Monitoring-Oriented Programming: A Tool-Supported Methodology for Higher Quality Object-Oriented Software

Feng Chen, Marcelo D'Amorim, and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL, 61801, USA
{fengchen, damorim, grosu}@uiuc.edu

March 25, 2004

Abstract

This paper presents a tool-supported methodological paradigm for object-oriented software development, called *monitoring-oriented programming* and abbreviated MOP, in which runtime monitoring is a basic software design principle. The general idea underlying MOP is that software developers insert specifications in their code via annotations. Actual monitoring code is automatically synthesized from these annotations before compilation and integrated at appropriate places in the program, according to user-defined configuration attributes. This way, the specification is checked at runtime against the implementation. Moreover, violations and/or validations of specifications can trigger user-defined code at any points in the program, in particular recovery code, outputting or sending messages, or raising exceptions.

The MOP paradigm does not promote or enforce any specific formalism to specify requirements: it allows the users to plug-in their favorite or domain-specific specification formalisms via *logic plug-in* modules. There are two major technical challenges that MOP supporting tools unavoidably face: *monitor synthesis* and *monitor integration*. The former is heavily dependent on the specification formalism and comes as part of the corresponding logic plug-in, while the latter is uniform for all specification formalisms and depends only on the target programming language. An experimental prototype tool, called JavaMOP, is also discussed, which currently supports most but not all of the desired MOP features. MOP aims at reducing the gap between formal specification and implementation, by integrating the two and allowing them *together* to form a system.

1 Introduction

This paper introduces *monitoring-oriented programming* (abbreviated MOP), a formal tool-supported object-oriented software development methodology in which runtime monitoring is a basic design principle. This work is based on the belief that specification and implementation should *together* form a system, and that they should have a dual role: the specification is checked at runtime against the implementation. Moreover, specification and implementation can and should *interact* with each other by design, rather than grafting monitoring requirements as an add-on to an existing system to increase its safety. Executing recovery code when the specification is violated is very important, but in MOP it is just a technically simplistic use of monitoring. In MOP, runtime violations and validations of specifications result in adding functionality to a system by executing any user-defined code at any user-defined places in the program; executing recovery code, outputting or sending messages, throwing exceptions, etc., are just special cases.

In MOP, specification requirements are given as logical formulae, and then corresponding monitoring code is automatically synthesized from them. Practice has shown that there is no "silver bullet" logic to formally express any requirements. Some can be best expressed using a certain logical formalism, for example temporal logics, while others can be best expressed using other logics, like that of JML, or domain-specific logics. On the other hand, programming languages are intended to be universal. For these reasons, MOP provides the capability of adding logics on top of a target programming language. More precisely, MOP proposes the general concept of a *logic plug-in* as a formalization of the informal notion of monitoring logic. The interface of logic plug-ins is rigorously defined and standardized, and a WWW repository of logic plug-ins is available, where MOP users can download and upload their favorite, or application domain-specific, logics for specifying requirements.

In this paper we present MOP as a general development paradigm to increase the quality of object-oriented software, having a series of desirable characteristics, not all of them supported by our current prototype tool JAVA-MOP. Our prototyping efforts should be regarded as *one possbile approach* towards supporting MOP, which by no means excludes other similar efforts. In short, one can understand MOP from at least three perspectives:

(1) As a discipline allowing one to *improve safety and reliability of a system by monitoring* its requirements against its implementation at runtime. Theoretically, programmers could potentially write monitoring code and integrate it manually in their application. However, by generating and integrating the monitoring code automatically, MOP provides several important advantages, including reduced risk of writing wrong monitors due to programmer's misunderstanding of specifications, complete integration of monitors at places that may change as the application evolves, software products more amenable to formal verification and validation, and increased separation of concerns.

(2) As an *extension of programming languages with logics*. One can add logical statements anywhere in the program, referring to past or future states of the program. These statements can be seen at least like any other programming language boolean expressions, so they give the user a maximum of flexibility on how to use them: to terminate the program, guide its execution, recover from a bad state, add new functionality, throw exceptions, etc.

(3) As a *lightweight formal method*. While firmly based on logical formalisms and mathematical techniques, MOP aims at avoiding verifying an implementation against its specification before operation, by *not letting it go wrong* at runtime. Therefore, if used properly, MOP is expected to scale well.

Section 2 discusses important other software development methodologies and techniques that are closely related to MOP. Section 3 presents the MOP approach as a general methodological paradigm with a series of desired and/or ideal features; many examples are given using the notation of our current prototype. Section 4 describes in detail our prototype tool supporting MOP, called JAVA-MOP. Finally, Section 5 concludes the paper and proposes future work. A technical appendix briefly discusses the monitor synthesis algorithms for temporal logics and extended regular expressions, which are currently part of the WWW repository of logic plug-ins and supported by JAVA-MOP.

2 Related Work

There are several software development approaches related to MOP. What makes MOP different is essentially its generality and modularity with respect to the logics underlying specification requirements, which

allow it to include other approaches as special cases. Part of our efforts described in this paper is to capture these relationships explicitly, by providing specific logic plug-ins.

Design by Contract (DBC) [30] is a software design methodology particularly well supported in Eiffel [43]. DBC allows specifications to be associated with programs as assertions and invariants, which are compiled into runtime checks. There are DBC extensions proposed for several languages. JASS [9] and JCONTRACTOR [1] are JAVA-based DBC approaches. JASS pre-compiles assertions into JAVA. JCONTRACTOR is implemented as a JAVA library allowing one to associate contracts (preconditions, postconditions, invariants) with JAVA classes or interfaces. JAVA 1.4 introduces simple assertions as part of the language, which are then compiled into runtime checks at the place where they were introduced, so they are similar to JASS' `check` primitive. If they evaluate to false then a specific kind of exception is raised. JAVA 1.4 does not support other DBC primitives, such as pre- and post-conditions, or loop variants and invariants. DBC approaches fall under the uniform format of logic plug-ins, so MOP can naturally support DBC variants as special methodological cases. To strengthen this claim, we have implemented a logic plug-in for JASS, so our current MOP prototype supports JASS annotations. However, MOP also allows monitors to be synthesized *out-line*, i.e., not as integral part of the program but as separate processes communicating with the program via proper and automatic instrumentation, which is crucial in assuring high reliability of software (e.g., no deadlocks), not provided by any DBC approach.

Java Modeling Language (JML) [25] is a behavioral specification language for JAVA. It can be used to specify designs for JAVA classes and interfaces, and provides the basis for further analysis, e.g., runtime debugging [6] and static analysis [12]. Basically, JML tries to combine the strengths of both Eiffel [30] and Larch [14]. JML assertions are presented as special annotations in JAVA source files, using a fixed logical formalism. Thus JML specifications can be easily incorporated into an MOP framework for JAVA, and it is in our plan to do that soon for JAVA-MOP via an appropriate logic plug-in. There are two kinds of behavior specifications in JML from which monitoring code can be generated, namely class invariants and pre-/post-conditions of methods. However, the merging points for monitoring code are implicitly determined for JML annotations: pre-/post-conditions are checked before and after method invocations, respectively, while the class invariants are checked each time client-visible methods are invoked. In addition to those rather standard merging points, an ideal feature of MOP in the context of safety critical applications, is to allow monitors that check properties at each object's state change. Our JAVA-MOP prototype does not support this ideal feature in full generality yet, but we intend to approach it via byte-code instrumentation soon.

Runtime Verification (RV) is an area [17, 18, 20, 44] dedicated to provide more rigor in testing. In RV, monitors are automatically synthesized from formal specifications. These monitors can then be deployed *off-line* for debugging, i.e., they analyze the execution trace "post-mortem" by potentially random access to states, or *on-line* for dynamically checking that safety properties are not being violated during system execution. JAVA-MAC [24, 26], JPAX [16, 15] and JMPAX [40, 41] are such RV systems. JAVA-MAC uses a special interval temporal logic as specification language, while JPAX and JMPAX currently support only linear temporal logic. These systems instrument the JAVA bytecode to emit events to an external, out-line monitor observer and have hardwired languages for requirements. JPAX was used to analyze NASA's K9 Mars Rover code [2, 3]. JMPAX extends JPAX with predictive capabilities.

