Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

An Overview of the MOP Runtime Verification Framework*

Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
201 N Goodwin Ave
Urbana, IL, 61801, USA
e-mail: {pmeredit,djin3,dgriffi3,-,grosu}@cs.uiuc.edu

November 5, 2011

Abstract. This article gives an overview of the Monitoring Oriented Programming framework (MOP). In MOP, runtime monitoring is supported and encouraged as a fundamental principle for building reliable systems. Monitors are automatically synthesized from specified properties and are used in conjunction with the original system to check its dynamic behaviors. When a specification is violated or validated at runtime, user-defined actions will be triggered, which can be any code, such as information logging or runtime recovery. Two instances of MOP are presented: JavaMOP (for Java programs) and BusMOP (for monitoring PCI bus traffic). The architecture of MOP is discussed, and an explanation of parametric trace monitoring and its implementation is given. A comprehensive evaluation of JavaMOP attests to its efficiency, especially in comparison with similar systems. The implementation of BusMOP is discussed in detail. In general, BusMOP imposes no runtime overhead on the system it is monitoring.

1 Introduction

Runtime monitoring of requirements can increase the reliability of the resulting hardware or software systems. There is an increasingly broad interest in uses of monitoring in software development and analysis, as reflected, for example, by abundant approaches proposed recently ([4, 12, 16, 19, 25, 26, 29, 31, 43, 46] among others), and also by the runtime verification (RV) and the formal aspects of testing (FATES) initiatives [10, 30, 31, 34, 35, 58] among many others. Hardware approaches to monitoring have seen less active research. Most attempts in hardware to perform monitor tasks have been for the purposes of performance measures or temperature control. [45] is an approach that generates monitors from formal properties that are implemented in hardware, but these hardware monitors are actually used to monitor software programs.

Monitoring oriented programming (MOP) [20–23, 47] is a generic monitoring framework that integrates specification and implementation by checking the former against the latter at runtime. In MOP, one specifies desired properties using logical formalisms with actions to handle violations or validations of the specified property. MOP tools will then automatically synthesize monitors from property specifications and integrate them within the application together with user-provided handling code.

1.1 Related Work

We next discuss relationships between the MOP framework and other related paradigms, including AOP, design by contract, runtime verification, and other trace monitoring approaches. Broadly speaking, all the approaches discussed below are instances of runtime monitoring. Interestingly, even though most of the systems mentioned below target the same programming languages, no two of them share the exact same logical formalism for expressing properties. This observation strengthens our belief that probably there is *no silver bullet logic* (or *super logic*) for all purposes. A major objective in the design of the MOP framework was to avoid hardwiring particular logical formalisms into the system.

1.1.1 Aspect Oriented Programming (AOP) Languages

Since its proposal in [42], AOP has been increasingly adopted and many tools have been developed to support AOP in different programming languages, e.g., AspectJ and JBoss [40] for Java, and AspectC++ [6] for C++. Built on these general AOP languages, numerous extensions have been proposed to provide domain-specific features for AOP. Among these extensions, Tracematches [4] and J-LO [16] support history(trace)-based aspects for Java.

* Supported in part by NSF grants CCF-0916893, CNS-0720512, and CCF-0448501, by NASA contract NNL08AA23C, and by a Samsung SAIT grant.

Tracematches enables the programmer to trigger the execution of certain code by specifying a parametric 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 parametric linear temporal logic (LTL) and the syntax adopted in J-LO is similar to Tracematches' except that the properties are specified in a different formalism. J-LO also uses the same parametricity semantics as Tracematches. J-LO mainly focuses on checking at runtime 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 checks. Both Tracematches and J-LO support parametric events, i.e., free variables can be used in the specified properties and will be bound to specific values at runtime for matching events.

The MOP framework has logic plugins, which encapsulate different logical formalisms and allow it to capture the capabilities of Tracematches and J-LO. JavaMOP is the instantiation of the MOP framework for Java programs (see Section 3.2).

JavaMOP allows for two different modes of matching traces, referred to as total trace matching and suffix trace matching. Total is the default mode of JavaMOP, while suffix mode is used by prefixing a JavaMOP property with the suffix modifier (see Fig. 7 and the accompanying text).

With total matching, for example, with the pattern a^*b, a sequence of events abb will trigger the validation handler of the generated MOP monitor only at the first b event and then the violation handler (if any) at the second b.

With suffix matching, however, the pattern will be matched twice, once for each b event: the first matches either the whole trace a b or the partial trace consisting of just the first b with zero occurrences of a, while the second matches the subsequent partial trace b (the second b in the trace) with zero occurrences of a; thus, the related advice will be executed twice.

With suffix matching one can count matches of a pattern open close without a need to reset the monitor after each match, as would be required with total match monitoring. On the other hand, total trace matching is more suitable for runtime verification of formal properties, because it is the only semantics that makes sense for some logical formalisms, such as LTL, and thus many users expect this behavior for pattern languages like regular expressions and context-free grammars, as well.

J-LO can be captured by the JavaMOP with total matching because LTL (see Section 6.3) is supported by the MOP framework. MOP supports regular expressions as part of its extended regular expression (ERE) logic plugin (see Section 6.2), and Tracematches may be captured by JavaMOP by using these ERE patterns with suffix matching.

1.1.2 Runtime Verification

In runtime verification, monitors are automatically synthesized from formal specifications, and can be deployed *offline* for debugging, or *online* for dynamically checking properties during execution. MaC [43], PathExplorer (PaX) [32], Eagle [11], and RuleR [12] are runtime verification frameworks for logic based monitoring, within which specific tools for Java – Java-MaC, Java PathExplorer, and Hawk [25], respectively – are implemented. All these runtime verification systems work in outline monitoring mode and have hardwired specification languages: MaC uses a specialized language based on interval temporal logic, JPaX supports just LTL, and Eagle adopts a fixed-point logic. Java-MaC and Java PathExplorer integrate monitors via Java bytecode instrumentation, making them difficult to port to other languages. Our MOP approach supports inline, outline, and offline monitoring; allows one to define new formalisms to extend the MOP framework; and is adaptable to new languages (we discuss two such instances in this paper).

Temporal Rover [26] is a commercial runtime verification tool based on future time metric temporal logic. It allows programmers to insert formal specifications in programs via annotations, from which monitors are generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specifications. Temporal Rover and its successor, DB Rover, support both inline and offline monitoring. However, they also have their specification formalisms hardwired and are tightly bound to Java. MOP currently has no metric temporal logic plugin.

1.1.3 Design by Contract

Design by Contract (DBC) [49] is a technique allowing one to add semantic specifications to a program in the form of assertions and invariants, which are then compiled into runtime checks. It was first introduced as a built-in feature of the Eiffel language [28]. Some DBC extensions have also been proposed for a number of other languages. Jass [13] and jContractor [2] 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 CSP [38], 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 class or interface. 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.

Java modeling language (JML) [44] is a behavioral interface specification language for Java. It provides a more comprehensive modeling language than DBC extensions. Not all features of JML can be checked at runtime; its runtime

checker supports a DBC-like subset of JML. Spec# [9] is a DBC-like extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions and also provides method contracts in the form of pre- and post-conditions as well as object invariants. Using the Spec# compiler, one can statically enforce non-null types, emit run-time checks for method contracts and invariants, and record the contracts as metadata for consumption by downstream tools.

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 monitoring DBC specifications as a special methodological case. In addition, the MOP framework also supports outline monitoring, which we find important in assuring software reliability (e.g., monitoring for and detecting and fixing deadlocks) but which is not provided by any of the current DBC approaches that we are aware of.

1.1.4 Other Related Approaches

Program Query Language (PQL) allows programmers to express design rules that deal with sequences of events associated with a set of related objects [46]. 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 is 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. PQL has a "hardwired" specification language based on context-free grammars (CFG) and supports only inline monitoring. CFGs can potentially express more complex languages than regular expressions, so in principle PQL can express more complex safety policies than Tracematches. The MOP CFG plugin described in Section 6.5 allows the MOP framework to specify most of the properties that may be specified in PQL.

Program Trace Query Language (PTQL) [29] is a language based on SQL-like relational queries over program traces. The current PTQL compiler, Partiqle, instruments Java programs to execute the relational queries on the fly. PTQL events are timestamped and the timestamps can be explicitly used in queries. PTQL queries can be arbitrarily complex and, as shown in [29], PTQL's runtime overhead seems acceptable in many cases but we were unable to obtain a working package of PTQL and compare it in our experiments with JavaMOP because of license issues. PTQL properties are globally scoped and their running mode is inline. PTQL provides no support for recovery, its main use being to detect errors.

The PSL to Verilog compiler, P2V [45], is the sole attempt to perform runtime monitoring of *formal properties* in hardware, other than our BusMOP instance (see Sections 3.3 and 5), of which we are aware. P2V is similar to BusMOP in that monitors are implemented in hardware rather than software, and that both approaches thus have no runtime overhead on the CPU. P2V, however, is more like the above approaches in that it is designed for monitoring actual programs rather than peripheral devices. Also it requires a dynamically extensible soft-core processor implemented on an FPGA, while our approach can potentially be applied to any COTS communication architecture. Further, P2V uses hardwired logic (PSL) while BusMOP allows different formalisms.

1.1.5 Discussion

All this research and associated tools show that runtime monitoring is an increasingly accepted, powerful, and beneficial approach for developing reliable software and hardware. Here we summarize the systems discussed above, and show how they may be classified in terms of the five orthogonal attributes of the MOP framework: programming language, logic, scope, running mode, and handlers. The programming language determines what language the programs to be monitored must be written in. The logic specifies which formalism is used to specify the property. The scope determines where to check the property; it can be class invariant, global, interface, etc. The running mode denotes where the monitoring code runs; it can be inline (weaved into the code), online (operating at the same time as the program), outline (receiving events from the program remotely, e.g., over a socket), or offline (checking logged event traces)[1]. The handlers specify what actions to perform under exceptional conditions; there can be violation and validation handlers. It is worth noting that for many logics, violation and validation are not complementary to each other, i.e., the violation of a formula does not always imply the validation of the negation of the formula.

Most runtime monitoring approaches can be framed in terms of these attributes, while in the MOP framework they may be configured. Fig. 1 lists the attributes for most of the software monitoring systems discussed above. For example, JPaX can be regarded as an approach that uses linear temporal logic (LTL) to specify class-scoped properties, whose monitors work in offline mode and only detect violation.

