Abstract
Monitoring-Oriented Programming (MOP) is a formal framework for software development and analysis. It aims at reducing the gap between formal specification and implementation via runtime monitoring. In MOP, the developer specifies desired properties using definable specification formalisms, along with code to execute when properties are violated or validated, which can be used not only to report, but especially to recover from errors. The MOP framework automatically generates monitors from the specified properties and then integrates them together with the recovery code into the original system. Since the recovery code typically is executed infrequently and can be validated more easily than the actual system, MOP is expected to increase software reliability at little amortized runtime overhead. This paper presents MOP from a pragmatic, rather than foundational perspective, as an instance of aspect-oriented programming (AOP) where one defines abstract aspects using logics; one is relieved from providing unnecessary implementation details, because these are generated and integrated automatically. Existing AOP tools provide crucial support: an MOP frontend for Java, called JavaMOP and also discussed in the paper, is implemented using AspectJ. A series of examples illustrate the strengths of MOP from different perspectives.

1. Introduction
Despite significant foundational and tool support advances in software bug detection, we are still far from developing error-free software and probably so will be for many years to come. Testing is effective in hunting bugs in practice, but its ad-hoc nature provides no correctness guaranties. On the other hand, traditional formal analysis methods, such as model checking, theorem proving, formal static analysis, as well as combinations of these, attempt to systematically and exhaustively check software; unfortunately, in spite of several successful applications and tools (e.g., [6, 30, 24]), formal methods still find a limited use in mainstream software development, essentially because of their reputation to not scale up well (including notorious state-explosion aspects, need for non-trivial user-provided code annotations, abundance of false alarms).

Runtime verification [26, 41, 7] is a relatively new formal analysis approach that aims at combining testing with formal methods in a mutually beneficial way. The idea underlying runtime verification is that system requirements specifications, typically formal and referring to temporal behaviors and histories of events or actions, are rigorously checked at runtime against the current execution of the program, rather than statically, against all hypothetical executions. If used for bug detection, runtime verification gives a rigorous means to state and test complex temporal requirements, but, like testing, still suffers from limited coverage. Runtime verification is particularly appealing when combined with test case generation [3] or with steering of programs [32]. Like in any new domain, there is a variety of runtime verification techniques, algorithms, formalisms, foundations and tools; some of these are discussed in Section 5.2, others can be found in the proceedings of the RV meetings [26, 41, 7].

Monitoring-Oriented Programming (MOP) was proposed in [13, 10, 11] as a programming paradigm built upon runtime verification intuitions and techniques, aiming at supporting reliable software development via monitoring and recovery. MOP takes monitoring of system requirements as a fundamental software development principle, providing an extensible formal framework to combine implementation and specification. The user of MOP specifies desired properties using definable formalisms. Monitoring code is then automatically generated from properties and integrated into the original program; the role of the monitoring code is to verify the runtime behavior of the system against the specified properties. Traditional runtime verification approaches mainly focus on detecting violations. MOP goes one step beyond that by allowing and encouraging the user to provide code to execute when properties are violated or validated, which can be used not only to report but especially to recover from errors at runtime.

Let us consider the following example about a simple and common safety property for a shared resource, namely that any access to the resource should be authenticated. For simplicity, suppose that all the operations on the shared resource are implemented in the class Resource, including methods access() and authenticate(). Then the safety property can be specified as a trivial regular expression over method invocations:

```
authenticate (authenticate + access)*
```

Figure 1. MOP specification for resource safety
The first line of the MOP specification in Figure 1 states that this property is a class invariant, i.e., it should hold in the scope of the class (specification attributes are discussed in Section 3.1). The second line chooses a desired formalism to express the corresponding formal requirement, in this case extended regular expressions (ERE); MOP allows users to “plug-and-play” new specification formalisms, provided that they respect the standardized interface of logic plug-ins (these are discussed in Section 2.1). The content enclosed by the curly brackets is specific to the chosen formalism. For EREs, the user needs to first build an abstraction that maps runtime events into logical elements, e.g., the invocation of authenticate() being mapped to an atom authenticate. Using the elements produced by the abstraction, a regular expression is given to describe the desired property. The last part of the MOP specification contains the code that will be triggered when the specification is violated and/or validated. It may be as simple as reporting errors, or as sophisticated as taking recovery actions to correct the execution to avoid crashes of the system. In this example, when the safety property is violated, i.e., when some access is not authenticated, we enforce the authentication simply by making a call to authenticate(). The MOP tool is able to analyze this specification, generate monitoring code for the regular pattern, and insert the monitor with the recovery handler into appropriate points of the program, namely, at the beginning of the access() method and at the end of the authenticate() method.

There are two important observations to make regarding the example above, each reflecting a crucial characteristic of MOP:

1. By generating the monitoring code automatically from the desired property and by integrating it at the relevant points in the program together with corrective code in case of property violation, the developer can and should have quite a high confidence that the resource is used correctly throughout the system. In fact, if we trust that the MOP tool generates and integrates the monitoring code correctly, then we can also trust that the resulting system is correct w.r.t. this safety property, no matter how complicated the system is. This informal reasoning can be actually formalized and the resulting program verified automatically in many cases, but this will be discussed elsewhere.