TEMPORAL ROVER [11, 10] is a commercial runtime verification system based on metric temporal logic (MTL) [35] specifications. Like MOP, it allows programmers to insert formal specifications in programs via annotations and then generates verification code from those specifications. TEMPORAL ROVER and its follower, DB ROVER, support both in-line (i.e., the code of the monitor becomes integral part of the code of the program) and out-line monitoring. However, these systems also have their requirement logics hardwired, and their generated monitors are synchronous. The MOP paradigm supports both in-line and out-line, both on-line and off-line, as well as both synchronous and asynchronous monitoring; these modes will be explained in Section 3. Although our current JAVA-MOP prototype does not support all these modes yet, as argued in Section 5, it is not hard to incorporate them. MOP also allows any logical specification formalism to be modularly added to the system. It is expected that all the RV systems that we are aware of will fall under the general MOP architecture, provided that appropriate logic plug-ins are defined. In fact, the development of MOP was partly motivated by our experience in the runtime verification field and especially by our strong interest in unifying the various approaches (that naturally occur in any new area).

Aspect Oriented Programming (AOP) [23] is a software development technique aiming at separation of concerns [45]. An *aspect* is a module that characterizes the behavior of cross-cutting concerns. Aspects are comprised of three basic blocks: join point, point cut, and advice. The first identifies relevant points in the control flow of a program, a point cut represents several join points concisely in a single abstraction, and an advice relates a point cut to an expression that is evaluated when control flow hits a join point. AOP provides a means to define behavior that cross-cuts different abstractions of a program, avoiding scattering code that is related to a single concept at different points of the program, thus aiming for maintenance and extensibility of the code. One can understand AOP as a language transformation technique that mechanically and transparently instruments the code with advice expressions. The support provided for transformation varies with the AOP language, and is called *mechanism*. The transformation syntax of AspectJ [31], for example, provides wildcards to denote identifiers (e.g., method names) in the hierarchy, and one can specify whether the advice is to be added in the reception, prior to or after a method call that matches a given pattern.

Similarly to AOP, MOP requires instrumentation to integrate monitors into the code; however, instrumentation is a small piece of MOP, namely a subpart of its *monitor integration* capability. MOP's most challenging part is indeed to *synthesize monitors* from user defined logical formulae. In AOP, the behavior of each advice is left to be defined by the programmer. For the sake of an analogy, in the case of in-line monitoring one can understand MOP as a synthesizer of AOP advises. However, MOP and AOP are intended to solve different problems. MOP is tuned and optimized to *merge* specification and implementation via monitoring, while AOP aims at separation of concerns. Our current MOP prototype uses AspectJ as an instrumentation infrastructure. However, AspectJ does not provide support for several important MOP features, such as asynchronous and/or out-line monitoring, as well as strong runtime support needed to check a property at each change of the state of an object (not only at joint points such as the start or the end of a method call), which is a must in the context of highly reliable and safety critical software.

Simplex [42] is a software architecture developed to support safe and reliable online upgrade of critical components, in spite of possible errors in the new components. Under Simplex, both the old component, assumed to be simpler but verified and to guarantee safe behavior, and the new one, typically efficient but hard to verify, run simultaneously. The global state is monitored and, if an undesirable state is observed, the new implementation is disabled and the old one takes over. Thus the system keeps running safely until the upgraded component is fixed. Essentially, Simplex is based on the basic idea of runtime monitoring. We are currently exploring, jointly with the main designer of Simplex and his students, the possibility of incorporating Simplex into MOP as a special important logic plug-in.

3 Overview of MOP

The general idea underlying MOP is that software developers insert specifications at critical places in their code via *annotations*. Actual monitoring code is automatically synthesized from these annotations before compilation and placed at appropriate places in the program, according to user-defined configuration attributes of the monitor. We have implemented a prototype, called Java-MOP [5], which is currently limited to the Java language and to four kinds of specification languages, namely, future time temporal logic, past time temporal logic [33, 27, 28], extended regular expressions and JASS.

```
... (Java code A) ...

/*@ FTLTL
   Predicate red : tlc.state.getColor() == 1;
   Predicate green : tlc.state.getColor() == 2;
   Predicate yellow : tlc.state.getColor() == 3;
   // yellow after green
   Formula : [](green -> (! red U yellow));
   Violation handler : ... (Java "recovery" code) ...
@*/

... (Java code B) ...
```

Figure 1: Java-MOP specification.

Before we present the general principles of MOP, we show via a simple example how an MOP framework is expected to work. Figure 1 shows a JAVA-MOP annotation, where a traffic light controller is monitored against the safety property "yellow after green" expressed as a future time linear temporal logic (FTLTL) formula. JAVA-MOP takes such an annotated code and generates the corresponding monitor as plain JAVA code, which is shown in Figure 2.

```
... (Java code A) ...

switch(FTLTL_1_state) {
case 1:
   FTLTL_1_state = ( tlc.state.getColor() == 3) ?
       1 : ( tlc.state.getColor() == 2) ?
          ( tlc.state.getColor() == 1) ? -2 : 2 : 1 ;
   break ;
case 2:
   FTLTL_1_state = ( tlc.state.getColor() == 3) ?
       1 : ( tlc.state.getColor() == 1) ? -2 : 2 ;
   break ;
}
if (FTLTL_1_state == -2) { ...(Violation Handler)... }
// Validation Handler is empty
... (Java code B) ...
```

Figure 2: Generated monitor for Figure 1

More details on the syntax of annotations are given in Section 3.1; Section 4.3.2 presents temporal logic and its corresponding monitor synthesis algorithm.
MOP can be technically regarded as an extension of programming languages with special statements which can refer to current, past or future states of the program. To effectively handle these special statements, a standardized annotation language is needed. Annotations are automatically converted into monitoring code, which can have several configurable attributes. The monitors may need meta-information about the program's state, such as the name of the current method or class, so a standardized communication interface between the program and annotations is also needed. Logics should be easy to incorporate as modules, which we call logic plug-ins, so another standardized interface is necessary for these. We next present the MOP approach in more depth.

3.1 Extending Programming Languages with Logical Annotations

Most works on extending languages with annotations mainly focus on detecting program errors. MOP is built on the belief that logical annotations can play a deeper, active role in the execution of a program. For example, Figure 3 shows how one can use MOP to guarantee authentication before access to protected resources, by simply adding an appropriate safety policy together with an action. The safety policy, expressed in negative form using extended regular expressions (ERE), states that "whenever one accesses a resource, it must be the case that at some moment in the past the end of the authentication process has been seen". This policy is violated if and only if the ERE formula `[^ end(a)]* start(m)`, stating that the end of the authentication has not occurred before the resource started to be managed, is validated. If that is the case then, instead of reporting an error or halting the system, one simply enforces the authentication. Note that this specification has the attribute `class`, which will be explained later in more detail, so the generated monitor will check the safety policy at each state change of the current object.

Therefore, it is often the case that it is easier to express a safety policy by what should *not* happen. This is one of the reasons for which MOP accepts both a violation and a validation handler; another reason comes from the fact that there are situations where one may want to take different actions depending on whether a policy is violated or validated. Violation and validation are not complementary properties.

Hence, by combining specifications expressed using appropriate underlying logical formalisms and code, and by synthesizing and integrating automatically corresponding monitoring code, critical properties of a program can be expressed and implemented rigorously and compactly. Moreover, the obtained program is more amenable to formal verification and validation, because significant parts of it are directly extracted from its specification, thus narrowing the gap between implementation and specification.

```
/*@ ERE {class}
   Event a : authenticate() ;
   Event m : manageResource() ;
   Formula :
      // If the resource is accessed without authentication
      (^ end(a))* start(m)
   Validation Handler: // enforce the authentication
      authenticate();
@*/
```

Figure 3: A class monitor to assure authentication.

3.1.1 MOP Annotations

The MOP annotations are introduced as comments in the host languages, e.g., `/*@...@*/` in JAVA. The structure of a typical annotation is shown in Figure 4.

```
/*@ [[Annotation Name]] <Logic Name> [{attributes}]
   ... Specification Body ...
   [Violation Handler:
      ... code handling the violation ...]
   [Validation Hander:
      ... code to trigger when validated ...
@*/
```

Figure 4: The structure of MOP annotations.

The annotation begins with an optional name, followed by a required name of its underlying logic, e.g., FTLTL or ERE in the examples in Figures 1 or 3, respectively, followed by optional monitor configuration attributes, which will be discussed later, followed by the main body of the annotation, whose format is logic-specific. In the case of FTLTL, for example, the body of the specification has two parts: one containing declarations of predicates based on program states, and the other containing the FTLTL formula built on these.