This observation essentially motivates the design discipline of the MOP framework and specification language, namely that one should be allowed to choose the most appropriate logic and the most efficient monitoring algorithm for her/his own applications: while programming languages are designed and intended to be universal, logics and specifications tend to work best when they are domain-specific.

1.2 Examples

Fig. 2 shows an example specification using JavaMOP; recall that this is the MOP instance for Java programs (see Sections 3.2 and 4). Detailed explanation of the specification syntax can be found in Sections 3.1 and 3.2.1. This specification, called SafeEnum, describes the correct behavior of using Enumerations in Java. Essentially, this specification requires that an Enumeration created from a Vector not be used if the Vector has been updated since the Enumeration was created. This is important in legacy code that still uses Vectors and

[1] Offline implies outline, and inline implies online.

Approach	Language	Logic	Scope	Mode	Handler
Hawk [25]	Java	Eagle	global	inline	violation
J-Lo [16]	Java	ParamLTL	global	inline	violation
Jass [13]	Java	assertions	global	inline	violation
JavaMaC [43]	Java	PastLTL	class	outline	violation
jContractor [2]	Java	contracts	global	inline	violation
JML [44]	Java	contracts	global	inline	violation
JPaX [32]	Java	LTL	class	offline	violation
P2V [45]	C, C++	PSL	global	inline	validation/ violation
PQL [46]	Java	PQL	global	inline	validation
PTQL [29]	Java	SQL	global	outline	validation
Spec# [9]	C#	contracts	global	inline/ offline	violation
RuleR [12]	Java	RuleR	global	inline	violation
Temporal Rover [26]	C, C++, Java, Verilog, VHDL	MiTL	class	inline	violation
Tracematches [8]	Java	Reg. Exp.	global	inline	validation

Fig. 1. Runtime Monitoring Breakdown.

```
full−binding connected decentralized SafeEnum(Vector v, Enumeration e) {
 Vector instanceV;
 Enumeration instanceE;
 event createE after(Vector v) returning(Enumeration e) :
        call(* Vector.elements()) && target(v)
   {instanceE = e; instanceV = v;}
 event updateV after(Vector v) :
        (call(* Vector.add*(..)) || call(* Vector.remove(..))) && target(v)
   {instanceV = v;}
 event useE after(Enumeration e) :
        call(* Enumeration.nextElement()) && target(e)
   {instanceE = e;}
 fsm :
  start [
   updateV −> start
   createE −> enumCreated
  ]
  enumCreated [
   useE −> enumCreated
   updateV −> invalidEnum
  ]
  invalidEnum [
   updateV −> invalidEnum
  ]
  @fail {
   System.out.println("Enumeration " + __MONITOR.instanceE
           + " created from Vector " + __MONITOR.instanceV
           + " not used properly at " + __LOC);
  }
}
```

Fig. 2. A JavaMOP Specification (SafeEnum)

Enumerations because Java does not warn of this practice, it simply allows for non-deterministic results.

The specification is composed of five parts. The first line is the header of the specification, starting with three modifiers, full-binding, connected, and decentralized; the first states that monitor instances for this property should only raise failures when every parameter for the monitor instance has been bound (Section 4.4), the second states that the objects bound to the parameters must be connected by an event that actually occurs (Section 4.4), and the last chooses the way to index monitors for different parameter bindings (Section 4.3). An ID for the specification is given after modifiers and followed by parameters of the property; in this example, two parameters are used, namely a Vector object v and an Enumeration object e.

The second part contains the declaration of two monitor variables: instanceV and instanceE. Each monitor instance for each instantiation of the specification parameters has distinct monitor instance variables. Thus, they can be used for many purposes: logging, extra states for monitoring, statistics, and so on. Here, they are used for bug reporting, to keep track of which Vector and Enumeration cause the failure.

The third part of the specification contains event declarations. Three events are defined: createE for the creation of an Enumeration, updateV for updates to a Vector, and useE for uses of an Enumeration. JavaMOP borrows (and extends; see Section 3.2) the syntax of AspectJ [41] for event declarations. For example, the createE event is declared to occur "after" a function call to the elements() method of class Vector. Note that the target clause is used to bind parameters in the event. Each event also sets one or both of the monitor variables, which will, again, be distinct for each binding of the parameters, using an *event action* (the Java code within the curly braces).

The fourth part of the specification is a formal description of the desired property. As discussed in Section 2, MOP is specification formalism independent, and one may choose different logics to specify properties. In this example, the property description begins with fsm, meaning that a finite state machine (FSM) is used, and continues with a finite state description of the monitor. Monitors for FSM properties are initially in the first state listed in the specification, in this case start. The monitor stays in the start state until an Enumeration is created from a given Vector. Once the Enumeration has been created, it is safe to use the Enumeration until such time as the underlying Vector is modified, at which point the invalidEnum state is entered. Using an Enumeration in the invalidEnum state will result in a failure of the property.

The last part of the specification consists of handlers to execute in different states of the corresponding monitor, such as pattern match or failure. In Fig. 2, the handler starts with @fail, defining the action, a simple warning in this case, to execute when the trace fails to match the pattern. The handler reports which Vector and Enumeration are used incorrectly, and the line number where the failure occurs (given by the MOP-reserved variable __LOC). The __MONITOR keyword is resolved to the monitor object by JavaMOP. This is needed because there is no way from the context to tell if a given variable reference refers to a variable declared locally or a monitor instance variable.

JavaMOP specifications are compiled into AspectJ [41] aspects. Specifications as short as the one in Fig. 2 compile into several hundred lines of AspectJ code. The generated aspect can then be weaved into a program one wishes to monitor, using any AspectJ compiler. Once weaved, simply running the program as normal results in a monitored run of the program.

Fig. 3 shows an example specification using BusMOP, the MOP instance for PCI Bus monitoring (see Sections 3 and 5). The main use for this instance is ensuring the proper use of peripherals connected to the PCI Bus. Improper use of

```
pci SafeCounterModify{
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  event countDisable : memory write address = base1 + X"220"
                       dbyte value(0) in '0'
  event cntrlMod : memory write address in base1 + X"220"
     {cntrlOld <= cntrlCurrent; cntrlCurrent <= value(15 downto 0); }
  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'
  ere : ((countEnable countDisable) | cntrlMod | countDisable)*
  @fail {
    mem_reg <= '1';
    address_reg <= base1 + X"220";
    value_reg(15 downto 0) <= cntrlOld;
    cntrlCurrent <= cntrlOld;
    enable_reg <= "0011";
  }
}
```

Fig. 3. A BusMOP Specification (SafeCounterModify)

peripherals may result from bugs in drivers or from misuse of the drivers by application programs. This specification, SafeCounterModify, states a desired property of the PCI703A digital-to-analog and analog-to-digital converter PCI board (ADC) [27]. The ADC has counters that are used to determine when input data is fully converted and ready to be placed on the PCI bus. The specification in Fig. 3 is concerned with the ADC's Counter 2. It requires that any modification to cntr_cntrl2, the control register on the ADC for Counter 2, happens only while the Counter 2 is not enabled (running). Counter 2 is enabled when the 0'th bit of cntr_cntrl2 is set to '1'.

As in Fig. 2, the first line is the header of the specification. The keyword pci specifies that this property should generate bus listening code for the PCI bus. Again an ID naming the specification is provided. This time, because BusMOP does not have parameters, there is no parameter list.

The second part of the specification declares two signals, cntrlCurrent and cntrlOld, much like the monitor variables of Fig. 2, but BusMOP has no monitor instances, so there is only one copy of the variables. These variables are used to store the previous value of cntr_cntrl2, which is the control register for Counter 2 on the ADC board. This is necessary because PCI bus properties cannot prevent incorrect behavior, but only detect and correct it. The stored value is used to restore the value of the register when the pattern fails to match (see below).

The third part of the specification contains event declarations, much like those in Fig. 2, but using an instrumentation language specific to PCI Bus traffic, rather than AspectJ. Three events are defined. The keyword dbyte used in each event tells BusMOP that the quantity will be 16 bits wide (i.e., double byte). Event countDisable occurs when cntr_cntrl2, which is address X"220" in the address space of the ADC (base1 contains the address of the beginning of the ADC's address space), has its 0th bit (value(0)) set to '0', which disables Counter 2. The third event, countEnable, is analogous, but, as mentioned earlier, the bit is set to '1'. The event cntrlMod occurs when cntr_cntrl2 is modified. The keyword in is used rather than = to define the address for cntrlMod. This is because when no value for the read or write is specified, it is possible to check a whole range of addresses. Note that this event overlaps with countDisable and countEnable. The order of the events in Fig. 3 is significant because simultaneous events are handled by reporting them in the declared order (see Section 5). Each cntrlEnable saves the previous value of the register, so that it may be restored if the property is violated. The special variable value refers to the value of the data on the bus. A pipeline is kept where the previous value is stored to cntrlOld before cntrlCurrent receives the new bus value, so that the previous value may be recovered if the pattern fails (the event action occurs before the pattern is checked).

As in Fig. 2, the fourth part is a formal description of the desired property, this time using an extended regular expression (ERE). This pattern specifies the desired behavior where all modifications must happen *after* disabling the counter (note again the order of event declarations, which ensures that the cntrlMod encountered from a countDisable is reported *after* the cntrlMod). The pair (countEnable countDisable) enforces that no changes can be made to cntrl_cntrl2 while Counter 2 is enabled, other than disabling it.

The last part of the specification is the handler for a pattern failure, similar to SafeEnum. An assignment of '1' to the special variable mem_reg alerts the system that a memory write is eminent. The address of the write is placed in address_reg (note that it is the control register for Counter 2). The special variable value_reg is the value to be written out by the monitor, and it is given the value of cntrlOld, which stores the previous value of cntr_cntrl2. Lastly, the enable_reg is specific to the PCI Bus interface (see Section 5).

BusMOP specifications are compiled into hardware description language (HDL). As in JavaMOP, the size of the generated code is far greater than that of the original specification. The HDL code is compiled into an FPGA bitstream and programmed onto an FPGA that is inserted into an empty slot on the PCI bus of the system one wishes to monitor.

