connected))))
Exception Handler:
  throw new Exception("Formula Violated");
@*/
```

Code annotations are marked by */*@ ... @*/*. The past time operators, $\boxdot$ and $\diamond$, are presented as [*] and (*) correspondingly. The MoP-Java handles two kinds of variables in Java classes, namely field variables and static variables. As to the local variables in methods, since they are destroyed once the method invocation is finished, they are not suitable for the monitoring. If the specified logic formula to be monitored is intended to refer to the behavior of a class instead of a particular object, then the generated code should declare static variables; this cannot be inferred from the specification itself, so an option called *static* is available for use in the head of annotations, following the logic-name keyword.

Optimization is always crucial in monitoring, since the monitoring code will inevitably add runtime overhead to the program. Most optimizations are accomplished by efficient algorithms generated by the logic engines and code generators, but some optimization can also be achieved by careful implementation of the modules, such as the elimination of redundant evaluation of atomic predicates. Most of these are handled by BTTs; others are handled by compilers.

A counter is used to track the annotations, and produce appropriate variable declarations to avoid name clashes. Hence, one gets the following monitoring Java code for the annotation in this example, assuming that it is the first monitoring annotation occurring in the program.

```
/*+ *************** Generated by the tool *****************/
static boolean PTLTL_1_pre[] = new boolean[3];
static boolean PTLTL_1_now[] = new boolean[3];
static boolean PTLTL_1_init = true ;
/***************** Generated code ends *************** +*/
...
/*+ *************** Generated by the tool *****************/
if (PTLTL_1_init){
  PTLTL_1_now[2] = ( x.equals("b") ) || ( x.equals("c") );
  PTLTL_1_now[1] = ( x.equals("d") );
  PTLTL_1_now[0] = true;
```

```
    PTLTL_1_init = false ;
  } else {
    PTLTL_1_pre[0] = PTLTL_1_now[0] ;
    PTLTL_1_pre[1] = PTLTL_1_now[1] ;
    PTLTL_1_pre[2] = PTLTL_1_now[2] ;
    PTLTL_1_now[2] = ( x.equals("b") ) || ( x.equals("c") );
    PTLTL_1_now[1] = ( x.equals("d") );
    PTLTL_1_now[0] = PTLTL_1_pre[0] && (! PTLTL_1_pre[2] ||
       ! PTLTL_1_now[1] || PTLTL_1_pre[1]);
  }
  if ( ! PTLTL_1_now[0]){
    throw new Exception("formula violated!");
  }
  /***************** Generated code ends *************** +*/
  ...
```

6 Related Work

In this section we discuss relationships between our approach and other related paradigms, such as design by contract, runtime verification and aspect oriented system development.

6.1 Design by Contract

Design by Contract (DBC) [14] is a technique allowing one to add semantic information to a program by specifying assertions regarding the program's runtime state, and then checking the specification at runtime. It was first introduced by Meyer as a built-in feature of the Eiffel language [19], allowing specification code to be associated with a class in the form of assertions and invariants which can be compiled into runtime checks. There are also some DBC extensions proposed for a number of other languages. Jass [3] and jContractor [1] are two Java-based approaches.

Jass is a precompiler which turns the assertion comments into Java code. Besides the standard DBC features such as pre-/post- conditions and class invariants, it also provides refinement checks. The design of trace assertions in Jass is mainly influenced by CPS [8], and the syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts, consisting of precondition, postcondition, and invariant, with any Java classes or interfaces. Contract methods can be included directly within the Java class or written as a separate contract class. Before loading each class, jContractor detects the presence of contract code patterns in the Java class bytecodes and performs on-the-fly bytecode instrumentation to enable checking of contracts during the program's execution. jContractor also provides a support library for writing expressions using predicate logic quantifiers and operators such as $Forall$, $Exists$, $suchThat$, and $implies$. Using jContractor, the contracts can be directly inserted into the Java bytecodes even without the source code.

We believe that the logics of assertions/invariants used in DBC approaches fall under the uniform format of our logic engines, so that an MoP environment following the principles described in this paper would naturally support DBC

RV'03, ENTCS 89, issue 2, pp 108 - 127. 2003.

as a special methodological case. However, our MoP design also supports offline monitoring which we find crucial in assuaring software reliability, which is not provided by any of the current DBC approaches that we are aware of.