The last part of the annotation contains user-defined actions, including a *violation handler* and a *validation handler*. Catching the violation is a programmer's main concern in most cases, and then common actions are to throw exceptions or produce error logs; MOP gives one full freedom in how to continue the execution of the program. One may also want to take an action when the specification is fulfilled at runtime. For example, a requirement "eventually F", which is validated when F first holds, may trigger a certain action; an interesting such action in an execution environment supporting self-modifying code is to remove the monitor and thus to eliminate entirely the runtime overhead. All these suggest a strong sense of programming with logics in MOP.

```
/*@ PTLTL {class ! }
   Event backup : backup() ;
   Event logout : logout() ;
   Predicate active : ActiveUserNum > 0 ;
   Formula :
      // Backup starts only after all users are logged out
      [*] (start(backup)->[logout, active)s)
   Violation Handler:
      throw new RuntimeException("Backup start failed!");
@*/
```

Figure 5: A unique class monitor for safe backup.

Figure 5 shows a concrete example of an MOP specification partially describing the backup behavior in some multi-user system, using past time linear temporal logic (PTLTL) [33, 27] extended with intervals [21]. The class and uniqueness (`!`) attributes will be described in the next subsection and PTLTL in Section 4.3.2, where the generated monitor will also be presented.

3.1.2 Monitor Configuration Attributes

The monitors generated from specifications can be used in different ways and can be integrated at different places in the program, depending on specific needs of the application under consideration. For example, some specifications need to be checked at only one point (e.g., pre- or post-conditions), while others must be monitored at any point in the program (e.g., Figure 1 and other class invariants). In some applications, one may want the generated monitors to use the same resources as the rest of the program, in others one may want to run the monitors as different processes. To give the software developers maximum flexibility, MOP allows one to configure the monitors using attributes in annotations. We have found the following attributes to be particularly useful.

The ***uniqueness*** attribute, denoted `!`. The default monitor configuration behavior is to create a monitor instance for each object of the corresponding class. However, if a monitor is specified as *unique*, it will only have one instance at runtime, regardless of the number of objects of that class.

There are two running mode attributes, stating how the generated monitoring code is executed. One is ***in-line*** versus ***out-line*** monitoring. Under *in-line* monitoring, which is the default, the monitor runs using the same resource space as the program. The monitor is inserted as one or more pieces of code into the monitored program. In the *out-line* mode, the monitoring code is executed as a different process, potentially on a different machine or CPU. The in-line monitor can often be more efficient because it does not need inter-process communication and can take advantage of compiler optimizations. However, an in-line monitor cannot detect whether the program deadlocks or stops unexpectedly. Out-line monitoring has the advantage that it allows, but does not enforce, a centralized computation model, that is, one monitor server can be used to monitor multiple programs. In order to reduce the runtime overhead of out-line monitoring in certain applications, one can define communication strategies; for example, the monitored program can send out the concrete values of the relevant variables and the out-line monitor evaluates the state predicates, or, alternatively, the monitored program sends directly the boolean values of the state predicates to the out-line monitor.

The other running mode attribute is ***on-line*** versus ***off-line***. Under *on-line* monitoring, which is the default, specifications are checked against the execution of the program dynamically and run-time actions are taken as the specifications are violated or validated. The *off-line* mode is mostly used for debugging purposes: the program is instrumented to log an appropriate execution trace in a user-specified file and a program is generated which can analyze the execution trace. The advantage of off-line monitoring is that the monitor has random access to the execution trace. Indeed, there are common logics for which on-line monitoring is exponential while off-line monitoring is linear (the execution trace needs to be traversed backwards) [37, 36].

Note that the two running mode attributes are orthogonal to each other. In particular, an in-line off-line configuration inserts instrumentation code into the original program for generating and logging the relevant states as the program executes. Alternatively, an out-line off-line configuration generates an observer process which receives events from the running program and generates and then logs the states relevant to the corresponding specification. The latter may be desirable, for example, when specifications involve many atomic predicates based on a relatively small number of program variables; in this case, the runtime overhead may be significantly reduced if the program is instrumented to just send those variables to the observer and let the latter evaluate the predicates.

In the case of on-line monitoring, a further attribute, ***synchronization***, is possible. This attribute states whether the execution of the monitor should block the execution of the monitored program or not. In in-line monitoring, for example, a synchronized monitor is executed within the same thread as the surrounding code, while an asynchronous one may create a new execution thread and thus reduce the runtime overhead on multi-threaded/multi-processor platforms. Synchronicity also plays a role in synthesizing code from logic formulae, because, for some logics, asynchronous monitoring is more efficient than synchronous monitoring. Consider, for example, the formula "next F and next not F" which is obviously not satisfiable: a synchronous monitor must report a violation right away, while an asynchronous one can wait one more event, derive the formula to "F and not F", and then easily detect the violation by boolean simplification. Note that synchronous monitoring requires running a satisfiability test, which for most logics is very expensive (PSPACE-complete or worse).

The ***effective scope*** attribute, which can be *class*, *method*, *block* or *checkpoint*, specifies the points at which the monitoring code is merged with the monitored program. As indicated by name, the class attribute

specifies an invariant of a class, so it should be checked whenever the class state changes. The method one refers to a specific method, which can be further divided into three sub-categories: pre-method (before the method), post-method (after the method), and exceptional-method (when the method raises exceptions). The annotation can also be associated to a block, e.g., as a loop invariant/variant. Similar to the method annotation, it is divided into three subtypes, namely, pre-block (before the block), post-block (after the block), and exceptional-block (when the block throws exceptions). Finally, the checkpoint attribute, which is the default, states that the monitoring code should be integrated at the exact current place in the program. Section 3.2.3 describes in more depth the monitor integration problem.

The ***category*** attribute allows the user to control the set of specifications to monitor, by assigning priorities to annotations. This way, one has the possibility to synthesize, integrate and monitor only a reduced number of properties.

3.1.3 Retrieving Program Information

Since specification and implementation live together in MOP, the specification needs information about the program in order to define and synthesize its runtime behavior. The general principle is to use the elements of the program to construct the elements of the specification, such as the atomic predicates which are defined on the state of the program. Three other pieces of information about the program, besides its state, have turned out to be practically very important in MOP to enhance its expressibility and functionality, namely, *positions*, *results* and *historic references*. Figure 6 shows an example of how a JASS specification in JAVA-MOP can use the three pieces of program information.

```
/*@ JASS {post-method}
   @ ensures x == Old.x ;
   @ ensures Result == a * a ;
   Violation Handler:
      throw new Exception("Function " + @FNCT_NAME_ +
         " violated specification " + @ANNT_Name);
@*/
int square(int a) { ... (Java Code) ... }
```

Figure 6: Specification retrieving program information.

The *position* information is extracted via special variable names, such as `@METHOD_NAME_` for the name of the method that contains the current annotation. Position information is useful to generate effective messages to users. Many other special position variables are available, including ones for the class and file names, line numbers, etc. The use of predefined variables also allows certain specifications to be independent of their position and of changes of names of methods, classes, files, etc. The *result* and *historic references* are useful to state certain properties, such as pre- and post-conditions of methods. The specification in Figure 6 states that after each execution of the `square` method, the value of the class field `x` stays unchanged and the result of the method is indeed the square of its input.

3.2 Extensible Logical Framework

In order to smoothly support requirements specifications expressed using potentially different underlying formalisms, support for attaching new logical frameworks to an MOP environment is needed. The layered MOP architecture in Figure 7 is currently adopted. It is especially tuned to facilitate extending the MOP environment with new logical formalisms added to the system as new components, which we simply call *logic plug-ins*. More specifically, a logic plugin is usually composed of two modules in the architecture, namely a *logic engine* and a *language shell*. By standardizing the protocols between layers, new modules can be easily and independently added, and modules on a lower layer can be reused by those on the adjacent upper layer. The architecture is composed of four layers.