2. Suppose that authentication-before-access was not a requirement of the system originally, but that it became a desired feature later in the development process (e.g., because of an increasing number of clients). Suppose also that, as a consequence, one wants to add authentication to an initial implementation of the system that provided no support and no checking for authentication. Using MOP, all one needs to do is to add an (unavoidable) authenticate() method, together with the MOP specification in Figure 1. This way, the MOP specification together with its violation handler added non-trivial functionality to the system. It is in fact hard to imagine any faster or more elegant way to add such non-trivial functionality to a system, even when one makes complete abstraction of I., the high-reliability aspect of the extension.

Monitors corresponding to specifications may need to observe the execution of the program at many different points; for example, the monitor for the regular pattern above needs to observe all the ends of authenticate() and all the beginnings of access(), all points which can be scattered all over the system. In this sense, every monitor can be regarded as a crosscutting feature in aspect-oriented programming (AOP) [31]. AOP aims at separation of concerns by allowing programmers to extract and encapsulate as separate modules, called aspects, features that are conceptually scattered in different components of the system, and then to merge them into the original program using (AOP) compilers. MOP can be regarded as a specialized instance of AOP, in which aspects are formal specifications instead of modules of ordinary code. Existing AOP tools provide crucial support for MOP to integrate generated monitoring code as well as recovery code into the system. From this point of view, MOP acts as a supplier of aspects: it converts the abstract specifications into concrete aspects that can be handled by existing AOP tools. For instance, our MOP frontend for Java discussed in Section 2.2, JavaMOP, is built on top of AspectJ [4]. Using JavaMOP, the specification in Figure 1 is translated into the AspectJ code in Figure 2, where ERE_00 is used to encode a state machine for verifying the regular expression.

Figure 2. MOP specification for resource safety
as well as its relationship to runtime verification; we briefly summarize and review these in Section 2. This paper discusses MOP from a more pragmatic perspective: (1) It presents it as an instance of AOP where one specifies abstract aspects using logics; (2) It presents an improved MOP specification language, based on a revised implementation of JavaMOP that takes advantage of AOP capabilities; and (3) It shows a series of uses of MOP that illustrate its strengths in building reliable software from different perspectives. Section 5 discusses related work and Section 6 concludes the paper.

2. Overview of MOP and JavaMOP

We here briefly introduce MOP and JavaMOP. Interested readers are referred to [11, 10] for more details, and also to [12] for tool downloads and the latest development news.

2.1 Extensible MOP Framework

The MOP framework separates monitor generation and monitor integration by adopting the layered architecture in Figure 3. This architecture is especially designed to facilitate extending the MOP framework with new formalisms or new programming languages. By standardizing the protocols between layers, new modules can be added easily and independently. Modules on lower layers can be reused by upper-level modules.

The topmost layer, called the interface layer, provides user friendly programming environments. For example, the reader is encouraged to try the web-based interface of JavaMOP at [12] (no download needed, examples provided). The second layer contains specification processors, which take charge of monitor integration. Each specification processor is specific to a target programming language and consists of a program scanner and a program transformer. The scanner extracts MOP specifications from the program and dispatches them to appropriate modules on the lower layer to process. The transformer then collects the monitoring code generated from the lower layer and integrates them into the original program. AOP plays a critical role here: the program transformer synthesizes AOP code and invokes AOP compilers to merge the monitors within the program. In particular, as discussed in Section 2.2, JavaMOP transforms generated monitoring code into AspectJ code. This way, MOP can be used with any programming languages that provides AOP support and tools.

The two lower layers contain logic plugins, which allow the user to add, remove, or modify formal specification formalisms. Logic plugins are usually composed of two modules: a language shell on the third layer and a logic engine on the bottom layer. The former generates programming language specific and specification formalism specific monitoring code in a standardized format, which can be understood by the specification processor on the upper layer. The logic engine, acting as the core of monitor generation, synthesizes monitors from specifications in a programming language independent way, e.g., as state machines. This way, logic engines can be reused across different programming languages.

The most important inter-layer protocol in this architecture is the one between the specification processors and the logic plugins, which determines the extensibility of the framework. The input to the logic plugin is simply the logic-specific content of the specification, e.g., the content between the curly brackets in Figure 1, while the output is standardized to consist of the following dimensions.

Monitored variables. Fields in the class, whose updates should be monitored.

Monitored events. Events to monitor along with associated actions, defined in the following format:

\[ <eventName> \ [event definition] \ [actions] \]

The event definition follows the syntax in Figure 7.

State declarations. Variables to maintain relevant program state information for the next step of monitoring. These variables need to be declared as new fields in the corresponding class.

Initialization. A fragment of program to initialize the monitor state.

Monitoring body. The main part of the monitor, which is executed any time an observation point is reached.

Local declarations. Temporary variables needed by the monitor within one step of the verification process. These variables are not stored for the next monitoring step.

Success condition. The condition stating that the monitored requirement has been fulfilled. When this condition becomes true, the user provided validation handler will be executed.

Failure condition. The condition that holds when the requirement is violated. When this condition becomes true, the user provided violation handles will be executed.

For example, Figure 4 shows the output of the Java-ERE logic plugin for the specification in Figure 1. Some variables in this output start with $ or #; these need to be renamed by the specification processor to avoid potential naming conflicts.