The examples given in Figs. 2 and 3 may monitor completely different properties in completely different problem domains, but they follow the same pattern and philosophy. By a clear separation of monitor generation and monitor integration, MOP provides fundamental and generic support for effective and efficient application of runtime monitoring in different problem domains, and can be understood from at least three perspectives:

1. As a discipline allowing one to improve safety, reliability and dependability of a system by monitoring its requirements against its implementation at runtime;
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 are 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, etc.;
3. As a lightweight formal method. While firmly based on logical formalisms and mathematical techniques, MOP's

purpose is not program verification. Instead, the idea is to avoid verifying an implementation against its specification before operation, by not letting it go wrong at runtime.

Section 2 introduces the generic MOP framework. Section 3 discusses the two current language instances of MOP, giving a brief overview and describing their syntax. Section 4 presents topics specific to the efficient implementation of JavaMOP, as well as a thorough evaluation of JavaMOP, while Section 5 focuses on BusMOP. A performance evaluation of BusMOP (the MOP instance for monitoring PCI bus traffic) is unnecessary, as it has zero runtime overhead.[2]

2 MOP framework

All monitoring systems share some features, such as program instrumentation and monitor integration, even when they aim at different domains or goals. MOP separates monitor generation and integration and provides a generic, extensible framework for runtime monitoring, allowing one to instantiate MOP with specific programming languages and specification formalisms to support different domains. In this section, we focus on the overall architecture of MOP.

2.1 Architecture

Fig. 4 shows the architecture of MOP. There are two kinds of high level components in MOP, namely the logic repository and language clients. The logic repository, shown in the bottom of Fig. 4, contains various logic plugins and a logic plugin manager component. The logic plugin is the core component to generate monitoring code from formulae written in a specific logic; for example, the Linear Temporal Logic (LTL) plugin synthesizes state machines from LTL formulae. The output of logic plugins is usually pseudocode and not bound to any specific programming language. This way, the essential monitoring generation can be shared by different instances of MOP using different programming languages. The logic plugin manager bridges the communication between the language clients and the logic plugin. More specifically, it receives the monitor generation request from the language client and distributes the request to an appropriate plugin. After the plugin synthesizes the monitor for the request, the logic plugin manager collects the result and sends it back to the language client. This way, one can easily add new logic plugins into the repository to support new specification formalisms in MOP without changing the language client.

2.2 Language Client

The language client hides the programming language-independent logic repository and provides language specific support for the different MOP instances. Because the language client is the language specific portion of an MOP instance, we occasionally refer to the language client by the name of the MOP instance to which it belongs. Language clients are responsible for all language-specific aspects of monitoring, such as instrumentation, parametricity, online/inline/outline, modifiers, etc. They are usually composed of three layers: the bottom layer contains language translators that translate the abstract output of logic plugins into concrete code in a specific programming language; the middle layer is the specification processor, which extracts formulae from the given property specification and then instruments the generated monitoring code into the target program; finally, the top layer provides usage interfaces to the user.

We next explain in some detail the Java language client for the JavaMOP instance (which by abuse of terminology we will simply call JavaMOP). JavaMOP generates AspectJ [41] aspects from a specification. At the bottom layer, it has language translators for context-free grammars (CFGs), the pseudocode output generated by the past time linear temporal logic with calls and returns plugin, and finite state machine descriptions. All plugins not mentioned use finite state machine descriptions as an output language. At the mid level, as mentioned, the Java client instruments the program with the generated monitor code by creating a stand-alone aspect that can be weaved into the program using any AspectJ compiler, such as ajc [7]. At the top level there is a command line interface and a web-based interface. The two current MOP instances are discussed in Section 3, and, respectively, Section 4 (JavaMOP) and Section 5 (BusMOP), and the discussions essentially apply to the language clients associated with each instance.

```
//Declaration of monitor state
int state = 0;
static final int transition_createE[] = {2, 3, 3, 3};
static final int transition_updateV[] = {0, 1, 1, 3};
static final int transition_useE[] = {3, 3, 2, 3};
//Code for state update
state = transition_createE[state];
state = transition_updateV[state];
state = transition_useE[state];
//Code for category checks
Category_fail = state == 3;
```

Fig. 5. Java code for the FSM in Fig. 2

2.3 Logic Plugins

Every logic plugin implements and encapsulates a monitor synthesis algorithm for a particular specification formalism, such as the past-time linear temporal logic (PTLTL) and the CFG plugins supported in the current MOP framework (see Section 6 for a complete list of available plugins). The logic plugin accepts, as input, a set of events and a formula or pattern written in the underlying formalism and outputs an abstract monitor. This abstract monitor is usually a piece of pseudocode, which checks a trace of events against the given formula.

We next explain in some detail one particular plugin, the plugin for FSM specifications. Fig. 5 shows the monitoring code generated by the MOP FSM plugin from the FSM specification in Fig. 2.

[2] Overhead is exactly 0% if no recovery actions are performed. Recovery actions take up a tiny portion of the bus bandwidth, and could theoretically add non-zero runtime overhead. This was negligible in practice, even with continuously recovering properties.

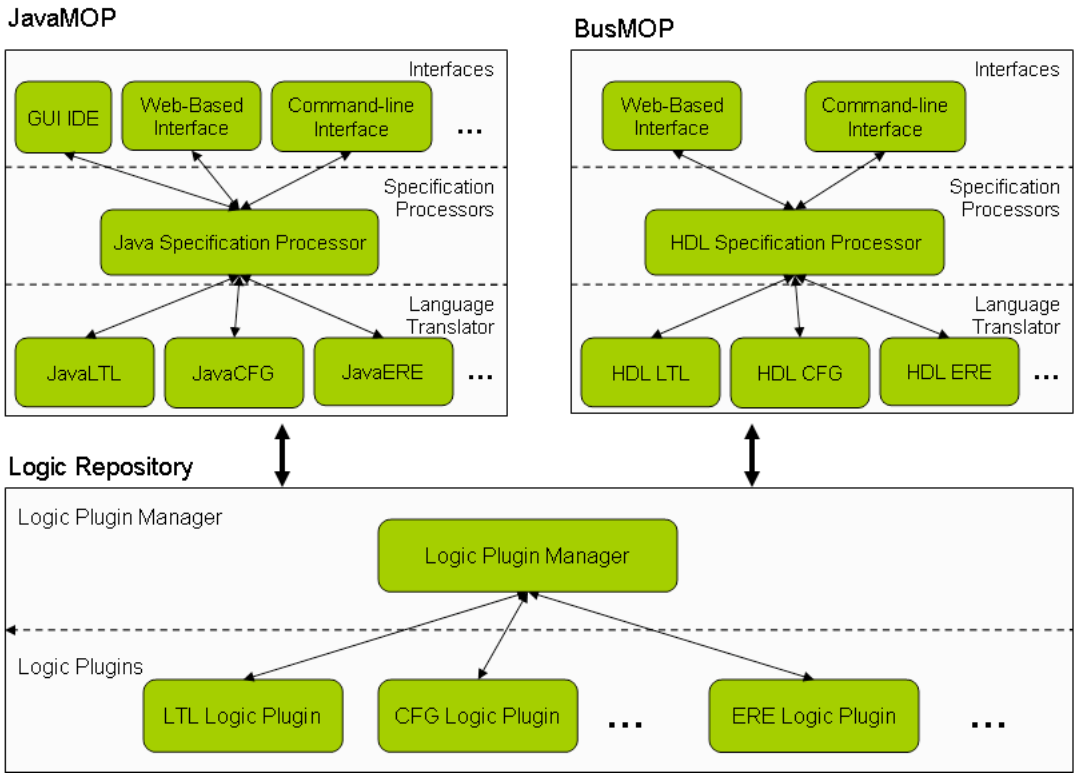


Fig. 4. Architecture of MOP

FSM monitors are simple, as one might expect. Static arrays keep the next state. There is one array for each event in the specification, as can be seen in Fig. 5. When an event arrives, the proper array is queried with the current state, and the next state is returned. After the state is updated, the category checks are preformed to see which handlers must run. Because the specification only checked @fail, we only have one check, which is for fail. As can be seen, fail is reached if the machine is in state 3. This code must be combined with generic code to handle the other properties of the specification, such as connectedness or full-binding, as well as the indexing system used for parametric trace slicing. The FSM plugin, as well as the others, is described in Section 6.

3 MOP Instances

As one may expect, when putting together various languages and specification formalisms, each with its own syntax and semantics, consistency and syntactic separation may become a non-trivial problem. In this section we discuss the four dimensions that need to be instantiated in order to develop a new MOP instance (like JavaMOP or BusMOP), how they are instantiated, and where the boundary between the various components of an instance is. Since the semantics of the various pieces is typically implicit and not formally defined, in what follows we place emphasis on syntax.

3.1 MOP Syntax

Every MOP instance needs to instantiate the MOP framework in four dimensions: 1) a specification language based on the problem domain, which is mainly related to how one defines events in the domain; 2) a target language for generated monitors; 3) supported logic plugin specification formalisms; and 4) the handlers allowed in the specification. Two MOP instances have been implemented and experimented with at this point: JavaMOP and BusMOP. We expect to see more MOP instances in the future because many problem domains can benefit from monitoring.

Each instance of MOP uses an instance of the generic MOP syntax. The syntax of any instance of MOP can be generated by defining certain syntactic categories (non-terminals) of the MOP grammar, which can be seen in Fig. 6. All of the grammars used to define MOP syntax in this article use Extended Backus-Naur Form (EBNF) [1]. Non-terminals in the grammars are surrounded by "⟨" and "⟩". Braces ("{" and "}") enclose portions of the grammar that may appear zero or more times. Brackets ("[" and "]") enclose portions of the grammar that are optional (i.e., it may or may not appear). Concrete examples of the syntax defined below can be seen in Figs. 2 and 3.