The first layer provides a friendly user interface, via which one can define, process, and debug MOP annotations. The second layer contains *annotation processors*, each specialized on a specific target programming language. It consists essentially of a program scanner, which isolates annotations to be processed by specialized modules and report errors if these do not follow the expected structure. Annotation processors

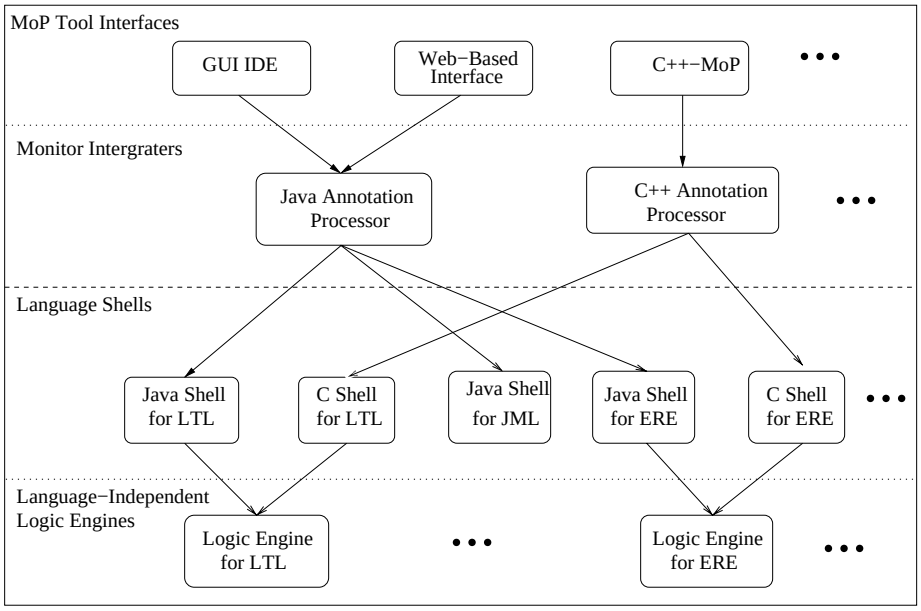

Figure 7: MOP Architecture.

extract the specifications and dispatch them to the corresponding logic plug-ins. Then they collect the generated monitoring code and integrate it within the monitored program according to the configuration attributes. The two upper levels are not expected to change or update frequently.

The two lower levels contain the *logic plug-ins*, which may be added, removed, or updated on a frequent basis. To employ a new specification language, the user needs to develop a new logic plug-in, or obtain one from others or the provided WWW repository (see Section 4.4), and plug it into her MOP environment. Then, annotations using the new specification language can be processed in a "push button" manner. Monitoring code generated for a specific programming language and specification formalism is generated on the third layer, by modules called *language shells*, acting as intermediaries between annotation processors and logic engines. In some simpler cases, such as DBC with just state predicates or JML, the shell can directly generate monitoring code without the need of a logic engine. Modules on the bottom layer are the *logic engines*, forming a core and distinguished feature of MOP. These are generic translators from logic formulae into efficient, typically provably optimal monitors. Their output uses abstract pseudocode, which is further translated into specific target code by corresponding language shells. Therefore, logic engines can be reused across different languages.

The protocols between adjacent layers are based exclusively on ASCII text, to achieve maximum flexibility, extensibility, and to ease debugging and testing. Basically, components in this architecture are implemented as individual programs receiving ASCII text from the standard input and outputting ASCII text to the standard output. Hence, the modules can be implemented by different roles using different languages, and can run on different platforms. Figure 8 shows how an MOP supporting tool transforms an annotated

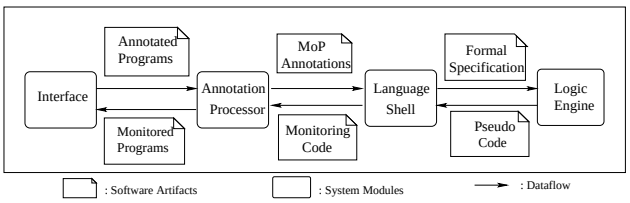

Figure 8: Program transformation flow in MOP.

program.

3.2.1 Interface of Logic Plug-ins

The most important interface is the one between the upper two layers and the lower two layers, which is necessary in order for MOP supporting tools to interact with the logic plug-ins. The input of a logic plug-in is clearly a formal specification, typically a logical formula, but the format of its output is less obvious, since different formal specification languages have different requirements. We next briefly discuss five dimensions that the output of any monitoring logic plug-in must consider in order to be attachable to an MOP supporting tool, regardless of its internal monitor synthesis technique. Several concrete logic plug-ins that have already been experimentally evaluated using our JAVA-MOP prototype are discussed in Section 4.3.

Declarations. Variables needed to store the state of the monitor. These are automatically inserted by the MOP supporting tool at appropriate places in the program, depending upon the target programming language and the specific configuration attributes of the monitor.

Initialization. This phase prepares the variables for starting the monitoring and is executed only once, the first time the monitoring breakpoint is encountered during the execution of the program.

Monitoring body. This is the main part of the monitor, executed whenever the monitor breakpoint is reached.

Success condition. This gives the condition stating that the monitoring requirement has been fulfilled, so that there is no reason to monitor it anymore. For example, for a formula "eventually F" in future time temporal logic, this condition becomes true when F holds the first time.

Failure condition. This gives the condition that states 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 (any) new functionality to the program.

3.2.2 Logic Plug-ins Development

Designing and implementing a monitoring logic plug-in is a highly nontrivial task, requiring a careful analysis of trade-offs among various existing possibilities, or even developing entirely new algorithms. Two or more logic plug-ins can be possible for a given logic, each with different advantages and disadvantages. One typical situation is for a logic to admit a logic plug-in in which all the states of the monitor are explicitly generated, such as an automaton, and one in which only one current state of the monitor is stored, but a means to generate the next state on-the-fly is provided. A major benefit of the modular MOP architecture is that users are encouraged to develop such logic plug-ins *once and for all*, and then to make them available worldwide via WWW repositories. We have devised monitor synthesis algorithms for future time, past time, metric and epistemic temporal logics (Section 4.3.2), as well as for extended regular expressions (Section 4.3.3) and JASS (Section 4.3.1); and we are currently developping logic plug-ins for other logics, including real-time logic (RTL) [34] and EAGLE [4].

A logic plug-in contains two sub-modules, a *logic engine* and a *programming language shell*. The logic engine generates monitoring pseudocode from the input specification, and then the language shell turns that pseudocode into target programming language code. This way, a mathematically skilled developer of the specific logic engine does not need to know the target programming language, and the logic engine can be reused for different programming languages. The language shells are usually straightforward.

3.2.3 Monitor Integration

Once a monitoring code is generated, an MOP supporting tool needs to integrate it into the original program according to the configuration attributes. Specifically, under the in-line mode the monitoring code is placed at proper positions within the monitored program, while under the out-line mode the MOP tool firstly constructs a stand-alone monitor program and then generates corresponding communication code which is placed at proper positions into the monitored program. For the off-line case, code that logs the relevant execution trace is inserted at specific places in the program and a stand-alone analysis program is also

generated, but its role is to detect errors in the log rather than to influence the execution of the program at runtime.

The effective scope attribute says where the monitoring, communication or logging code needs to be placed. The method-related monitor is placed at the beginning of the method or at the end of it, and the monitor associated to the block is handled similarly, but with a smaller scope. The checkpoint monitor is straightforward to integrate: just replace the annotation with the generated code.

Class-scoped monitors, i.e., those generated from annotations with the configuration attribute `class`, are the most difficult to integrate; JAVA-MOP currently supports only a partial integration of these monitors. One approach towards a complete solution would be to insert the monitoring code at all places where relevant events can take place, e.g., before or after method calls, or after variable assignments. However, this simplistic approach can add an unacceptable runtime overhead and may raise some additional technical difficulties because of aliasing. JASS provides a tempting trade-off to reduce the complexity of this problem, by requiring that the specification holds only in those states that can be acknowledged by the client of a library, e.g., before or after the invocation to a method. We are currently adopting this trade-off in our present JAVA-MOP prototype. One should be aware, however, that such a partial solution may allow safety leaks in critical systems.

Monitor integration gives MOP a flavor of AOP, and indeed, our current prototype uses ASPECTJ to realize the instrumentation. However, MOP's major purpose is to extend programming languages with logics as a means to combine formal specification and implementation, rather than to explicitly address cross-cutting concerns. The essential and most challenging parts in the development of a framework supporting MOP are to set up the general logic-extensible infrastructure and to develop the individual logic plug-ins. AOP techniques are only employed to facilitate the final monitoring code integration. Besides, MOP supports local specifications, such as block properties and/or checkpoint, which are not considered in AOP. Moreover, MOP is concerned with the runtime behavior of the program rather than with its static structure. As previously mentioned, our current prototype can not support rigorous class invariants monitoring, because that requires stronger runtime support, beyond the current abilities of AOP.