2.2 JavaMOP

JavaMOP is an MOP development tool for Java. It provides several interfaces, including a web-based interface, a command-line interface and an Eclipse-based GUI, providing the developer with different means to manage and process MOP specifications. To flexibly support these various interfaces, as well as for portability reasons, we designed JavaMOP following the client-server architecture in Figure 5, which is an instance of the general MOP architecture in Figure 3. The client part includes the interface modules and the JavaMOP specification processor, while the server contains a message dispatcher and logic plug-ins for Java. The specification processor employs AspectJ for monitor integration. In other words, JavaMOP translates outputs of logic plugins into AspectJ code, which is then merged within the original program by the AspectJ compiler. The message dispatcher is responsible for the communication between the client and the server, dispatching requests to corresponding logic plug-ins. The communication can be either local or remote, depending upon the installation of the server.

Note that the efficiency of JavaMOP has nothing to do with the runtime overhead of the resulting system, the same way that the efficiency of a compiler has nothing to do with the performance of the...
This architecture provides a more portable tool, since the client and development tool which runs best on Linux and Unix. Therefore and ERE, are implemented in Maude [15], an efficient meta-logic some of the logic engines, namely those for linear temporal logics clients are implemented in Java to run on different platforms, while one logic server can provide and cache monitor generation, which compiled code. In particular, the client-server architecture of JavaMOP and the remote communication add no performance penalty on the generated system. The advantage of this architecture is that one logic server can provide and cache monitor generation, which can require intensive computation, for multiple clients. Besides, our clients are implemented in Java to run on different platforms, while some of the logic engines, namely those for linear temporal logics and ERE, are implemented in Maude [15], an efficient meta-logic development tool which runs best on Linux and Unix. Therefore this architecture provides a more portable tool, since the client and the server are allowed to run on different platforms and the server can cache monitors for common formulae.

Three kinds of specification languages are currently supported in JavaMOP: Java Modeling Language (JML) [33], Extended Regular Expressions (ERE) and (Past-Time and Future-time) Linear Temporal Logic (LTL). We next introduce them informally.

JML. JML provides a comprehensive modeling language with some features that are difficult, sometimes almost impossible, to monitor (e.g., the assignable clause) [33]. We defined only those features supported by the JML runtime checker in [14], including method specifications, type invariants, and historic constraints. We currently do not support abstract specifications, i.e., ghost variables and model fields (but they can be easily incorporated since declaring and using variables inside specifications is also supported in JavaMOP). Since JML is specific to Java and uses Java syntax, its monitor synthesis procedure does not involve complex logic inference and simplifications, so a separate logic engine is unnecessary.; the logic plug-in for JML consists only of the language shell.

ERE. Regular expressions provide an elegant and powerful specification language for requirements, because an execution trace of a program is in fact a string of states. The advantage of regular expressions over many other logics is that they are a standard notation to which many programmers have already been exposed. Extended regular expressions (EREs) add complementation, or negation, to regular expressions, allowing one to also specify patterns that must not occur during an execution. Complementation gives one the power to express (non-elementarily) more compactly patterns on traces. However, complementation leads to a non-elementary exponential explosion in the number of states of the corresponding automaton if naive ERE monitoring algorithms are used. An efficient ERE monitoring algorithm (described in [39, 40]) has been implemented as a JavaMOP logic plugin.

LTL. Temporal logics [36] prove to be favorite formalisms for formal specification and verification of systems. Safety properties can be naturally expressed using temporal logics, so these logics can also be very useful in MOP. Building upon LTL monitor synthesis algorithms in [27, 38, 18], we implemented logic engines and corresponding Java language shells to support future-time (FTLTL) and past-time (PTLTL) variants of temporal logics.

## 3. MOP Specification Language

MOP provides a specification language to define desired properties. The design of this language is mainly driven by two factors: uniformity in supporting different formalisms and programming languages, and the ability to control monitor behaviors. Programming-language-specific and logic-specific contents are carefully distinguished from other generic contents in MOP specifications. This results in a general specification schema that can be ported to different programming languages, while the developer is still allowed to fully configure monitors using various attributes. Efforts have also been made to increase the expressibility and programming capability of MOP’s specification language. Currently, the MOP specification language can be regarded as a specialized AOP language, tuned to support specifying formal abstract aspects.

MOP specifications can be either embedded into the source code as special annotations or stored in separate specification files. Both formats hold different advantages. Annotations are more suitable for properties related to specific positions in the source code, e.g., assertions and pre-/post-conditions for methods. On the other hand, separate specification files are conceptually clearer when their corresponding properties involve multiple points of the program, e.g., class invariants. JavaMOP supports both kinds of specifications.