3.1.1 Shared syntax

The following syntax constructs are shared by different MOP instances:

Shared syntax

⟨*Specification*⟩ ::= {⟨*Instance Modifier*⟩} ⟨*Id*⟩ ⟨*Instance Parameters*⟩ "{"
{⟨*Instance Declaration*⟩}
{⟨*Event*⟩}
{⟨*Property*⟩
{⟨*Property Handler*⟩}
}
"}"
⟨*Event*⟩ ::= ["*creation*"]"*event*" ⟨*Id*⟩ ⟨*Instance Event Definition*⟩ "{" ⟨*Instance Action*⟩ "}"
⟨*Property*⟩ ::= ⟨*Logic Name*⟩ " : " ⟨*Logic Syntax*⟩
⟨*Property Handler*⟩ ::= "@" ⟨*Logic State*⟩ ⟨*Instance Handler*⟩

Instance-specific syntax

⟨*Instance Modifier*⟩ ::= ⟨*Id*⟩
⟨*Instance Parameters*⟩ ::= ⟨*JavaMOP Parameters*⟩ | ⟨*BusMOP Parameters*⟩ | ...
⟨*Instance Declaration*⟩ ::= ⟨*JavaMOP Declaration*⟩ | ⟨*BusMOP Declaration*⟩ | ...
⟨*Instance Event Definition*⟩ ::= ⟨*JavaMOP Event Definition*⟩ | ⟨*BusMOP Event Definition*⟩ | ...
⟨*Instance Action*⟩ ::= ⟨*JavaMOP Event Action*⟩ | ⟨*BusMOP Event Action*⟩ | ...
⟨*Instance Handler*⟩ ::= ⟨*JavaMOP Event Handler*⟩ | ⟨*BusMOP Event Handler*⟩ | ...

Logic-plugin-specific syntax

⟨*Logic Name*⟩ ::= ⟨*Id*⟩
⟨*Logic Syntax*⟩ ::= ⟨*FSM Syntax*⟩ | ⟨*ERE Syntax*⟩ | ⟨*LTL Syntax*⟩ | ⟨*PTLTL Syntax*⟩ | ⟨*CFG Syntax*⟩ | ⟨*PTCaRet Syntax*⟩ | ...
⟨*Logic State*⟩ ::= ⟨*FSM State*⟩ | ⟨*ERE State*⟩ | ⟨*LTL State*⟩ | ⟨*PTLTL State*⟩ | ⟨*CFG State*⟩ | ⟨*PTCaRet State*⟩ | ...

Fig. 6. MOP Syntax

- ⟨*Specification*⟩ — ⟨*Specification*⟩ describes the generic MOP specification syntax which can be instantiated for MOP language instances and MOP logic plugins.
- ⟨*Event*⟩ — The ⟨*Event*⟩ declaration code allows for the definition of events, which may then be referred to in the property (see ⟨*Property*⟩ below). Event declarations can also have arbitrary code associated with them (⟨*Instance Action*⟩), which is run when the event is observed (⟨*Instance Event Definition*⟩), e.g. code to modify the program or the monitor state. For manual indication of events that can start a trace, the keyword creation is used at the beginning of each declaration.[3]
- ⟨*Property*⟩ — Every MOP specification may contain zero or more properties. A ⟨*Property*⟩ consists of a named formalism (⟨*Logic Name*⟩), followed by a colon, followed by a property specification using the named formalism (see ⟨*Logic Syntax*⟩ below) and usually referring to the declared events. If the property is missing, then the MOP specification is called *raw*. Raw specifications are useful when no existing logic plugin is powerful or efficient enough to specify the desired property; in that case, one embeds the custom monitoring code manually within the ⟨*Instance Action*⟩ code.
- ⟨*Property Handler*⟩ — Handlers contain arbitrary code from the instance source language, and are invoked when a certain logic state (see ⟨*Logic State*⟩ below) or category is reached, e.g., match, fail, or a particular state in a finite state machine description.

[3] The creation keyword has no effect in BusMOP specifications.

3.1.2 Instance-specific syntax

The following constructs are based on the particular instance of MOP used for a particular specification. More information on the instances of MOP can be found in the remainder of this section, and Sections 4 (JavaMOP) and 5 (BusMOP).

- ⟨*Instance Modifier*⟩ — ⟨*Instance Modifier*⟩s are specific to each language instance of MOP. Syntactically, they can be any valid identifier restricted by the given language. They change the behavior of the monitoring code.
- ⟨*Instance Parameters*⟩ — allow one to define the parameters of a parametric specification using the language corresponding to the MOP instance. Not all MOP instances are parametric (e.g., BusMOP), however, so this non-terminal may be empty.
- ⟨*Instance Declaration*⟩ — ⟨*Instance Declaration*⟩s are specific to each language instance of MOP. They allow for the declaration of monitor local variables.
- ⟨*Instance Event Definition*⟩ — ⟨*Instance Event Definition*⟩s are specific to each language instance of MOP. They define the conditions under which an event is triggered.
- ⟨*Instance Action*⟩ — An event can have arbitrary code associated with it, called an action. The action is run when the event is observed. An action can modify the program or the monitor state, and the syntax of the allowed statements are dependent upon the MOP instance in question. Typically the statements used in actions have different variables and functions that may be referred to than handlers.

This is why different non-terminals are used for actions and handlers.
- ⟨*Instance Handler*⟩ — ⟨*Instance Handler*⟩s are arbitrary code that is executed when a property handler is triggered.

3.1.3 Logic-plugin-specific syntax

The following constructs are based on the logic plugin(s) used in a particular specification. More information on logic plugins can be found in Section 6.

- ⟨*Logic Name*⟩ — An identifier to indicate in which logic a property is defined.
- ⟨*Logic Syntax*⟩ — This refers to the syntax of the actual property definition, and is defined in the syntax section for each plugin.
- ⟨*Logic State*⟩ — ⟨*Logic State*⟩s are constants defined for each plugin, stating for which monitor states or categories (match, fail, etc.) a handler may be written.

3.2 The JavaMOP Instance

JavaMOP is an MOP development tool for Java, supporting several logical formalisms and a general specification language using them to describe Java program behaviors [22]. It compiles property specifications into optimized monitoring code. The generated code uses AspectJ [41], and is currently[4] program-independent. For example, a user can write a JavaMOP specification for a library. Then, JavaMOP generates monitoring code for this specification. This code can be applied to any program that uses the library.

In JavaMOP, an event corresponds to a pointcut, which an AspectJ compiler (such as ajc [7]) can use to weave monitoring code into the original program. Pointcuts include function call, function return, function begin, function end, field assignment, object creation, and more complex ones with pointcut operators, which combine multiple simpler pointcuts. JavaMOP generates monitoring code for each pointcut—corresponding to an event in a JavaMOP specification—to maintain monitoring state, to check if the program conforms to the specification, and to trigger a handler if appropriate.

A system behavior can be described using one of several logical formalisms supported by JavaMOP, including all those described in Section 6. A specification will be interpreted by the logic repository, a generic server used by all instances of MOP, and transformed into generic monitor code as mentioned in Section 2. JavaMOP translates the monitor pseudocode to AspectJ code. Any logic which can be translated to finite state machines (ERE, LTL, PTLTL) are reported to JavaMOP using the MOP finite state machine plugin syntax to reduce the number of translation algorithms necessary in JavaMOP (see Section 6.1).

A user can write a handler in Java for each monitoring state. There can be more monitoring states than simple match and fail, depending on logical formalism. A handler can be used for logging, recovering, blocking, or any other purpose. Since handlers are specified as arbitrary Java code, a user has quite a bit of latitude to achieve his or her purposes.

[4] We intend to incorporate program static analysis to further reduce runtime overhead soon [17].

3.2.1 JavaMOP Syntax

The syntax of JavaMOP is discussed below, as an instance of the generic MOP syntax defining the relevant modifiers and language-specific syntax (Java for declarations and event/handler actions, enriched with AspectJ for event definitions). The formal syntax can be seen in Fig. 7. Anything not explicitly described below can be considered to be identical to the generic MOP syntax. Note that some non-terminals such as ⟨*Event*⟩ refer to language instance specific non-terminals, which are defined below for JavaMOP.

- ⟨*JavaMOP Modifier*⟩ — The three binding modifiers refer to the different binding modes described in Section 4.4, the default is any-binding. The modifier *"unsynchronized"* tells JavaMOP that the monitor state needs not be protected against concurrent accesses; the default is synchronized. The unsynchronized monitor is faster, but may suffer from races on its state updates if the monitored program has multiple threads. The *"decentralized"* modifier refers to decentralized monitor indexing. The default indexing is centralized, meaning that the indexing trees needed to quickly access and garbage-collect monitor instances are stored in a common place; decentralized indexing means that the indexing trees are scattered all over the code as additional fields of objects of interest. Decentralized indexing typically yields lower runtime overhead, though it may not always work for all settings. More information on indexing can be found in Section 4.3. The *"perthread"* modifier causes JavaMOP to consider events from each thread as though from separate runs of the program, (i.e., one parametric monitor for each thread monitors only events from its own thread). The *"suffix"* modifier causes JavaMOP to consider a trace as matching if any suffix of that trace would match.
- ⟨*JavaMOP Parameters*⟩ and ⟨*JavaMOP Declaration*⟩ — These are ordinary Java parameters (as used in methods) and Java declarations. The former are the parameters of the JavaMOP specification and the latter are additional local monitor variables that one can access and modify in both event actions and property handlers. Each parameter from ⟨*JavaMOP Parameters*⟩ should be used in at least one event of the specification.
- ⟨*JavaMOP Action*⟩, ⟨*JavaMOP Handler*⟩, and ⟨*JavaMOP Event Definition*⟩ — ⟨*JavaMOP Action*⟩ are normal Java statements that may also refer to monitor local variables. ⟨*JavaMOP Handler*⟩, however, slightly extends Java with three special variables:
  - __RESET — a special expression (evaluates to void) that resets the monitor to its initial state, but does not affect any user defined varibles of the monitor;
  - __LOC — a string variable that evaluates to the line number generating the current event;

⟨*JavaMOP Modifier*⟩ ::= "*full−binding*" | "*maximal−binding*" | "*any−binding*" | "*connected*" | "*unsynchronized*"
| "*decentralized*" | "*perthread*" | "*suffix*"
⟨*JavaMOP Parameters*⟩ ::= "(" [{⟨*JavaMOP Type*⟩ ⟨*Id*⟩ ","} ⟨*JavaMOP Type*⟩ ⟨*Id*⟩}] ")"
⟨*JavaMOP Declaration*⟩ ::= syntax of declarations in Java
⟨*JavaMOP Event Definition*⟩ ::= ⟨*AspectJ AdviceSpec*⟩ " : "
⟨*AspectJ Pointcut*⟩ ["&&" ⟨*JavaMOP Pointcut*⟩]
⟨*JavaMOP Pointcut*⟩ ::= "*thread*" "(" ⟨*Id*⟩ ")"
| "*condition*" "(" ⟨*BooleanExpression*⟩ ")"
| ⟨*AspectJ Pointcut*⟩
| ⟨*JavaMOP Pointcut*⟩ "&&" ⟨*JavaMOP Pointcut*⟩
⟨*AspectJ Pointcut*⟩ ::= syntax of Pointcut in AspectJ
⟨*AspectJ AdviceSpec*⟩ ::= syntax of AdviceSpec in AspectJ
⟨*TypeList*⟩ ::= list of Exception types in Java
⟨*Boolean Expression*⟩ ::= ⟨*Id*⟩ | "!" ⟨*Boolean Expression*⟩
| ⟨*Boolean Expression*⟩ ⟨*Boolean Operator*⟩ ⟨*Boolean Expression*⟩ | "(" ⟨*Boolean Expression*⟩ ")"
⟨*Boolean Operator*⟩ ::= " || " | "&&" | " | " | "&" | " == " | "! = "
⟨*JavaMOP TypeList*⟩ ::= "(" [{⟨*JavaMOP Type*⟩ ","} ⟨*JavaMOP Type*⟩] ")"
⟨*JavaMOP Action*⟩ := Java statements, which may refer to monitor local variables
⟨*JavaMOP Handler*⟩ := Java statements with additional keywords
⟨*JavaMOP Type*⟩ := Any valid Java type

Fig. 7. JavaMOP Syntax

– __MONITOR — a special variable that evaluates to the current monitor object, so that one can read/write monitor variables.