3.3 MOP as a Software Development Paradigm

MOP aims at smoothly propagating formal specifications from the requirements phase to the implementation, guarding the consistency of specifications at runtime. The formalization of requirements is always a challenge for software developers, partly because of the variety of specification formalisms and because of the lack of general formal languages able to capture requirements across different domains. MOP is a software development methodological paradigm where domain experts can specify requirements formally in the languages they are most familiar with. A push-button infrastructure then takes these formal requirements, generates appropriate monitors and then checks them against the runtime behavior of the program to detect possible errors, to prevent bad behaviors from happening during execution, or to modify the behavior of the program at will as specifications are violated or validated.

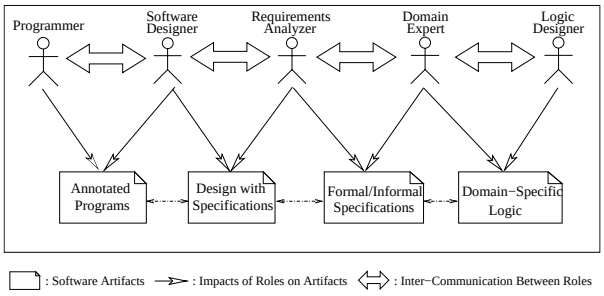

Figure 9: Software development process in MOP

Figure 9 shows a possible software development process using the MOP paradigm. Besides the usual roles in software development, such as domain experts, requirements analyzers, software designers, and programmers, a new role, *logic designers*, is introduced. An important step now is to find out the logic

which best meets the application domain needs. This is done by a cooperation between the logic designer and the domain expert. Once the logic is defined, the logic expert designs the corresponding monitor synthesis algorithm, or searches for an existing one. This algorithm is eventually implemented as a logic plug-in and plugged into the MOP framework. Each logic plug-in is implemented once and for all. It can be reused for other applications, or parts of them, specific to the same domain. Then the domain expert works with the requirements analyzer to specify the concrete system's requirements both formally, using the adopted logic, and informally.

In the following phases, the requirements are turned into system designs and passed on to the design model. During the implementation phase, the programmer embeds the formal specifications into the program using annotations. The only additional work required is to provide the definitions of the atomic predicates and events, which is expected to be straightforward in general. 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 domain-specific logics; all they need to know is the meaning of the logic elements used in the annotation and how to connect them to the state of the program. Similarly, logic experts and domain experts do not need to know programming, not even what programming language will eventually be used.

4 Java-MOP

We next present a prototype tool partly supporting the MOP paradigm, called JAVA-MOP. It consists of an integrated collection of specialized programs, allowing one to easily, naturally and automatically process annotations in JAVA programs, via a user-friendly interface. This prototype has a web page at `http://fsl.cs.uiuc.edu/mop/java-mop`, where it is also available for download.

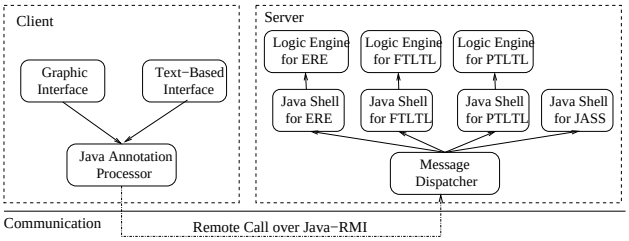

Figure 10: The Architecture of JAVA-MOP

4.1 Overview

JAVA-MOP is essentially a distributed client-server application. It's architecture is shown in Figure 10 and discussed in more detail the next subsection. Users of JAVA-MOP only need to download and install the client, which is implemented entirely in JAVA and is platform independent. One can also download and start the server locally and then reconfigure the client to request monitor synthesis services locally, but this is not necessary unless one wants to experiment with one's own logic plug-ins. Since some of our logic plug-ins are platform-dependent, currently the server can only be installed under Unix.

The client provides both a textual and a graphical user interface. The textual one consists of a simple command taking one or more names of annotated JAVA files as input and generating corresponding files in which monitors are synthesized and integrated appropriately. The textual interface is typically used for batch processing, which is recommended when lots of monitors need to be generated and/or when they involve complex logical formulae which need a long time to be processed. It usually takes a few fractions of a second to synthesize each monitor, but we have encountered ERE formulae which needed close to 2 hours to synthesize corrsponding monitors of hundreds of states. The original annotated files are backed up before transforming them, and therefore all the transformations can be easilly undone.

A friendly GUI interface is also implemented and available for download, based on the ECLIPSE platform [32], containing a powerful and extensible JAVA IDE. The JAVA-MOP tool provides an editor plug-in for ECLIPSE. Having ECLIPSE and the provided plug-in installed, users have the option to open and edit JAVA

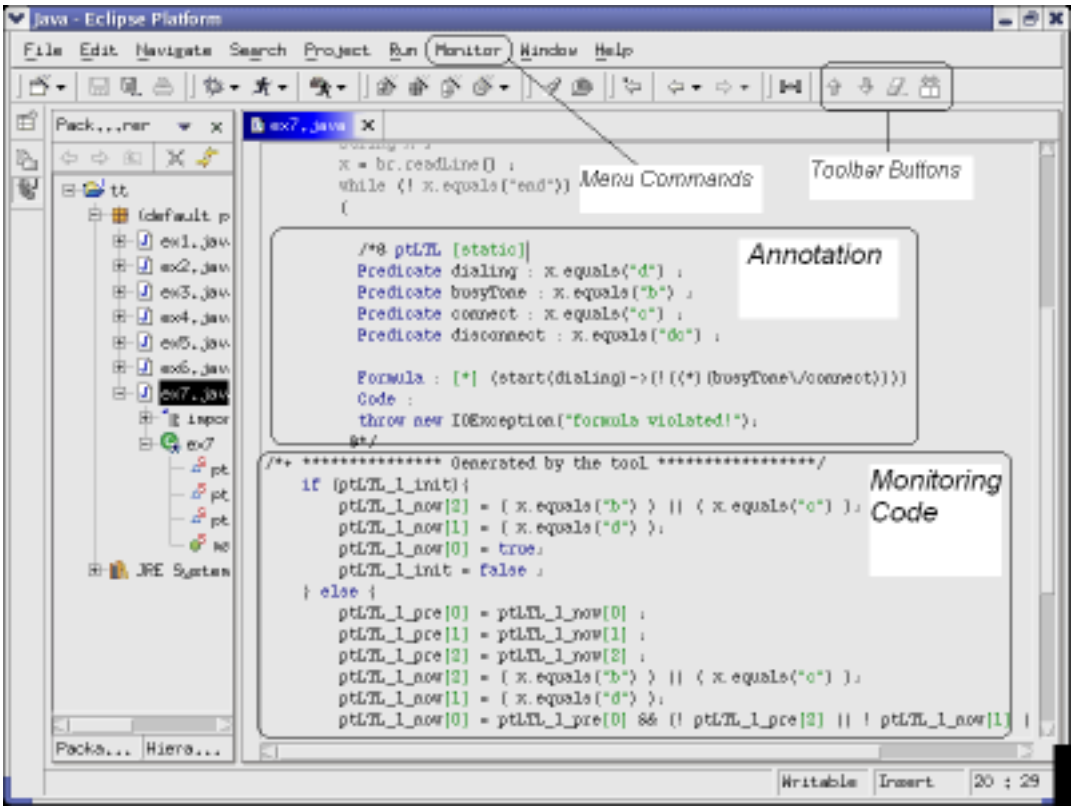

Figure 11: A snapshot of AJE.

files with our editor, called `Annotated` JAVA `Editor` and abbreviated AJE. Figure 11 shows a snapshot of an AJE session. The specification annotations, as well as the generated codes, are highlighted, while other parts of the source file are greyed. Users can navigate through annotations, generate or remove monitoring code by simple clicks or hot keys, interactively edit annotations and instantly check or modify the generated code.

4.2 Implementation

The architecture of JAVA-MOP, shown in Figure 10, is a specialization of the general MOP architecture in Figure 7. The client includes the interface modules and the JAVA annotation processor, while the server contains a message dispatcher and the logic plug-ins for JAVA. The message dispatcher takes charge of the communication between the client and the server, dispatching requests to corrensponding logic plug-ins. The communicatation is based on remote method invocation (RMI).