Figure 6 shows the syntax of MOP specifications. In this syntax, a MOP specification is composed of three parts: the header, the body and the handlers. We next discuss each of these in more detail.

```java
class invariants. JavaMOP supports both kinds of specifications.

Figure 6 shows the syntax of MOP specifications. In this syntax, a MOP specification is composed of three parts: the header, the body and the handlers. We next discuss each of these in more detail.
The header contains generic information to control monitor generation and integration, consisting of optional attributes, the scope and the name of the specification, as well as the name of the formalism used in this specification, i.e., the unique name identifying the corresponding logic plug-in.

Attributes are used to configure the monitor with different installation capabilities. They are orthogonal to the actual monitor generation but determine the final code generated by the MOP tool. Four attributes are available. One is static, saying that the specification refers to the class, not to the object. For a static specification, only one monitor instance is generated at runtime and is shared by all the objects of the corresponding class. By default, monitors are non-static, meaning that every object will be monitored individually. In JavaMOP, the variables used to represent the state of the monitor, i.e., those declared in the “State declarations” section of the logic plugins’ output, will be added to the corresponding class as either static or non-static fields, according to staticness of the monitor. In JavaMOP, inserting new class fields is done through the inter-type member declaration of AspectJ (e.g., the declaration of Resource.ERE).

Two other attributes, outline and offline, are used to choose the running mode of the monitor. An important observation in practice is that different properties may require different running modes of monitors. For example, a monitor can be executed in the context (thread) of the monitored system, or it can run outside of the monitored system, as a standalone process or thread. We call the former the inline monitor, which is also the default mode of the specification, and the latter the outline monitor. The inline monitor can interact with the system directly, facilitating information retrieval and error recovery, but some problems, e.g., deadlocks, cannot be detected by inline monitors. Besides, inline monitors may cause significant runtime overhead when the runtime verification involves intensive computation. The outline monitor provides a better solution for such cases. In the outline mode, the monitored system sends messages that contain necessary information about the system to the monitor when a relevant event is encountered. However, communication with outline monitors can reduce the performance of the system and, equally importantly, the outline monitor cannot access the internal state of the monitored system, limiting its capability of error recovery. Usually, the outline monitor may try to fix detected problems by stopping/resetting the monitored program or releasing certain resources.

In addition to inline and outline monitoring, there is yet another useful way to check execution against specifications, sometimes the only feasible way: to log the trace in a file and then to make it available to the “monitor”. Since such monitors can run after the monitored system ceases, they are called offline monitors. Offline monitors are suitable for some properties that can be decided only after the system stops or properties that requires a backwards traversal of the trace; they may also be useful for debugging and analysis purposes.

These running modes impose different requirements on monitor synthesis. More specifically, in JavaMOP, inline monitors are merged into the original program as new pieces of code. This is achieved by encapsulated the entire monitoring code as an aspect, such as the example in Figure 1 and Figure 2. For outline and offline monitors, a standalone monitor class is synthesized to carry out the verification process, which can run independently as a new thread or process. The MOP tool then generates aspects containing either message passing code or event logging code based on the mode of the monitor.

The last attribute, async, can only be combined with the outline mode of the monitor. It requires the monitor to run asynchronously. When omitted, the monitor runs in synchronized mode, forcing the system to wait until the monitor finishes its work.

The scope of specifications defines the working scope of monitors, determining the points at which properties are checked. Four scopes are currently supported: class, interface, method, and the default scope. The scope class means that the property is a class invariant and should be checked whenever the involved fields are updated or the involved methods are called. However, to locate all possible updates on an arbitrary variable requires precise alias analysis, which is an undecidable problem. JavaMOP only supports tracking updates on fields of primitive types, which is achieved using pointcuts in AspectJ. The scope interface denotes a constraint on the interface, which should be checked at every observable state change, specifically on boundaries of public method calls; an interface-scoped property in MOP is therefore similar to a class invariant in JML [33]. The scope method is used to specify constraints on the designated method, including pre-, post-, and exceptional conditions. The default scope is “assertions”: monitoring code is placed inside the source code and checked whenever hit during the execution.

The name of the specification is an optional item in the header, which can be useful for documentation purposes or as a reference. When a defaultScoped specification is written outside of the source code, it has to be named to allow the developer to refer to it in the source code, where it needs to be checked. In JavaMOP, one can use //@ specification-name to refer to a named specification in a Java class.

The logic name is the last and most important item in the header, designating the formalism to use in the specification and also identifying the corresponding logic plugin. One may even have multiple logic plugins to generate monitoring code for one logic formalism, if different algorithms have been devised. In such cases, these logic plugins should have different names even though they are for the same logic, to allow the MOP tool to invoke the appropriate logic plugin. Presently, the following logic names can be used in JavaMOP: JML, ERE, PTLTL and PTLTL.

3.2 Body: Describing Properties

The body of an MOP specification formally defines the desired property, which will be extracted and sent to the corresponding logic plugin by the specification processor. Considering the diversity of specification formalisms, it is difficult, and also undesired, to design a uniform syntax for all possible formalisms. So the syntax of the specification body varies with the underlying formalisms. For JML, we adopt its original syntax except for the format of annotations. This is based on the consideration of compatibility: now one...