Similarly, the advice used to define JavaMOP events slightly extends the AspectJ advice syntax. The ⟨*JavaMOP Event Definition*⟩ follows the AspectJ syntax except for its extension with ⟨*JavaMOP Pointcut*⟩, which can only be added in a top-level conjunct context. ⟨*AspectJ Pointcut*⟩ and ⟨*AspectJ AdviceSpec*⟩ are both standard AspectJ syntax [7]. The additional pointcuts have the following meaning:

– *"thread"* — The thread pointcut captures the current thread and takes an identifier as a parameter. The identifier can be a class name or a variable name. For the former, the type of the captured thread should be a sub-class of the given class to trigger the event. For the latter, the captured thread is bound to the variable. The thread pointcut allows for the easy specification of properties which are parameterized by the current thread of execution.
– *"condition"* — The condition pointcut takes a boolean expression as a parameter. An event containing a condition pointcut is not triggered if the boolean expression evaluates to false. This differs only from the if pointcut in standard AspectJ in that monitor instance variables may be used in the conditional expression.

3.3 The BusMOP Instance

BusMOP [52] was designed to address the safety problem of third party consumer off-the-shelf (COTS) components. The complexity of safety critical systems has grown to the point where the ability to use COTS in a safe manner is almost mandatory. Additionally, the vast majority of OS crashes in PCs are caused by faulty peripherals or their drivers. BusMOP answers both of these problems by allowing the specification and monitoring of properties with respect to PCI Bus traffic (soon to be expanded to other bus architectures).

In BusMOP, the events correspond to reads and writes of specified values to specified memory locations on the bus. PCI Bus interrupts are also allowed as events. The monitors, and the logic to extract events from bus traffic, are synthesized from hardware design language (HDL) code and programmed onto a field programmable gate array (FPGA), which is plugged into the PCI Bus.

BusMOP supports the FSM, ERE, LTL, and PTLTL plugins of MOP (see Section 6). PTCaRet and CFG have the problem of unbounded logic response time, which would cause the monitor to not meet timing constraints in some cases, and are thus not suitable for inclusion in BusMOP. It is also not clear exactly where the structured capabilities of these logics are useful when considering flat bus traffic traces.

Handlers in BusMOP can be specified using arbitrary VHDL code. Several resources are provided for the user for use in handler code, such as serial output for logging, and the actual ability to write to the PCI Bus to perform recovery. Recovery actions in BusMOP require bus arbitration to undo deleterious actions of faulty peripherals or their drivers. This bus arbitration is the only possible overhead incurred by BusMOP, in cases of heavy Bus traffic. In the majority of systems, BusMOP can be used with no runtime overhead.

3.3.1 BusMOP Syntax

Below we discuss the BusMOP syntax. Anything not explicitly described below can be considered to be identical to the generic MOP syntax. Note that some non-terminals such as ⟨*Event*⟩ refer to language instance specific non-terminals, which are defined below for BusMOP. The grammar for the syntax can be seen in Fig. 8.

⟨*BusMOP Modifier*⟩ ::= "*pci*"
⟨*BusMOP Parameters*⟩ ::= ϵ (i.e., none)
⟨*BusMOP Declaration*⟩ ::= syntax of declarations in VHDL
⟨*BusMOP Event Definition*⟩ ::= " : " ⟨*Memory or IO*⟩ ⟨*Read or Write*⟩ "*address*" " = " ⟨*Arithmetic Op*⟩
"*value*"["(" ⟨*Index*⟩ ")"]["*not*"]"*in*" ⟨*Range*⟩ "{"[⟨*BusMOP Action*⟩]"}"
| " : " ⟨*Memory or IO*⟩ ⟨*Read or Write*⟩ "*address*" "*in*" ⟨*Range*⟩
| "*interrupt*" "{"[⟨*BusMOP Action*⟩]"}"
⟨*Memory or IO*⟩ ::= "*memory*"|"*io*"
⟨*Read or Write*⟩ ::= "*read*"|"*write*"
⟨*Range*⟩ ::= ⟨*Arithmetic Op*⟩ [", " ⟨*Arithmetic Op*⟩]
⟨*Arithmetic Op*⟩ ::= ⟨*Number*⟩ | ⟨*ID*⟩
| ⟨*Arithmetic Op*⟩ " + " ⟨*Arithmetic Op*⟩
| ⟨*Arithmetic Op*⟩ "&" ⟨*Arithmetic Op*⟩
| ⟨*Arithmetic Op*⟩ " − " ⟨*Arithmetic Op*⟩
| "(" ⟨*Arithmetic Op*⟩ ")"
⟨*Number*⟩ ::= ⟨*VHDL number or bitstring*⟩
⟨*ID*⟩ ::= ⟨*VHDL identifier*⟩
⟨*BusMOP Action*⟩ := Bus statements, which may refer to monitor local variables
⟨*BusMOP Handler*⟩ := Bus statements, which may refer to monitor local variables, with additional output variables

Fig. 8. BusMOP Syntax

- ⟨*BusMOP Modifier*⟩ — Modifiers in BusMOP are used to distinguish the bus architecture to be monitored. Currently, only standard parallel PCI is supported. PCI-E will be supported by using a bridge adapter that will sit between the PCI Express bus slot and the peripheral to monitor.
- ⟨*BusMOP Parameters*⟩ and ⟨*BusMOP Declaration*⟩ — BusMOP is not parametric because there is no clear unit of parametrization. ⟨*BusMOP Declaration*⟩'s are standard VHDL signal declarations. These are used to define additional local monitor variables that one can access and modify in both event actions and property handlers.
- ⟨*BusMOP Event Definition*⟩ — BusMOP event definitions use an original syntax to define interesting potential bus traffic. At the basic level there are three types of events: memory, IO, and interrupt. The last event is triggered when there is an interrupt on the bus. The first two are further subdivided into reads and writes. The difference between memory and IO is the address space of the read or write in question. This is important for correctly specifying the necessary bus enable signals in the generated code. Reads and writes can be concerned with the read or write of a specific location with a select range of values, or a read or write to a range of locations where the value is of no concern. Specifying a range of read or write addresses is valuable for enforcing memory safety policies (such as, if the value 0xdeadbeef is written to address 0xffff0000 then allow no writes to some buffer until 0x000000 is written to 0xffff0000). ⟨*Arithmetic Op*⟩ allows for arithmetic operations combining variables and literal numbers. This is useful both for specifying monitor local variables and monitor input variables. Placing a numerical index on the keyword "*value*" indicates that one bit, specified by the index, should be checked rather than the whole value read or written. The monitor input variables hold the values of inputs to the monitor, and are as follows:
  - The value register holds the value of the read or write in question.
  - The address register holds the address of the read or write in question.
  - The baseN registers allow a user to specify a memory value relative to a given peripheral. Without this support monitoring would be very difficult due to the plug-and-play PCI bus interface that assigns memory spaces to peripherals at boot time.
- ⟨*BusMOP Action*⟩ — Actions are arbitrary VHDL statements that may refer to monitor local variables as well as the input variables described above in ⟨*BusMOP Event Definition*⟩. As mentioned in ⟨*Instance Action*⟩, these statements are executed when the event for which they are defined is observed.
- ⟨*BusMOP Handler*⟩ — Handlers are arbitrary VHDL statements that may refer to monitor local variables as well as the input variables described above in ⟨*BusMOP Event Definition*⟩. Additionally, there are variables the may be set in order to perform recovery actions. They are as follows:
  - The io_reg is used to specify a read or write to I/O space. It is asserted as '1' to select the I/O space.
  - The mem_reg is used to specify a read or write to memory space by asserting it as '1'.
  - The address_reg is used to specify the 32 bit address of a read or write.
  - The value_reg is used to specify the value of a 32 bit read or write.
  - The enable_reg is used to specify the byte enables for a read or write (specific to PCI, see Section 5).
  - The serial_reg allows output of an ASCII value to a serial port for debugging.
  - The stop_reg register that stops the peripheral in question from reading from or writing to the PCI bus when it is asserted as '1'.

As mentioned in ⟨*Instance Handler*⟩ these statements are executed when a property handler is triggered.

4 JavaMOP

Each instance of MOP has issues specific to its domain. JavaMOP must deal with the complexities of parametric monitoring, in order to make itself useful in highly object-oriented systems. We first provide an introduction to parametric trace slicing (Section 4.1). We next cover improving the efficiency of parametric monitoring4.2 Lastly, we discuss different modes of parameter binding, which define which parameter instance monitors trigger handlers (Section 4.4).

4.1 Parametric Trace Slicing and Naive Monitoring

Parametric specifications are widely used in practice, particularly in object oriented languages, like Java, where we need to describe properties over a group of objects. For example, consider again the property in Fig. 2 from Section 1. Here the events are parametrized by the Vector v and the Enumeration e. This is because we do not want uses of an Enumeration e_1 to be flagged as an error because of an intervening modification to Vector v_2, when it has Vector v_1 as its underlying Vector.