Both the client interface and the dispatcher are implemented in JAVA, while the logic plug-ins are currently written in diverse languages. We use Maude [8] to implement the logic engines for FTLTL, PTLTL, and ERE, because Maude turned out to be an elegant and efficient meta-logic development environment providing builtin capabilities for both parsing and transformation of logical formulae. The corresponding language shells for these three logics consist of Perl scripts, because their functions are straightforward and textual. The JASS plug-in is different. Since JASS' logic is very simple, no underlying logic engine is needed. However, the syntax of JASS' specification language is slightly more complex, so we had to build a parser for JASS annotations as part of its plug-in. One advantage of the flexible ASCII-text based protocols betweeen modules in MOP is that developers of logic plug-ins can freely choose their favorite or appropriate implementation languages, without worring of conflicts with the platform or the target language at the client's side.

```
aspect FooMonitoringAspect{
   ... other code ...
   pointcut allMethods(): execution (* Foo.*(..));
   before() : allMethods() {
      ... class invariant monitors ...
   }
   after() : allMethods() {
      ... class invariant monitors ...
   }
   ... other code ...
}
```

Figure 12: Pointcut and advices for class monitors

The annotation processor of JAVA-MOP currently uses ASPECTJ to integrate monitors whose specifications have the configuration attributes `class` (i.e., class invariants) or `pre-` and/or `post-method`. Precisely, we use the `exection` pointcut for methods and synthesize `before` and `after` advices from the generated monitoring code. Figure 12 shows part of the aspect generated to monitor class invariants in a class `Foo`. Figure 13 shows another aspect, this time generated from a method specification. If more than one specification relate to the same method, then all the monitors are aggregated and placed into one advice. For annotations inside methods, such as `block` annotations, the annotation processor analyzes the structure of the program to find out the exact position to place the monitoring code.

Those specifications having the default attribute in-line will be transformed into monitors that share the same state as the monitored program. Therefore, variables in the declaration part of the outputs generated by the corresponding logic plug-ins need to be added to the execution environment of the program. The annotation processor generates fresh names for all the variables added for monitoring purposes, thus avoiding any potential name conflicts. Another important task of the annotation processor is to provide program contex information to the monitoring code, as discussed in Section 3.1.3. This is done simply by replacing the JAVA-MOP predefined variables with their current values which can be easily found by introspecting the source code of the program, e.g., the current method name for `@METHOD_NAME_` or the current line number for `@LINE_NUMBER_`.

4.3 Logic Plug-ins

We next discuss the logic plugins that we have developed so far and that are currently supported by the JAVA-MOP prototype. The presentation in this section is rather informal and based on examples, but the appendix gives formal definitions and monitor synthesis algorithms. The interested reader is encouraged to check all the examples in this paper at our WWW plug-in repository (see Section 4.4), where one can generate and visualize monitors via an HTML client.

4.3.1 Design by Contract - JASS

As previously argued in the paper, since the various DBC approaches are based on the idea of including specifications into the code and then pre-compiling them into runtime checks, MOP is expected to smoothly include these approaches provided that appropriate logic plug-ins are developed. To test and evaluate this hypothesis, we have implemented a JAVA logic plug-in for JASS, so users of JAVA-MOP can state JASS annotations in their code. The original syntax of JASS annotations has been slightly modified, to fit the uniform, logic-independent syntactic conventions adopted in JAVA-MOP.

JASS supports the following types of assertions: method pre-conditions and post-conditions, loop variants and invariants, and class invariants. Pre-conditions must be satisfied by the caller but the callee is in charge of checking it. Post-conditions need to be satisfied at the exit of a method. Loop variants are integer expressions that must decrease at every iteration and whose evaluation should never reach 0. Such assertions are used to check the termination of loops. Finally, loop invariants must be satisfied at the boundaries of a loop and are useful to understand the semantics of loops. Class invariants are properties over the fields of an object and are checked on the boundaries methods.

```
aspect MathMonitoringAspect{
   ... other code ...
   //the Old is introduced as a new field of Math
   //and it can be also used in other monitors
   Math Math.Old;
   pointcut exec_square(int a) : argu(a)
      && execution(int Math.square(int));
   before(int a) : exec_square(a) {
      Old = (Math) this.clone();
   }
   after(int a) returning(int Result) :
      exec_square(a) {
      boolean JASS_1_isViolated;
      JASS_1_isViolated = false;
      int label;
      JASS_1_isViolated = !(x == Old.x);
      label = 0;
      if (!JASS_1_isViolated) {
         JASS_1_isViolated = !(Result == a * a);
         label = 1;
      }
      if (JASS_1_isViolated) {
         String labelMsg = "";
         switch (label) {
         case 0:labelMsg = "an-1"; break;
         case 1:labelMsg = "an-2"; break;
         default: break;
         }
         throw new RuntimeException("Method " +
              "int square(int)" +
              " violated specification " + labelMsg);
      }
   }
   ... other code ...
}
```

Figure 13: Aspect generated for Figure 6

Let us next consider the JASS annotation in Figure 6, referring to the previous state and also checking the computed result against the expected result. This example also shows how to use the information retrieved from the program context. Assume that the method `square` is defined in the class `Math` having a field named `x`, and that `Math` implements the interface `Cloneable`, a standard requirement of JASS in order for one to be allowed to access the previous state with `Old`. JAVA-MOP passes this annotation to the JASS plug-in, collects its output, and then generates the ASPECTJ aspect in Figure 13.

JASS therefore allows one to state properties characterizing not only the current state, but also its relationship to the previous one. While this is sufficient in many applications, there are many others where stronger relationships with other states are desired. We next present MOP plug-ins for more powerful logics, allowing one to refer to entire execution traces of states, including past and future ones.

4.3.2 Temporal Logics

Temporal logics have proved to be indispensable and expressive formalisms in the field of formal specification and verification of systems [33, 27, 28, 7]. Most practical safety properties can be naturally expressed using temporal logics, so these logics can also be very useful as requirements specification formalisms in an MOP software development framework. Since MOP can be regarded at some extent as a complementary, but still related, approach to formal verification, we provide logic plug-ins to support past and future time variants of temporal logics.

In future time linear temporal logic (FTLTL), one can state properties about the future execution of a system. Consider the example in Figure 1, which states a safety property of a traffic light controller using the FTLTL formula `[](green -> (! red U yellow))`. In this formula, `[] F` is read as *always* `F` and states that the property `F` should hold in any future state. `->` and `!` are the propositional logic operations *implies* and *not*. `F U F'` is read `F` *until* `F'` and states that `F'` should hold in some future state and `F` should hold in all the states until then. Therefore, the formula above states that "if the green light is on then the red light should not be turned on until the yellow has been turned on". Figure 2 already showed the final monitoring

```
// Declaration
   int $state = 1;
// Monitoring body
   switch($state) {
   case 1:
      $state = ( tlc.state.getColor() == 3) ? 1
         : ( tlc.state.getColor() == 2) ?
         ( tlc.state.getColor() == 1) ? -2 : 2 : 1 ;
      break ;
   case 2:
      $state = ( tlc.state.getColor() == 3) ? 1
         : ( tlc.state.getColor() == 1) ? -2 : 2 ;
      break ;
   }
// Failure condition
   $state == -2
```

Figure 14: Generated monitor for Figure 1

code generated from this specification. Figure 14 shows the output of the FTLTL logic plug-in, before it is merged within the original code by the annotation processor. Note that the additional variables in the generated code have generic names, starting with the `$` symbol; the annotation processor will replace these by appropriate fresh names, to avoid conflict with other variable names already occuring in the program.

```
// Declaration
   boolean $pre[] = new boolean[3];
   boolean $now[] = new boolean[3];
   fnct $start_backup = start(backup);
   fnct $end_logout = end(logout);
// Initialization
   $now[2] = ($end_logout) && !(ActiveUserNum > 0);
   $now[1] = ($start_backup);
   $now[0] = true;
// Monitoring body
   $pre[0] = $now[0] ;
   $pre[1] = $now[1] ;
   $pre[2] = $now[2] ;
   $now[2] = !(ActiveUserNum > 0)&&(($end_logout)||$pre[2]);
   $now[1] = ( $start_backup );
   $now[0] = $pre[0] && (! $now[1] || $pre[1] || $now[2]);
// Failure condition
   ! $now[0]
```

Figure 15: Generated monitor for Figure 5

Dually, in past time linear temporal logic (PTLTL), one can specify properties refering to the past states. PTLTL has been shown to have the same expressiveness as FTLTL [13]. However, it is exponentially more succinct than FTLTL [29] and often more convenient to specify safety policies. Figure 5 showed an example PTLTL specification of a safe backup. In that formula, `[*] F` is read *always in the past* `F` and states that `F` should hold in any past state; `[F,F')s` is read *strong interval* and holds if and only if `F` was true at some point in the past while `F'` has not been seen to be true since then, including that moment (there is also a weak version of interval). Therefore, the PTLTL property in Figure 5 says that the backup job can be start only when there is a logout action in the past, and after that moment, there has been no active user on the server. Figure 15 shows the generated monitor for this specification. The monitor uses two arrays, `new` and `pre`, to store the current program state and the past program state. Besides, there are two special variables, namely `start_backup` and `end_logout`, to record the execution of the methods. The JAVA annotation processor will instrument the corresponding methods to set the values of these variables during the execution.