can translate JML specifications into MOP specifications simply by changing their headers and providing necessary handlers. For those formalisms that are used to express properties over traces, including ERE and LTL, we designed a general syntax for all of them, as shown in Figure 7, since they share many common features of runtime monitoring. In this syntax, the body is composed of an optional block for local variable declarations, a list of event definitions and a formula specifying the property. We start the detailed explanation with the two required parts, i.e., event definitions and formula, because they play the major roles in the specification.

For example, JavaMOP uses the syntax of AspectJ in the event definitions and then uses as atoms in the formula that describes the desired property rigorously. During monitor synthesis, the language shell extracts and sends the formula to the logic engine, which then generates the monitoring code from the formula. The monitoring code generated by the logic engine can be some pseudo code that is independent of any specific programming language. It will then be translated into the target language by the language shell. Therefore, the syntax of the formula varies with the formalisms.

Additional programming capabilities are also supported in order to strengthen the expressiveness of MOP specifications. The developer can declare local variables in the specification and associate actions to event definitions. The declared variables are called monitor variables, which are only visible inside the monitor. Specifically, the monitor variables can be used in event actions and the handlers discussed below. Event actions are the code associated to specific events, which will be executed upon occurrences of the corresponding events. Based on monitor variables and event actions, one can specify more complicated properties and/or implement more powerful handlers. For example, one may add counters into regular expressions to express properties like $AB^3A$. Figure 15 shows another example for employing monitor variables in the handlers. It is worth noting that the events and predicates defined in the specification body are also monitor variables, meaning that they can be used in event actions and handler as boolean variables, as shown in Figure 8 and Figure 16.

### 3.3 Handlers: Taking Actions

The major contribution of MOP that distinguishes it from traditional runtime verification techniques is the support for handlers. Specifically, MOP allows the developer to provide special code that will be executed when the property is violated or validated. Such code is called the handler in MOP specifications. Although many errors are related to violations of specifications, it can be simpler to describe erroneous behaviors than to define desired properties in some cases, e.g., patterns of security attacks. Therefore, handlers can be associated to not only violations but also validations of properties. Besides, even though handlers support runtime error recovery naturally, they are not necessary error recovery code; in fact, they can be any actions to trigger according to the properties. In this sense, an MOP specification can be regarded, which can refer to past and future events, as a complicated branch statement with the specified property as the condition and the handlers as true/false branches. Section 4.2 illustrates the way to concisely implement some complicated behaviors using MOP via some examples.

The handlers are implemented in the target programming language and will be a part of the generated monitor. Since the monitor is synthesized separately and merged into the program afterwards, handlers may not have direct access to the information of the monitored system. So MOP provides some built-in variables to allow handlers to access context information or to take some special actions, e.g., @this referring to the current object and @reset resetting the state of the monitor to the initial state. These variables will be replaced with appropriate values or pieces of code during monitor synthesis. For example, @this in Figure 1 is renamed to ob in Figure 2.

### 4. MOP at Work

Based on automatic code generation and program instrumentation, MOP provides a powerful support for effectively applying runtime monitoring and recovery in software development to improve the
class Controller {
    float input;
    /*@*/
    scope = class,
    Logic = JML
    {    
      invariant input >= LOWERBOUND && input <= UPPERBOUND;
    }
    Violation Handler :
      if (input < LOWERBOUND) input = LOWERBOUND
      else input = UPPERBOUND;
    /=*/
    ...}
}

Figure 8. Specification to ignore bad sensors

Most safety properties are defined over execution traces and can be formally expressed by trace formalisms, such as temporal logics or regular expressions. Therefore, MOP provides an effective support for enforcing safety properties in software, as illustrated in the security example in Figure 1. What follows gives another example about temporal constraints on class interfaces, showing the usage of validation handlers in MOP.

Correct usage of a class interface sometimes needs to follow certain temporal constraints on the order of method invocations. For example, in Windows applications, one can open a registry key to access the registry by OpenRegistryKey() and she should close the key using CloseRegistryKey() afterward to release all the allocated resources. However, these constraints are not always enforced by the system, meaning that the related violations can be ignored during the execution, although such violations may lead to unexpected problems eventually. For the registry key example, it has been reported that many developers tend to use a generic Windows APT, CloseHandle(), to close the opened key, and this mismatch of operations is not caught by the system because the registry is also a handle ([19]). Unfortunately, the CloseHandle() does not release all allocated resources, introducing a safety leak into the program.

Now let us simplify the scene and fit this problem into Java: we assume that all the operations related to the registry key are implemented in the RegistryKey class. Then we can specify the defective implementation as a regular pattern, as shown by the formula in Figure 9. This specification is interface-scoped since only the invocations for methods are concerned in this property. Two events are defined: openRegistryKey for the exit of the openRegistryKey() method, representing that the key is opened, and closeHandle for the entry of the closeHandle() method, used to capture the violation earlier to allow recovery from the error. We describe the defective behavior instead of the desired property in this example, because it allows us to provide more accurate recovery code and also because the desired property, namely that an invocation of openRegistryKey() should eventually lead to a corresponding invocation of closeRegistryKey(), is a liveness property and cannot be verified until the system ceases ([38]). So the validation of the specified pattern indicates a violation of the desired property. Therefore, a validation handler is used to correct the execution of the system, which simply invokes the closeRegistryKey() method and skips the execution of closeHandle(). This way, the constraint is enforced automatically at runtime, avoiding the safety leak.