When monitoring a parametric specification, the observed execution trace is parametric, i.e., the events in the trace come with parameter information. For example, a possible parametric trace for the specification in Fig. 2 is:
updateV$\langle v \mapsto v_1 \rangle$ createE$\langle v \mapsto v_1, e \mapsto e_1 \rangle$ createE$\langle v \mapsto v_1, e \mapsto e_2 \rangle$ createE$\langle v \mapsto v_2, e \mapsto e_3 \rangle$ useE$\langle e \mapsto e_3 \rangle$ useE$\langle e \mapsto e_1 \rangle$ updateV$\langle v \mapsto v_1 \rangle$ useE$\langle e \mapsto e_1 \rangle$ useE$\langle e \mapsto e_2 \rangle$.

Every event in this trace is associated with a concrete parameter binding, such as $\langle v \mapsto v_1, e \mapsto e_2 \rangle$ that indicates that the parameters v and e in Fig. 2 are bound to concrete objects v_1 and e_2, respectively. Such a parametric trace represents a set of non-parametric traces each of which corresponds to a particular parameter binding. For example, the above trace contains eleven non-parametric traces for eleven parameter bindings (one for each of five singleton objects, and one for each element in the cross product of the singleton objects). The non-parametric trace for four of these bindings are summarized in the table below.

Parameter binding	Non-parametric trace slice
$\langle v \mapsto v_1 \rangle$	updateV updateV
$\langle v \mapsto v_1, e \mapsto e_1 \rangle$	updateV createE useE updateV useE
$\langle v \mapsto v_1, e \mapsto e_2 \rangle$	updateV createE updateV useE
$\langle v \mapsto v_2, e \mapsto e_3 \rangle$	createE useE

The second and third of these fail to match the pattern in Fig. 2, thus two failures are produced. It is highly non-trivial to monitor parametric specifications efficiently since there can be a tremendous number of parameter bindings during a single execution. For example, in a few experiments that we carried out, millions of parameter bindings were created [23]. Most other approaches for monitoring parametric specifications handle parameters in a logic-specific way [4, 16, 46], that is, they extended the underlying specification formalisms with parameters and devised algorithms for the extended formalism. Such a solution results in very complicated monitor synthesis algorithms and makes it difficult to support new problem domains. In MOP, parameters are handled in a completely logical formalism independent manner and separated from the monitor synthesis process, vastly simplifying the implementation of new logic plugins. Surprisingly, this logic independent consideration of parameters turns out to be more efficient than those closely coupled systems (see Section 4.5) thanks to the clean separation of concerns. In the following sections we will explain parametric monitoring in more detail.

4.1.1 Events, Traces, Properties, and Parameters

First, we introduce the notions of event, trace and property, first non-parametric and then parametric. Trace slicing is then defined as a reduct operation that forgets the events that are unrelated to the given parameter instance. Most of this discussion is derived from [24].

Definition 1. *Let $\mathcal{E}$ be a set of (non-parametric) events, called **base events** or simply **events**. An **$\mathcal{E}$-trace**, or simply a (non-parametric) **trace** when $\mathcal{E}$ is understood or not important, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.*

Example. Consider again the SafeEnum policy from Fig. 2. $\mathcal{E} = \{$createE, updateV, useE$\}$ and execution traces corresponding to this are sequences of the form createE useE updateV createE useE, etc. For now we ignore the distinction between "good" and "bad" execution traces. □

Definition 2. *An **$\mathcal{E}$-property** P, or simply a (base or non-parametric) **property**, is a function $P : \mathcal{E}^* \to \mathcal{C}$ partitioning the set of traces into categories $\mathcal{C}$. It is common, but not enforced, that $\mathcal{C}$ includes "*match*", "*fail*", and "*don't know*" (or "?") categories. In general, $\mathcal{C}$, may be any set and is referred to as the set of* verdict categories *when it eases readability.*

Example. Consider again Fig. 2. The FSM has no match category as we did not define it. The fail category is reached by "falling off the machine", that is, receiving an event in a state for which there is no transition. For example, the trace createE updateV useE would result in the fail category. □

We next extend the above definitions to the parametric case, i.e., traces containing events that carry concrete data instantiating abstract parameters.

Example. Event useE is parametric in the Enumeration; if e is the name of the generic Enumeration parameter and e_1 and e_2 are concrete Enumerations, then parametric useE events have the form useE$\langle e \mapsto e_1 \rangle$, useE$\langle e \mapsto e_2 \rangle$, etc. □

In what follows, let $[A \to B]$ be the set of total functions and $[A \rightharpoonup B]$ be the set of partial functions, both from A to B.

Definition 3. ***(Parametric events and traces).** Let X be a set of **parameters** and let V be a set of corresponding **parameter values**. If $\mathcal{E}$ is a set of base events like in Def. 1, then let $\mathcal{E}\langle X \rangle$*

*be the set of corresponding **parametric events** $e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is a partial function in $[X \rightarrow V]$. A **parametric trace** is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.*

To simplify writing, we occasionally assume the parameter values set V implicit.

Example. A parametric trace for our property in Fig. 2 can be:

$\mathsf{updateV}\langle v \mapsto v_1\rangle$ $\mathsf{createE}\langle v \mapsto v_1, e \mapsto e_1\rangle$ $\mathsf{createE}\langle v \mapsto v_1, e \mapsto e_2\rangle$ $\mathsf{createE}\langle v \mapsto v_2, e \mapsto e_3\rangle$ $\mathsf{useE}\langle e \mapsto e_3\rangle$ $\mathsf{useE}\langle e \mapsto e_1\rangle$ $\mathsf{updateV}\langle v \mapsto v_1\rangle$ $\mathsf{useE}\langle e \mapsto e_1\rangle$ $\mathsf{useE}\langle e \mapsto e_2\rangle$.

We take the freedom to only list the parameter values when writing parameter instances, that is, $\langle v_1\rangle$ instead of $\langle v \mapsto v_1\rangle$. With this notation, the above trace is:

$\mathsf{updateV}\langle v_1\rangle$ $\mathsf{createE}\langle v_1, e_1\rangle$ $\mathsf{createE}\langle v_1, e_2\rangle$ $\mathsf{createE}\langle v_2, e_3\rangle$ $\mathsf{useE}\langle e_3\rangle$ $\mathsf{useE}\langle e_1\rangle$ $\mathsf{updateV}\langle v_1\rangle$ $\mathsf{useE}\langle e_1\rangle$ $\mathsf{useE}\langle e_2\rangle$.

As mentioned earlier, this trace induces *eleven trace slices*. The slice corresponding to $\langle v_1, e_1\rangle$ is

updateV createE useE updateV useE □

Definition 4. *Partial functions θ in $[X \rightarrow V]$ are called **parameter instances**. $\theta, \theta' \in [A \rightarrow B]$ are **compatible** if for any $x \in \mathsf{Dom}(\theta) \cap \mathsf{Dom}(\theta')$, $\theta(x) = \theta'(x)$. We can **combine** compatible instances θ and θ', written $\theta \sqcup \theta'$, as follows:*

$$(\theta \sqcup \theta')(x) = \begin{cases} \theta(x) & \text{when } \theta(x) \text{ is defined} \\ \theta'(x) & \text{when } \theta'(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

*$\theta \sqcup \theta'$ is also called the **least upper bound (lub)** of θ and θ'. θ is **less informative** than θ', or θ' is **more informative** than θ, written $\theta \sqsubseteq \theta'$, if for any $x \in X$, if $\theta(x)$ is defined then $\theta'(x)$ is also defined and $\theta(x) = \theta'(x)$.*

Definition 5. ***(Trace slicing)** Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and θ in $[X \rightarrow V]$, let the **θ-trace slice** $\tau{\restriction_\theta} \in \mathcal{E}^*$ be the non-parametric trace defined as:*

- *$\epsilon{\restriction_\theta} = \epsilon$, where ϵ is the empty trace/word, and*
- *$(\tau\, e\langle\theta'\rangle){\restriction_\theta} = \begin{cases} (\tau{\restriction_\theta})\, e & \text{when } \theta' \sqsubseteq \theta \\ \tau{\restriction_\theta} & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$*

The trace slice $\tau{\restriction_\theta}$ first filters out all the parametric events that are not relevant for the instance θ, i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ, it forgets the parameters so that the trace can be checked against base, non-parametric properties. It is crucial to discard events for parameter instances that are not relevant to θ during the slicing, including those more informative than θ, in order to achieve a "proper" slice for θ: in our running example, the trace slice for $\langle v_1\rangle$ should contain only updateV events and no createE or useE events.

Definition 6. *Let X be a set of parameters together with their corresponding parameter values V, like in Def. 3, and let $P : \mathcal{E}^* \rightarrow \mathcal{C}$ be a non-parametric property like in Def. 2. Then we define the **parametric property** $\Lambda X.P$ as the property (over traces $\mathcal{E}\langle X\rangle^*$ and categories $[[X \rightarrow V] \rightarrow \mathcal{C}]$)*

$$\Lambda X.P : \mathcal{E}\langle X\rangle^* \rightarrow [[X \rightarrow V] \rightarrow \mathcal{C}]$$

defined as $(\Lambda X.P)(\tau)(\theta) = P(\tau{\restriction_\theta})$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and any $\theta \in [X \rightarrow V]$. If $X = \{x_1, ..., x_n\}$ we may write $\Lambda x_1, ..., x_n.P$ instead of $(\Lambda\{x_1, ..., x_n\}.)P$. Also, if P_φ is defined using a pattern or formula φ in some particular trace specification formalism, we take the liberty to write $\Lambda X.\varphi$ instead of $\Lambda X.P_\varphi$.

A parametric property is therefore similar to a normal property, except that the domain is parametric traces, and the output, rather than being one category, is a mapping of parameter instances to categories. This allows the parametric property to associate an output category for each parameter instance from $[X \rightarrow V]$.

4.1.2 Monitors and Parametric Monitors

Here we define monitors M and parametric monitors $\Lambda X.M$. Like for parametric properties, which are just properties over parametric traces, parametric monitors are also just monitors, but for parametric events and with instance-indexed states and output categories: a parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$, with P the property monitored by M [24].