4.3.3 Extended Regular Expressions

Software engineers and programmers can understand easily regular patterns, as shown by the immense interest in and the success of scripting languages like PERL. 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 (ERE) add complementation, or negation, to regular expressions, allowing one to specify patterns that must *not* occur during an execution. Complementation gives one the power to express patterns on traces non-elementarily more compactly. However, complementation leads to a non-elementary exponential explosion in the number of states of the corresponding automaton, so naive ERE monitoring algorithms may be impractical. Preliminary efforts in [38, 39] show how to generate simply exponential monitoring algorithms for ERE. A logic engine for ERE and a corresponding logic plug-in incorporating these algorithms has been implemented. As an example, Figure 16 shows the monitor generated for the safety policy in Figure 3, which is actually a finite state machine.

```
// Declaration
   int $state = 0;
   fnct $end_a = end(authentication());
   fnct $start_m = start(manageResource());
// Monitoring body
   switch($state) {
   case 0 :
      $state = ( $start_m ) ? 1 @OTHEREVENT ) ? 0 :-2 ;
      break;
   }
// Failure condition
   $state == -2
// Success condition
   $state == 1
```

Figure 16: Generated monitor for Figure 3

4.4 Online Logic Plugin Repository

Due to the standardized interface of MOP components, once a logic plug-in is developed, which in most cases is expected to be a highly nontrivial and time-consuming task, potentially any other MOP user can use it. To facilitate the reuse of logic plug-ins and to encourage users to propose, design, implement, and share them, we set up an online WWW repository of logic plug-ins which can be found at the URL `http://fsl.cs.uiuc.edu/mop`. The MOP user can try a logic plug-in online, via a provided HTML client, before she decides to download and install it. This repository is also implemented so that it can serve as a remote logic plug-in server, as discussed in Section 4.1. Thus, it is possible for the user to install few or no logic plug-ins on her machine, but to retrieve them by need from the repository. This facilitates the usage of the MOP environment, because the user does not need to search, download, compile and install all wanted plug-ins on her own server.

5 Conclusion and Future Work

We presented monitoring-oriented programming (MOP) as a general object-oriented development paradigm, aiming at increasing the quality of software by merging formal specifications into programs via monitors sythesized and integrated automatically. Specification formalisms can be easily added to and removed from an MOP supporting system, due to logic plug-ins. A WWW repository of logic plug-ins allows MOP users to download or upload them, thus avoiding the admitedly heavy task of developing their own logic plug-ins. The logic-independece and the broad spectrum of generated monitor integration capabilities allow MOP to paradigmatically captuare other related techniques as special important cases. We also introduced JAVA-MOP, a prototype system supporting most of the desired features of MOP.

There is much future work to be done, part of it already started. We intend to extend our prototype to support all the running modes soon, to incorporate other related approaches, such as JML, and especially to evaluate the MOP paradigm on students and in practical large size applications. One very technical and urgent problem that needs to be solved is the limitation of AOP tools like `AspectJ` w.r.t. to strongly checking class invariants (i.e., checking them at each state change rather than only at the boundaries of

methods). We believe that MOP may also lead to novel programming techniques that can provably increase the quality of software by enabling more scalable verification and validation methodologies.

Acknowledgment. We thank Ralph Johnson for useful comments and insights on a previous version of this paper.

References

[1] P. Abercrombie and M. 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] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Păsăreanu, G. Roşu, and W. Visser. Experiments with test case generation and runtime analysis. In *Abstract State Machines Workshop (ASM'02)*, volume 2589 of *Lecture Notes in Computer Sciences*, pages 87–107. Springer-Verlag, 2003.

[3] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Păsăreanu, G. Roşu, W. Visser, and R. Washington. Automated testing using symbolic execution and temporal monitoring. *Theoretical Computer Science*, to appear, 2004.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation(VMCAI'04)*, Lecture Notes in Computer Science, 2004.

[5] F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Proceedings of the 3rd Workshop on Runtime Verification (RV'03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 106–125. Elsevier Science, 2003.

[6] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In *Software Engineering Research and Practice (SERP'02)*, pages 322–328. CSREA Press, 2002.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual. June 2003.

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

[10] D. Drusinsky. Temporal rover. http://www.time-rover.com.

[11] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.

[13] D. M. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Proceedings of the 1st Conference on Temporal Logic in Specification*, volume 398 of *LNCS*, pages 409–448. Springer, 1989.

[14] J. Guttag and J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[15] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[16] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. IEEE, 2001.

[17] K. Havelund and G. Roşu. *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001. Proceedings of a *Computer Aided Verification (CAV'01)* satellite workshop.

[18] K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002. Proceedings of a *Computer Aided Verification (CAV'02)* satellite workshop.

[19] K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, to appear.

[20] K. Havelund and G. Roşu. *Runtime Verification*. Kluwer, to appear. Special issue of *Formal Methods in System Design* dedicated to RV'01.

[21] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. EASST best paper award at ETAPS'02.

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

[23] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[24] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[25] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, 2000.

[26] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[28] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.

[29] N. Markey. Temporal logic with past is exponentially more succinct. *EATCS Bull*, 79:122–128, 2003.

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

[31] E. Org. Aspectj project. http://eclipse.org/aspectj/.

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

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

[34] 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. IEEE Computer Society Press, 1990.

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

[36] G. Roşu and K. Havelund. Synthesizing Dynamic Programming Algorithms from Linear Temporal Logic Formulae. RIACS Technical report TR 01-08, January 2001.

[37] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, to appear.

[38] G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 499–514. Springer-Verlag, 2003.

[39] K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Proceedings of the 3rd Workshop on* Runtime Verification (RV'03), volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 162–181. Elsevier Science, 2003.

[40] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of 4th joint* European Software Engineering Conference *and* ACM SIGSOFT Symposium on the Foundations of Software Engineering, (ESEC/FSE'03). ACM, 2003.

[41] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, Lecture Notes in Computer Science. Springer, 2002. to appear.

[42] L. Sha, R. Rajkumar, and M. Gagliardi. The simplex architecture: An approach to build evolving industrial computing systems. In *Proceedings of The ISSAT Conference on Reliability*, 1994.

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

[44] O. Sokolsky and M. Viswanathan. *Runtime Verification 2003*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003. Proceedings of a *Computer Aided Verification (CAV'03)* satellite workshop.

[45] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

In this appendix, we present the logics supported in JAVA-MOP, and briefly introduce the monitor generation algorithm implemented in corresponding logic engines.

A Temporal Logics

A.1 Future Time Linear Temporal Logic

Syntax and Semantics. FTLTL has the following constructors:

$$\begin{array}{rll} F ::= & \mathit{true} \mid \mathit{false} \mid A \mid \neg F \mid F \ \mathit{op}\ F & \text{Propositional ops} \\ & \Box F \mid \Diamond F \mid F \ \mathcal{U}\ F \mid \circ F & \text{Future time ops} \end{array}$$

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 $t : Nat^+ \rightarrow 2^{\mathcal{P}}$ for some set of atomic propositions $\mathcal{P}$. This function maps each time point (a natural number) into the set of propositions that hold at that point. That is, t models an infinite trace over the alphabet $2^{\mathcal{P}}$. 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 definitely holds in some future time point, and until then X holds (the weaker version can be defined in terms of $\mathcal{U}$ and $\Box$). 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 $\Box(X \rightarrow \Diamond Y)$ is true if for any time point ($\Box$) it holds that if X is true then eventually ($\Diamond$) Y is true. Note that it is possible that X never holds and then the formula is vacuously satisfied.

We next show how the FTLTL monitors based on binary transition tree finite state machines (BTT-FSM) [15] are generated by an appropriate logic engine in our MOP environment.

Algorithm. It is shown in [15, 19] 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. This structure is specifically tuned for monitoring purposes. That is, they aim at reducing 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, there is no possible continuation of the observed sequence of states which would form a model satisfying the formula. In other words, the interpretation of the formula stabilizes once a computation reaches one of these states.

The BTT-FSM for the traffic light controller example shown in Figure 1, whose formula is $\Box(green \rightarrow \neg(red)\ \mathcal{U}\ yellow)$, can be seen in Figure 17. The formula reflects the requirement "after green yellow comes" .