4.1 Improving Software Reliability via Recovery

Monitoring has been widely accepted in many engineering disciplines as an effective mechanism to improve the dependability and safety of systems, e.g., fuses in electricity and watchdogs in hardware. We argue that monitoring can also play a key role in software development to obtain highly dependable systems, where MOP provides a fundamental support. In what follows, we demonstrate some applications of MOP that employ runtime monitoring and recovery to build reliable software.

Let us start by a very simple example about survivability of control systems. For many control systems, it is more important to keep the system alive than always getting optimized results. For instance, when the system receives bad sensor signals, it usually tries to ignore the signals and continue in order to avoid potential crashes caused by defective signals. Suppose that the control system is implemented in the Controller class, which uses the field input to receive the sensor signal. Figure 8 then shows an MOP specification to automatically detect and filter out bad signals in the control system. This specification is defined as a class invariant for the Controller class, so it will be checked upon every update of input. JML is used to specify the expected range of the signal. When the property is violated, i.e., the signal is out of range, the violation handler is triggered to adjust the signal into the normal range, ensuring liveness of the control system.

This example may appear to be too simple since the developer has no difficulties in placing the checking and recovery code manually. However, there are still some advantages of using MOP here. First, the updates of input can be scattered into several components in the system, making manual insertion of checking code inefficient and error-prone. On the other hand, MOP provides a fully automated way to monitor the property throughout the system, reducing the programming efforts and improving the modularity of the program. Second, the formal specification of the property supported by MOP is more rigorous and clear than concrete implementations and closer to requirements, facilitating program understanding and software maintenance. This advantage is fortified in the following examples where more complicated properties are needed.

Runtime monitoring is particularly effective for detecting violations of safety properties, e.g., security policies. Violations of such properties usually do not lead to visible errors of the system immediately, making them hard to catch by traditional testing and debugging. Besides, runtime recovery is highly desirable for such violations because they often cause serious damages in the system, such as malicious accesses to resources.

class RegistryKey {
    /*@*/
    scope = interface
    Logic = ERE
    {    
      Event openRegKey: end(exec(* openRegKey()));
      Event closeHandle: begin(exec(* closeHandle()));
      Formula : (openRegKey closeHandle)*
    }
    Validation Handler:
      @this.closeRegKey();
      return;
    /=*/
    ...}
    void openRegKey(){ ... } ;
    void closeRegKey(){ ... } ;
    void closeHandle(){ ... } ;
    ...
}

Figure 9. Specification for safe usage of registry keys
class CarController {
    int currentSpeed;
    int targetSpeed = 0;
    void setCruiseControl() {
        ... targetSpeed = currentSpeed;
    }
    void releaseCruiseControl() {
        ... targetSpeed = 0;
    }
    void doBrake();
}

Figure 10. Car controller class

/*@*
scope = class
Logic = FTLTL
Event setCC : end(exec(* setCruiseControl()));
Event releaseCC : end(exec(* releaseCruiseControl()));
Predicate upperBounded : currentSpeed < (targetSpeed + 5);
Predicate lowerBounded : currentSpeed > (targetSpeed + 5);
Formula : [setCC && (! upperBounded || ! lowerBounded)]
Violation Handler :
   @this.releaseCruiseControl();
   @Reset;
}@*/

Figure 11. Specification for cruise control

Let us consider a more complicated example to further illustrate the specification capability supported by MOP. Cruise control is a common feature of many kinds of cars. It allows the driver to set a cruise speed during driving, and then the car control system will automatically maintain the speed by regulating the gas flow until the driver cancels the cruise mode. However, it is not always good to use cruise control mode. For example, if the car is running on a steep downhill or on wet ground, it may go faster than it should. In such case, the driver needs to retain the control of the car for safety reasons. This represents a rather common safety policy in many automation systems, e.g., automated flight systems: when unexpected situations are detected, the system should return the control to the operator.

There can be many variations of the cruise control function. We here focus on a simplified behavior; a more advanced version will be discussed in Section 4.2. The simplified cruise control behavior only concerns the action of setting and canceling the cruise mode and can be informally described as follows: “Once the cruise control has been set, the speed of the car should not be 5 miles more than or less than the selected cruise speed until the cruise control is released.”

Suppose that the car control system is implemented in the CarController class in Figure 10, which contains the operations for starting/stopping cruise control (setCruiseControl() and releaseCruiseControl), as well as the fields for recording speeds. Then Figure 11 gives the FTLTL-based specification to formally specify the desired behavior of the system. In this class-scoped specification, in addition to the two events that represent the actions of starting and stopping the cruise mode, two predicates are also defined to check the proper range of the car speed. Therefore, the monitoring code will be inserted after the two cruise mode related methods, as well as after every update of currentSpeed and targetSpeed to check the values of the predicates. The formula is written in future time linear temporal logic (FTLTL), where [@] is interpreted as “always”, meaning the property should hold all the time, -> and \ are normal boolean operations for “implies” and “and”, and U is the “until” operator, stating that the left operand should hold until the right operand holds. The MOP framework automatically synthesizes the monitoring code to verify this formula, which can be depicted by the state machine in Figure 12.

When a violation is detected, which means that some unexpected situations are encountered, e.g., the car speed is abnormal or the driver presses the cruise control button again, the control system will try to correct the problem by interrupting the cruise mode and return the control to the driver. The @Reset keyword in the validation handler resets the state of the monitor in order to continue the monitoring process.

Summary. All these examples illustrate that runtime monitoring and recovery can play a key role in developing reliable software, and that MOP can provide fundamental support for applying monitoring in software development. Its extensible formal framework allows the developer to choose appropriate formalisms to specify desired properties in an abstract and modular way, while the violation/validation handlers facilitate implementing accurate runtime recovery.