Monitors are defined as a variant of Moore machines:

Definition 7. *A **monitor** M is a tuple $(S, \mathcal{E}, \mathcal{C}, 1, \sigma : S \times \mathcal{E} \rightarrow S, \gamma : S \rightarrow \mathcal{C})$, where S is the set of states, $\mathcal{E}$ is the set of input events, $\mathcal{C}$ is the set of output categories, $1 \in S$ is the initial state, σ is the transition function, and γ is the output function. The transition function is extended to handle traces of events (i.e., $\sigma : S \times \mathcal{E}^* \rightarrow S$) the standard way.*

The notion of a monitor above is often impractical. Actual implementations of monitors need not generate all the state space *a priori*, but rather on an "as needed" basis. Allowing monitors with infinitely many states is a necessity in our context. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on how many may be reached. For example, monitors for context-free grammars have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well.

Definition 8. *$M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \rightarrow \mathcal{C}$ if $\gamma(\sigma(1, w)) = P(w)$ for each $w \in \mathcal{E}^*$. Every monitor M defines the property $\mathcal{P}_M$: $\mathcal{E}^* \rightarrow \mathcal{C}$ with $\mathcal{P}_M(w) = \gamma(\sigma(1, w))$; We write $\mathcal{P}_M$ to denote the property defined by M. Monitors M and M' are **equivalent**, written $M \equiv M'$ if $\mathcal{P}_M = \mathcal{P}_{M'}$.*

We next define parametric monitors in the same style as the other parametric entities defined in this paper: starting with a base monitor and a set of parameters, the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

Definition 9. *Given parameters X with corresponding values V and monitor $M = (S, \mathcal{E}, \mathcal{C}, 1, \sigma : S \times \mathcal{E} \rightarrow S, \gamma : S \rightarrow$*

$\mathcal{C}$), *the* ***parametric monitor*** $\Lambda X.M$ *is the monitor* $([[X \rightharpoonup V] \to S], \mathcal{E}\langle X\rangle, [[X \rightharpoonup V] \to \mathcal{C}], \lambda\theta.\imath, \Lambda X.\sigma, \Lambda X.\gamma)$, *with* $\Lambda X.\sigma : [[X \rightharpoonup V] \to S] \times \mathcal{E}\langle X\rangle \to [[X \rightharpoonup V] \to S]$ *and* $\Lambda X.\gamma : [[X \rightharpoonup V] \to S] \to [[X \rightharpoonup V] \to \mathcal{C}]$ *defined as*

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases}$$
$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

for any $\delta \in [[X \rightharpoonup V] \to S]$ *and any* $\theta, \theta' \in [X \rightharpoonup V]$.

Therefore, a state δ of parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ, takes parametric events as input, and outputs categories indexed by parameter instances (one category of M per instance). Intuitively, one can think of a parametric monitor as a *collection* of *"monitor instances"*. Each monitor instance, which is indexed by a parameter instance, keeps track of the state of one trace slice. The rule for $\Lambda X.\sigma$ can be read as stating that when an event with parameter instance θ' is evaluated, it updates the state for all monitor instances more informative than the instance for θ', and the instance for θ' itself, leaving all other monitor instances untouched. The rule for $\Lambda X.\gamma$ simply states that γ is applied to a state, as normal, but the state is found by looking up the state of the monitor instance for θ.

4.1.3 Naive Parametric Monitoring

Intuitively, the necessary steps for online monitoring of parametric properties are as follows:

1. Begin with a monitor instance for the empty parameter instance $\perp$ initialized to the start state of the monitor, $\imath$.
2. As each event, $e\langle\theta\rangle$, arrives there are two possibilities:
   - There is already a monitor instance for θ, in this case the instance is simply updated with e.
   - There is not already a monitor instance for θ, in this case an instance is created for θ. It is initialized to the state of the *most informative* θ' less informative than θ. Such a θ' is guaranteed to exist because we begin with a monitor instance for $\perp$, which is less informative than all other possible θ's. We also create monitor instances for every parameter instance that may be created by combining θ with previously seen parameter instances. Each of these created instances is initialized similarly to the instance for θ, using the most informative instance less than itself. All created monitor instances are updated with e after initialization.
3. e is then used to update the monitor instances for all θ' that are strictly more informative than θ.

We next present a more concrete monitoring algorithm for parametric properties first introduced in [24]. It is derived from the algorithm $\mathbb{A}\langle X\rangle$, which is omitted here, that was also first presented in [24]. A first challenge here is how to represent the states of the parametric monitor. We encode the functions $[[X \rightharpoonup V] \rightharpoonup S]$ as tables with entries indexed by parameter instances in $[X \rightharpoonup V]$ and with contents states in S. Such tables will have finite entries since each event binds only a finite number of parameters. Fig. 9 shows our monitoring algorithm for parametric properties. Given parametric property $\Lambda X.P$ and M a monitor for P, $\mathbb{B}\langle X\rangle(M)$ yields a monitor that is equivalent to $\Lambda X.M$, that is, a monitor for $\Lambda X.P$.

Algorithm $\mathbb{B}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma))$

Globals : mapping $\Delta : [[X \rightharpoonup V] \rightharpoonup S]$

function main(τ)
1 $\Delta(\perp) \leftarrow \imath$; $\Theta \leftarrow \{\perp\}$
2 foreach $e\langle\theta\rangle$ *in order in* τ do
3 : foreach $\theta' \in \{\theta\} \sqcup \Theta$ do
4 : : $\Delta(\theta') \leftarrow \sigma(\Delta(\max(\theta']_\Theta), e)$
5 : : $\Gamma(\theta') \leftarrow \gamma(\Delta(\theta'))$
6 : endfor
7 : $\Theta \leftarrow \{\perp, \theta\} \sqcup \Theta$
8 endfor

Fig. 9. Naive Monitoring Algorithm $\mathbb{B}\langle X\rangle$

$\mathbb{B}\langle X\rangle$ first assigns $\imath$, the initial state, to $\Delta(\perp)$ (Δ is a mapping from parameter instance to monitor state). Θ, which contains all known parameter instances is initialized to contain $\{\perp\}$, as $\perp$ is always known. For each event $e\langle\theta\rangle$ that arrives during program execution (line 2), $\mathbb{B}\langle X\rangle$ generates every compatible parameter instance by combining $\{\theta\}$ with all the previously known parameter instances. It then updates the state of every one of these compatible parameter instances (θ') on line 4 with the state, transitioned by event e, of the "monitor instance" corresponding to the "largest" parameter instance less than or equal to θ'. At the same time we also calculate the output corresponding to that monitor instance and store it in table Γ. Rather than storing a whole slice as in Def. 5, the knowledge of the slice is encoded in the state of the monitor instance for θ'. After the algorithm completes Γ contains the category for each possible trace slice. An actual implementation is free to report a category (e.g., match) as soon as it is discovered. In fact, in JavaMOP, it is necessary to report a category as soon as it occurs so that recovery actions can be performed, and also because the category of a trace may change several times throughout its lifetime, while $\mathbb{B}\langle X\rangle$ only gives the final result.

4.2 Efficient Parametric Monitoring

Algorithm $\mathbb{B}\langle X\rangle$ is correct, and easy to understand, but it is not very efficient. It creates many more monitor instances than are actually required to correctly monitor a given property. An algorithm designed for runtime monitoring should receive a trace one event at a time, rather than all at once as $\mathbb{B}\langle X\rangle$. Next we show an algorithm that receives a trace one event as a time. We also discuss optimizations to the algorithm that *vastly* improve efficiency [21].

Algorithm $\mathbb{C}\langle X\rangle(M = (S, \mathcal{E}, \mathcal{C}, \imath, \sigma, \gamma))$

Globals : mapping $\Delta : [[X \rightarrow V] \rightarrow S]$
mapping $\mathbb{U} : [X \rightarrow V] \rightarrow \mathcal{P}_f([X \rightarrow V])$

Initialization : $\Delta \leftarrow \langle\rangle, \mathbb{U} \leftarrow \langle\rangle$
$\mathbb{U}(\theta) \leftarrow \emptyset$ for any $\theta \in [X \rightarrow V]$

function main($e\langle\theta\rangle$)
1 if $\Delta(\theta)$ undefined then
2 : foreach $\theta_m \sqsubset \theta$ (in reversed topological order) do
3 : : if $\Delta(\theta_m)$ defined then goto 5 endif
4 : endfor
5 : if $\Delta(\theta_m)$ defined then defineTo(θ, θ_m)
6 : elseif e is a creation event then defineNew(θ)
7 : endif
8 : foreach $\theta_m \sqsubset \theta$ (in reversed topological order) do
9 : : foreach $\theta_{comp} \in \mathbb{U}(\theta_m)$ compatible with θ do
10 : : : if $\Delta(\theta_{comp} \sqcup \theta)$ undefined then
11 : : : : defineTo($\theta_{comp} \sqcup \theta, \theta_{comp}$)
12 : : : endif
13 : : endfor
14 : endfor
15 endif
16 foreach $\theta' \in \{\theta\} \cup \mathbb{U}(\theta)$ do $\Delta(\theta') \leftarrow \sigma(\Delta(\theta'), e)$ endfor

function defineNew(θ)
1 $\Delta(\theta) \leftarrow \imath$
2 foreach $\theta'' \sqsubset \theta$ do $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$ endfor

function defineTo(θ, θ')
1 $\Delta(\theta) \leftarrow \Delta(\theta')$
2 foreach $\theta'' \sqsubset \theta$ do $\mathbb{U}(\theta'') \leftarrow \mathbb{U}(\theta'') \cup \{\theta\}$ endfor

Fig. 10. Monitoring Algorithm $\mathbb{C}\langle X\rangle$.

4.2.1 Algorithm $\mathbb{C}\langle X\rangle$

Fig. 10 shows the algorithm $\mathbb{C}\langle X\rangle$[5] for online monitoring of parametric property $\Lambda X.P$, given that M is a monitor for P. Note that we assume the notation of $\langle\rangle$ for empty maps throughout the remainder. The algorithm shows which actions to perform, e.g., creating a new monitor state and/or updating the state of related monitors, when an event is received. Algorithm $\mathbb{C}\langle X\rangle$ refines Algorithm $\mathbb{B}\langle X\rangle$ in Fig. 9 for efficient online monitoring. $\mathbb{C}\langle X\rangle$ essentially expands the body of the outer loop in $\mathbb{B}\langle X\rangle$ (lines 3 to 7 in Fig. 9). The direct use of $\mathbb{B}\langle X\rangle$ would yield prohibitive runtime overhead when monitoring large traces, because its inner loop requires searching for all parameter instances in Θ that are compatible with θ; this search can be very expensive. $\mathbb{C}\langle X\rangle$ introduces an auxiliary data structure and illustrates a mechanical way to accomplish the search, which also facilitates further optimizations.