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 referred to [15] for more details.

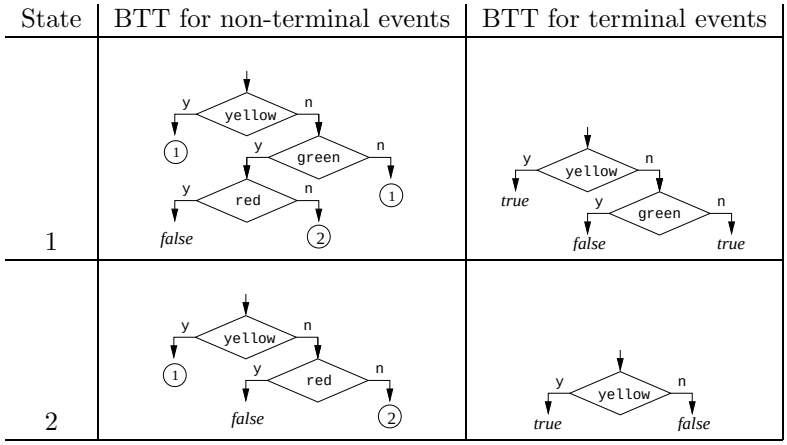

Figure 17: A BTT-FSM for the formula $\Box(green \rightarrow \neg(red\ \mathcal{U}\ yellow))$

The logic module implemented in our 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 there might exist for some formulae, such as $\diamond F$. The size of these monitors can be exponential (on 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.

A.2 Past Time Linear Temporal Logic

Syntax and Semantics. We allow the following constructors for PTLTL formulae, where A is a set of "atomic propositions":

$$\begin{array}{lll} F ::= & \mathit{true} \mid \mathit{false} \mid A \mid \neg F \mid F \ \mathit{op}\ F & \text{Propositional oper.} \\ & \circ F \mid \diamond F \mid \boxdot F \mid F \ \mathcal{S}_s \ F \mid F \ \mathcal{S}_w \ F & \text{Past time oper.} \\ & \uparrow F \mid \downarrow F \mid [F,F)_s \mid [F,F)_w & \text{Monitoring oper.} \end{array}$$

The propositional binary operators, *op*, are the standard ones, that is, disjunction, conjunction, implication, equivalence, and exclusive disjunction ($\vee$,$\wedge$,$\rightarrow$,$\leftrightarrow$,$\oplus$). The standard past time and the monitoring operators are called "temporal operators", because they refer to other (past) moments in time. The operator $\circ\ F$ 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 on runtime verification in [26]. 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 as "start F"; it says that the formula F just started to be true, that is, it was false previously but it is true now ($F \wedge \odot\neg F$). 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. For example, in the phone system application presented later, the formula $\boxdot(\uparrow(\mathit{dialing}) \rightarrow \neg \circ (\mathit{busyTone} \vee \mathit{connected}))$, states that one cannot dial when the phone is busy or connected. In the traffic light controller, the formula $\boxdot(\neg(\mathit{red} \circ \mathit{green}) \wedge \neg(\mathit{yellow} \circ \mathit{red}))$ expresses the requirement that a green never precedes a red light, and a red never preceeds a yellow.

The operators $[F_1, F_2)_s$ and $[F_1, F_2)_w$ are read "strong and weak interval F_1, F_2, respectively". For the strong case the semantics is as follows: 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. The weaker version does not enforces that sometime F_1 was true. For example, the formula $[\mathit{startup}, \mathit{shutdown})_s$ states that at some time in a server was started and since then no shutdown was observed.

In [21], a dynamic programming algorithm is defined that generates monitors for PTLTL formulae. What follows shows how such generator can be cast as a logic plugin in MOP.

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. Intuitively, we verify if a formula P is not violated based on the history. Suppose it is not violated up to some point in time. In the next input, we just need to verify if the formula is violated in the last step and if the current transition violated the formula.

For example, according to the formal, nonrecursive, semantics, a trace $t = s_1 s_2 ... s_n$ satisfies the formula $[F_1, F_2)_w$ 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 $\neg(\ t \models F_2)$. Otherwise, if the trace has more than one event then first of all $\neg(\ t \models F_2)$, 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.

Efficient monitors can be generated based on the recursive semantics of PTLTL. 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 $update$: **State** $\times$ *Event* $\rightarrow$ **State**, which updates the state when provided with a given event. According to the recursive semantics, one can use two boolean arrays to model the interpretations of the formulae at the present and at the last step: $now[8]$ and $pre[8]$. $now[i]$ is true if and only if the current φ_i is true, and $pre[i]$ is true if and only if the φ_i was true in the last step. The algorithm in [21] is implemented as a logical engine that generates the following output for the formula above. Expressions of the form $s(state)$ denote a boolean value that a state predicate s is evaluated:

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 defined with the intention to be always satisfied, so there is no success condition for PTLTL. The above program can be further optimized, considering that only three formulae need to be

remembered to calculate the next states: namely $pre[6]$, $pre[3]$ and $pre[2]$. The values of formulae whose pre interpretations are not used can be replaced. By applying this rule followed by boolean simplifications, we end up in the following optimized monitor body, which is the real output of our logic module:

$state \leftarrow update(state, s_j)$
$now[6] \leftarrow r(state)$ *or* $s(state)$
$now[3] \leftarrow (pre[3]$ *or* $q(state))$ *and* $(now[6]$ *or not* $pre[6])$
$now[2] \leftarrow p(state)$
if $(now[2]$ *and not* $pre[2]$ *and not* $now[3])$
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.

B Extended Regular Expressions

Syntax and Semantics. 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 an expression R, denoted by $\mathcal{L}(R)$, is defined inductively as

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset, \\
\mathcal{L}(\epsilon) &= \{\epsilon\}, \\
\mathcal{L}(A) &= \{A\}, \\
\mathcal{L}(R_1 + R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2), \\
\mathcal{L}(R_1 \cdot R_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\}, \\
\mathcal{L}(R^\star) &= (\mathcal{L}(R))^\star, \\
\mathcal{L}(R_1 \cap R_2) &= \mathcal{L}(R_1) \cap \mathcal{L}(R_2), \\
\mathcal{L}(\neg R) &= \Sigma^\star \setminus \mathcal{L}(R).
\end{aligned}$$

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.

Algorithm. A simple, straightforward, and practical approach to monitor EREs is to generate optimal deterministic finite automata (DFA) from EREs [22]. 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 DFA 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 regular 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:

$$
\begin{array}{lllll}
(1) & (R_1 + R_2)\{a\} & = & R_1\{a\} + R_2\{a\} \\
(2) & (R_1 \cdot R_2)\{a\} & = & \\
 & (R_1\{a\}) \cdot R_2 + \texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi} & & \\
(3) & (R^\star)\{a\} & = & (R\{a\}) \cdot R^\star \\
(4) & (\neg R)\{a\} & = & \neg(R\{a\}) \\
(5) & b\{a\} & = & \texttt{if } (b == a) \texttt{ then } \epsilon \texttt{ else } \emptyset \texttt{ fi} \\
(6) & \epsilon\{a\} & = & \emptyset \\
(7) & \emptyset\{a\} & = & \emptyset
\end{array}
$$

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 derivatives 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 DFA.

Experiments with this logic engine are very encouraging. Our implementation, which is also available graphically on the Internet via a CGI server(`http://fsl.cs.uiuc.edu/rv`), rarely took longer than one second to generate a DFA. For example, the optimal DFA 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 detached in Figure 18. Basically, stating that it is not possible to have a green followed by a red in the middle of any trace.

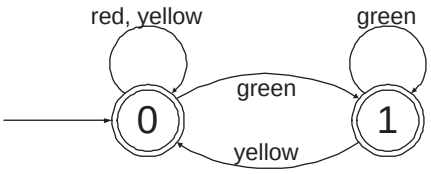

Figure 18: The optimal BTT-FSM for ¬ ((¬ `empty`) (`green red`) (¬ `empty`))

Similarly for the FTLTL generated automaton, the output of the ERE logic engine can be generated by inspection on the automaton definition:

Declarations. *integer state*

Initialization. $state = 0$

Monitoring Body. The following case statement:

$case(state)$
$0 \ : \ state = (yellow \vee red) \ ? \ 0 \ : \ green \ ? \ 1 \ : \ -1;$
$1 \ : \ state = green \ ? \ 1 \ : \ yellow \ ? \ 0 \ : \ -1;$

Failure Condition. $state = -1$