6.2 Runtime Verification

MaC [12] and PathExplorer (PaX) [5] are two logic based monitoring tools, both of which generate monitoring systems from formal specifications. These are general frameworks for logic based monitoring, within which specific tools for Java, Java-MaC and Java PathExplorer, are implemented. MaC uses a special interval temporal logic based language to specify the program behaviors, while JPaX supports just LTL. In order to send the application's states to the monitor, these systems need to instrument the Java bytecodes, which is hard to achieve in some other languages such as C++. Both these runtime verification systems are based on offline monitoring and have hardwired languages for requirements. Besides, the Mac requires less user intervention, namely the user does not need to know where properties are in the code. Our approach supports both inline and offline monitoring, and allows one to add any monitoring formalism to the system.

Temporal Rover [4] is a commercial code generator based on future time temporal logic specifications. It allows programmers to insert formal specifications in programs via annotations and then generates verification code from those specifications, similarly to MoP. Besides generating monitoring code, an Automatic Test Generation (ATG) component is also provided, which can generate test sequences from logic specification. Temporal Rover and its follower, DB Rover, support both inline and offline monitoring. However, these systems also have their requirements logics hardwired.

6.3 Aspect Oriented Software Development

Aspect Oriented Software Development (AOSD) [11] was proposed to support the advanced separation of concerns [20] and has proven that there are numerous advantages to extract and address cross-cutting features separately. In AOSD, cross-cutting features of the applications are developed as stand-alone fragments, and then merged into a whole program with the help of some mechanism, such as compilers or pre-processors. This approach aims at making the structure of the software clearer and at maintaining a better mapping between the requirements specification and the implementation. Current AOSD approaches introduce new language entities to implement aspects and utilize configuration files to combine them together. Usually, the implementation of aspects is still based on the host language. In our opinion, however, there are some important aspects that can be better specified by domain-specific knowledge or logics rather than the host programming language. With the support of automatic code generation, the mapping from specification to implementation is straightforward and can be formally verified.

RV'03, ENTCS 89, issue 2, pp 108 - 127. 2003.

7 Conclusion and Future Work

Monitoring-oriented programming has been introduced in this paper, as a light-weight formal methods paradigm in software development facilitating the combination of specification and implementation. System properties are expressed using formal specifications given as annotations inserted at various user selected places in programs, and then efficient monitoring code is automatically generated. Our proposal is language- and logic- independent, and we argue that it smoothly integrates other interesting system development paradigms, such as design by contract and aspect oriented programming. A prototype has been implemented for Java. Future work includes incorporating more kinds of formal specifications which have been accepted in current software practice, such as DBC, as well as supporting more languages.

References

[1] Parker Abercrombie and Murat Karaorman. jcontractor: Bytecode instrumentation techniques for implementing design by contract in java. In *Electronic Notes in Theoretical Computer Science*, volume 70. Elsevier Science Publishers, 2002.

[2] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[3] Michael Moller Detlef Bartetzko, Clemens Fischer and Heike Wehrheim. Jass - java with assertions. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[4] Doron Drusinsky. Temporal rover. http://www.time-rover.com.

[5] Klaus Havelund and Grigore Roşu. Monitoring java programs with java pathexplorer. In Klaus Havelund and Grigore Roşu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[6] Klaus Havelund and Grigore Roşu. Monitoring programs using rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California, 26-29 November 2001.

[7] Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 342–356. Springer, 2002. Grenoble, France, 8-12 April 2002, Lecture Notes in Computer Science, Volume 2280.

[8] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, New York, 1985.

[9] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[10] Sampath Kannan Insup Lee, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[11] Gregor Kiczales and John Lamping. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[12] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Java-mac: a run-time assurance tool for java programs. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001.

[13] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[14] B. Meyer. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, Upper Saddle River, New Jersey, 2000.

[15] Eclipse Org. Eclipse project. http://www.eclipse.org.

[16] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[17] R. Alur and T.A. Henzinger. Real-Time Logics: Complexity and Expressiveness. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401, Washington, D.C., 1990. IEEE Computer Society Press.

[18] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *RealTime Systems*, 2(4):255–299, 1990.

[19] Eiffel Software. Eiffel language. http://www.eiffel.com/.

[20] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.