4.2 Programming using Abstract Aspects

MOP not only supports runtime monitoring and recovery in software development, but it also provides the developer with a means to program using abstract aspects, triggered by sophisticated conditions expressed using logic formalisms. We next show some examples, illustrating the advantages of employing abstract aspects in programming.

Let us start with a typical example of AOP, namely updating the display in graphic application [22], and consider only a simple scenario here, that is, changing positions of points. In order to display the correct content, the display has to be updated whenever a point moves. Suppose that the point is implemented in a class Point, which uses fields x and y to represent its position. With AspectJ, one can implement an aspect that invokes the display update method after every method of Point that may change the position. However, when updating the display is costly, e.g. redrawing a TV wall, it is desirable to make the update only when it is necessary, that is, when the point’s position really changes.

This requirement can also be implemented using aspects, though slightly more tediously, adding appropriate new variables to record the original position and statements to record and compare original and updated positions. However, MOP provides a trivial solution to implement this more efficient strategy of updating the display, as shown in Figure 13. This specification is interface-scoped, stating that it will be executed on the boundaries of every...
class Point{
    int x, y;
    /*@ scope = interface Logic = JML {
     \ensures \old.x == x \&\& \old.y == y 
    } Violation handler:
    Display.update()
    @} ...
}

---

Figure 13. Efficient display updates

class File {
    /*@ static scope = interface Logic = ERE {
     [ int count = 0; ]
     Event open : end(call(* open(..)));
     Event write : end(call(* write(..))) {writes = 0;}
     Event close : end(call(* close(..)));
     Formula : (open write+ close)*
    } Validation handler:
    count ++;
    Logger.log(count + ' ':'' + writes);
   Violation handler:
    @Reset;
    @} ...
}

---

Figure 14. Counting write-only file openings

public method of the class. This scope is chosen for simplicity; one can always associate this specification only to those methods that may change the position of the point. JML is used to specify the condition that may trigger the update of the display. \old is a JML keyword referring to the original state of the object before the execution of the method. So the specified formula essentially checks the original position and the updated position. If they are not equivalent, the display will be updated. Here the MOP specification acts like a complicated branch statement, whose condition may refer to the history state of the object.

In MOP, specification and implementation are tightly coupled together: the implementation is constantly “supervised” and “corrected” by the specification, while the specification is “activated” by events generated by the implementation at various points that can be scattered all over the program. In other words, the specification can be regarded as an abstract aspect of the implementation, that “becomes alive” wherever certain logical properties of interest hold. The display example above considered quite a trivial property, one that only needs to look one step back in order to check its validity. However, MOP can support through its logic plugins much more complex properties, that refer to both past and future behaviors of programs. If used properly, we believe that this capability of MOP can be used as a powerful programming technique.

In other words, with the support of proper formalisms, MOP allows the developer to define trace related behavior in the system, which has proved to be useful in practice ([23]). Let us now consider an example about profiling some specific information during the execution of a program, say to count the number of file openings that are for write operations only. This requirement can be expressed using a regular expression: open write+ close. Hence, we can implement it in an ERE-based MOP specification, as shown in Figure 14. In this specification, a monitor variable count is used as the counter that will be incremented when the pattern is matched. Note that the violation handler is needed to reset the monitor state in order to keep the monitor active.

One advantage of using abstract aspects in MOP is simplicity, both in understanding and in maintaining programs. For example, suppose that more details about the file operations are needed now, such as the number of writes for every file opening. This new profiling function can be obtained by adding one more monitor variable into the specification in Figure 14 to count the writes, as shown in Figure 15.

Let us now re-consider the cruise control example. The simplified cruise control system previously discussed only takes operations on the cruise mode into account, but many other actions may happen under the cruise mode in practice. An important situation is when the driver brakes; in this case, the cruise control should also...
be stopped. This improved function can be implemented easily by MOP, as a slightly changed variant of the specification in Figure 16, shown in Figure 16.

A new event, brake, is added to catch the braking action. And the formula is changed to incorporate the brake event. More importantly, the “always” operator, [], is removed to allow the validation of the formula to happen; in finite trace LTL, an “always” property will not be validated until the system stops. Hence the validation handler not only needs to cancel the cruise mode for the braking action, but also uses a @Reset action to restart the monitor. It also shows that the defined events and predicates can be used in the handlers to indicate the last event causing the violation/validation. In this specification, the formula plays the role of a complex trace-based condition that triggers either the violation handler or the validation handler in order to implement the desired behavior.

Summary. MOP combines specification and implementation by regarding the specification as an abstract aspect of the implementation and triggering “recovery” code when validated or violated. The user is freed to focus on correctly and formally describing the actual requirements of the system rather than decomposing them into hard to check and error-prone implementation details. This way, MOP promotes an abstract “separation of concerns” for software development and also facilitates program understanding and software maintenance.

5. Related Work

We next discuss relationships between our approach and other related paradigms, including design by contract, runtime verification and AOP.

5.1 Design by Contract

Design by Contract (DBC) [35] is a technique allowing one to add semantic specifications to a program in the forms of assertions and invariants, which are then compiled into runtime checks. It was first introduced by Meyer as a built-in feature of the Eiffel language [21]. Some DBC extensions have also been proposed for a number of other languages. Jass [8] 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 [28], and the syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts 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 bytecode 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 bytecode even without the source code.

JML is a behavioral interface specification language for Java. It provides a more comprehensive modeling language than DBC extensions. But not all features of JML can be runtime checked, and its runtime checker supports a DBC-like subset of JML, a large part of which is also supported by JavaMOP.

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 our principles would naturally support DBC as a special methodological case. In addition, our MOP design also supports outline and offline monitoring which we find crucial in assuring software reliability but is not provided by any of the current DBC approaches that we are aware of.

5.2 Runtime Verification

In runtime verification, monitors are automatically synthesized from formal specifications, and can be deployed off-line for debugging, or on-line for dynamically checking properties during execution. MaC [32] and PathExplorer (PaX) [25] are two runtime verification frameworks for logic based monitoring, within which specific tools for Java, Java-MaC and Java PathExplorer, are implemented. Both runtime verification systems work on outline monitoring mode and have hardwired specification languages: MaC uses a specialized language based on interval temporal logic, while PaX supports just LTL. Besides, they integrate the monitors via Java bytecode instrumentation, making them difficult to port to other programming languages. Our approach supports inline, outline and offline monitoring, allows one to define their own formalisms to extend the MOP framework, and is easy to be adapted for new programming languages providing that corresponding AOP tools are available.

Temporal Rover [20] is a commercial runtime verification tool based on future time temporal logic specifications. Similar to MOP, it allows programmers to insert formal specifications in programs via annotations, from which verification code is then generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specification. Temporal Rover and its follower, DB Rover, support both inline and offline monitoring. However, they also have their specification formalisms hard-wired and are tightly bound to Java.

Although our current JavaMOP prototype does not support all these techniques yet, 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.

5.3 Aspect Oriented Programming Languages

Since AOP was introduced in [31], it has been widely accepted and many tools have been developed to support AOP in different programming languages, e.g., AspectJ and JBoss [29] for Java and AspectC++ [2] for C++. Built on these general AOP languages, numerous extensions have been made to provide domain-specific features for AOP. Among these extensions, Tracematches [23] and J-LO [9] support history(trace)-based aspects for Java.

Tracematches enables the programmer to trigger the execution of certain code by specifying a regular pattern of events in a computation trace, where the events are defined over entry/exit of AspectJ pointcuts. When the pattern is matched during the execution, the associated code will be executed. In this sense, Tracematches supports trace-based pointcuts for AspectJ. J-LO is a tool for runtime checking temporal assertions. These temporal assertions are specified using LTL and the syntax adopted in J-LO is similar to Tracematches except that the formulae are written in different logics. J-LO mainly focuses on runtime checking properties rather than providing programming support. In J-LO, the temporal assertions are inserted into Java files as annotations that are then compiled into runtime checking code. Both Tracematches and J-LO support parametric events in trace matching, i.e., free variables can be used in the event patterns and will be bound to specific values at runtime for matching events.

Conceptually, both Tracematches and J-LO can be naturally captured by MOP, because both regular expressions and LTL are supported in MOP. In fact, their regular patterns and temporal assertions can be easily translated into MOP specifications that contain only action events and validation handlers. However, the logic
formalisms currently supported in JavaMOP do not use parametric events; we are in the process of developing a logic plugin for Eagle [17], a logic that includes both LTL and ERE, and also supports parametric events. On the other hand, Tracematches can be regarded an extended AOP framework that supports trace-based aspects; since Tracematches aims at improved performance using the ABC framework [5], it may also be served as a basis to support monitor integration in JavaMOP, especially for those logics that can be translated into regular patterns. It is also worth mentioning that Tracematches and J-LO are implemented using Java bytecode compilation and instrumentation, while MOP acts as an aspect synthesizer, making it easier to port to different programming languages provided that they have AOP tool support.

5.4 Other Related Approaches

There are several other approaches to detecting and/or correcting errors at runtime, which do not fall into any of the three categories above but yet appear to have interesting connection to MOP. We mention two of them. Acceptability-oriented computing [37] aims at enhancing flawed computer systems to respect basic acceptability properties. For example, by augmenting the compiled code with bounds checks to detect and discard out-of-bound memory accesses, the system may execute successfully through attacks that trigger otherwise fatal memory errors. Acceptability-oriented computing is mainly a philosophy and methodology for software development; one has to devise specific solutions to deal with different kinds of failures. We do believe though that MOP can serve as a platform to experiment with and support acceptability-oriented computing, provided that appropriate specification formalisms express “acceptability policy” and appropriate recovery ensures that it is never violated.

Program Query Language (PQL) allows programmers to express design rules that deal with sequences of events associated with a set of related objects [34]. Both static and dynamic tools have been implemented to find solutions to PQL queries. The static analysis conservatively looks for potential matches for queries and are useful to reduce the number of dynamic checks. The dynamic analyzer checks the runtime behavior and can perform user-defined actions when matches are found, similar to MOP handlers. PQL has “hardwired” specification language and supports only inline monitoring. We expect that PQL can be supported via an appropriate logic plugin and that its dynamic tool falls under the general MOP framework.

6. Conclusion

We presented Monitoring-Oriented Programming (MOP), an extendible formal framework for reliable software development based on automatic monitor generation and integration. AOP plays a key role: the MOP tool synthesizes aspects from formal specifications and invokes the AOP compiler to merge the monitors within the original program. This way, the MOP framework can be ported to new program languages provided that AOP support and tools are available.

References