Algorithm $\mathbb{C}\langle X\rangle$ also extends algorithm $\mathbb{B}\langle X\rangle$ to support *creation events*. Recall from Section 3.1 that users may specifically choose creation events using the keyword creation. Supporting *creation events* in algorithm $\mathbb{C}\langle X\rangle$ is justified and

[5] This algorithm was referred to as $\mathbb{C}^+\langle X\rangle$ in [21]. The distinctions between $\mathbb{C}\langle X\rangle$ and $\mathbb{C}^+\langle X\rangle$ are small, and elided for conciseness.

$\Lambda m, c, i$. createC$\langle m, c\rangle$ updateM$\langle m\rangle^*$createI$\langle c, i\rangle$
usel$\langle i\rangle^*$ updateM$\langle m\rangle^+$ usel$\langle i\rangle$

Fig. 11. Parametric Property (UnsafeMapIterator)

motivated by experience with implementing and evaluating $\mathbb{B}\langle X\rangle$ in [24], mainly by the following observation: one often chooses to start monitoring at the witness of a specific set of events (versus the beginning of the program).

Two mappings are used in $\mathbb{C}\langle X\rangle$: Δ and $\mathbb{U}$. Δ stores the monitor states for parameter instances, and $\mathbb{U}$ maps a parameter instance θ to *all the parameter instances* that have been defined and are properly more informative than θ. In what follows, "to create a parameter instance θ" and "to create a monitor state for parameter instance θ" have the same meaning: to define $\Delta(\theta)$.

We next use an example about the interaction between the classes Map, Collection and Iterator in Java (Fig. 11), as this provides a better demonstration of the power of $\mathbb{C}\langle X\rangle$ than does the pattern in Fig. 2. This also provides us an opportunity to show how parametric trace slicing is truly generic with respect to the logical formalism by using extended regular expressions (ERE) in place of finite state machines (FSM). Map and Collection implement data structures for mappings and collections, respectively. Iterator is an interface used to enumerate elements in a collection-typed object. One can also enumerate elements in a Map object using Iterator. But, since a Map object contains key-value pairs, one needs to first obtain a collection object that represents the contents of the map, e.g., the set of keys or the set of values stored in the map, and then create an iterator from the obtained collection. An intricate safety property in this usage, according to the Java API specification, is that when the iterator is used to enumerate elements in the map, the contents of the map should not be changed, or unexpected behaviors may occur. The parametric LTL formula in Fig. 11 specifies the incorrect behavior of the system.

In Fig. 11, createC is an event corresponding to creating a collection from a map, createI corresponds to creating an iterator from a collection, updateM corresponds to updating the map, and useI corresponds to using the iterator. The pattern says that a Collection is created from a Map, an Iterator is created from the Collection, the Map is updated at least once ($^+$ means one or more times), and then the Iterator is used after the update. The extra updateM$\langle m\rangle^*$ and useI$\langle c, i\rangle^*$ define places where these events are still valid. When an observed execution matches this pattern, the UnsafeMapIterator property is broken.

In Fig. 12, we show the contents of Δ and $\mathbb{U}$ after every event (given in the first row of the table) is processed. The observed trace is updateM$\langle m_1\rangle$ createC$\langle m_1, c_1\rangle$ createC $\langle m_2, c_2\rangle$ createI$\langle c_1, i_1\rangle$. We assume that createC is the only creation event. The first event, updateM $\langle m_1\rangle$, is not a creation event and nothing is added to Δ and $\mathbb{U}$. The second event, createC$\langle m_1, c_1\rangle$, is a creation event. So a new monitor state is defined in Δ for $\langle m_1, c_1\rangle$, which is also added to the lists in $\mathbb{U}$ for $\perp$, $\langle m_1\rangle$ and $\langle c_1\rangle$. Note that $\perp$ is less informative than any

	$\mathsf{updateM}\langle m_1\rangle$	$\mathsf{createC}\langle m_1, c_1\rangle$	$\mathsf{createC}\langle m_2, c_2\rangle$	$\mathsf{createI}\langle c_1, i_1\rangle$
Δ	$\emptyset$	$\langle m_1, c_1\rangle : \sigma(1, \mathsf{createC})$	$\langle m_1, c_1\rangle : \sigma(1, \mathsf{createC})$ $\langle m_2, c_2\rangle : \sigma(1, \mathsf{createC})$	$\langle m_1, c_1\rangle : \sigma(1, \mathsf{createC})$ $\langle m_2, c_2\rangle : \sigma(1, \mathsf{createC})$ $\langle m_1, c_1, i_1\rangle : \sigma(\sigma(1, \mathsf{createC}), \mathsf{createI})$
$\mathbb{U}$	$\emptyset$	$\langle\rangle : \langle m_1, c_1\rangle$ $\langle m_1\rangle : \langle m_1, c_1\rangle$ $\langle c_1\rangle : \langle m_1, c_1\rangle$	$\langle\rangle : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle$ $\langle m_1\rangle : \langle m_1, c_1\rangle$ $\langle c_1\rangle : \langle m_1, c_1\rangle$ $\langle m_2\rangle : \langle m_2, c_2\rangle$ $\langle c_2\rangle : \langle m_2, c_2\rangle$	$\langle\rangle : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_1\rangle : \langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle c_1\rangle : \langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_2\rangle : \langle m_2, c_2\rangle$ $\langle c_2\rangle : \langle m_2, c_2\rangle$ $\langle i_1\rangle : \langle m_1, c_1, i_1\rangle, \langle m_2, c_2, i_1\rangle$ $\langle m_1, c_1\rangle : \langle m_1, c_1, i_1\rangle$ $\langle m_1, i_1\rangle : \langle m_1, c_1, i_1\rangle$ $\langle c_1, i_1\rangle : \langle m_1, c_1, i_1\rangle$

Fig. 12. Sample run of $\mathbb{C}\langle X\rangle$. The first row gives the received events; the second and the third rows give the content of Δ and $\mathbb{U}$, respectively, after every event is processed. Monitor states are represented symbolically in the table, e.g., $\sigma(1, \mathsf{createC})$ represents the state after the event createC.

other parameter instances. The third event $\mathsf{createC}\langle m_2, c_2\rangle$ is another creation event, incompatible with the second event. Hence, only one new monitor state is added to Δ. $\mathbb{U}$ is updated similarly. The last event $\mathsf{createI}\langle c_1, i_1\rangle$ is not a creation event. So no monitor instance is created for $\langle c_1, i_1\rangle$. It is compatible with the existing parameter instance $\langle m_1, c_1\rangle$ (found from the list for $\langle c_1\rangle$ in $\mathbb{U}$) introduced by the second event but not compatible with $\langle m_2, c_2\rangle$ due to the conflict binding on c. Therefore, a new monitor instance is created for the combined parameter instance $\langle m_1, c_1, i_1\rangle$ using the state for $\langle m_1, c_1\rangle$ in Δ. $\mathbb{U}$ is also updated to add the new instance into lists of parameter instances that are less informative.

4.2.2 Enable Set Optimization

While $\mathbb{C}\langle X\rangle$ is an improvement over $\mathbb{B}\langle X\rangle$, it is possible to improve $\mathbb{C}\langle X\rangle$ by making assumptions on the given monitor M. In other words, one may monitor properties written in any specification formalism, e.g., ERE, CFG, PTLTL etc., as long as one also provides a monitor generation algorithm for said formalism. However, this generality leads to extra monitoring overhead in some cases. It is possible to optimize monitor creation using the concept of *enable sets* [21]. Algorithms for computing enable sets can be found in Section 6.1 and 6.5.

To motivate the optimization, let us continue the run in Fig. 12 to process one more event, $\mathsf{useI}\langle i_1\rangle$. The result is shown in Fig. 13. $\mathsf{useI}\langle i_1\rangle$ is not a creation event and no monitor instance is created for $\langle i_1\rangle$. Since $\langle i_1\rangle$ is compatible with $\langle m_2, c_2\rangle$, a new monitor instance is defined for $\langle m_2, c_2, i_1\rangle$. The monitor instance for $\langle m_1, c_1, i_1\rangle$ is then updated according to useI because $\langle i_1\rangle$ is less informative than $\langle m_1, c_1, i_1\rangle$. $\mathbb{U}$ is also updated to add $\langle m_2, c_2, i_1\rangle$ to the lists for all the parameter instances less informative than $\langle m_2, c_2, i_1\rangle$. New entries are added into $\mathbb{U}$ during the update since some of the less informative parameter instances, e.g., $\langle m_2, i_1\rangle$, have not been used before this event.

Creating the monitor instance for $\langle m_2, c_2, i_1\rangle$ is needed for the correctness of $\mathbb{C}\langle X\rangle$, but it can be avoided when more information about the program or the specification is available.

	$\mathsf{useI}\langle i_1\rangle$
Δ	$\langle m_1, c_1\rangle : \sigma(1, \mathsf{createC})$ $\langle m_2, c_2\rangle : \sigma(1, \mathsf{createC})$ $\langle m_1, c_1, i_1\rangle : \sigma(\sigma(\sigma(1, \mathsf{createC}), \mathsf{createI}), \mathsf{useI})$ $\langle m_2, c_2, i_1\rangle : \sigma(\sigma(1, \mathsf{createC}), \mathsf{useI})$
$\mathbb{U}$	$\langle\rangle : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle, \langle m_2, c_2, i_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_1\rangle : \langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle c_1\rangle : \langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_2\rangle : \langle m_2, c_2\rangle, \langle m_2, c_2, i_1\rangle$ $\langle c_2\rangle : \langle m_2, c_2\rangle, \langle m_2, c_2, i_1\rangle$ $\langle i_1\rangle : \langle m_2, c_2, i_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_2, c_2\rangle : \langle m_2, c_2, i_1\rangle$ $\langle m_2, i_1\rangle : \langle m_2, c_2, i_1\rangle$ $\langle c_2, i_1\rangle : \langle m_2, c_2, i_1\rangle$ $\langle m_1, c_1\rangle : \langle m_1, c_1, i_1\rangle$ $\langle m_1, i_1\rangle : \langle m_1, c_1, i_1\rangle$ $\langle c_1, i_1\rangle : \langle m_1, c_1, i_1\rangle$

Fig. 13. Following the run of Fig. 12.

For example, according to the semantics of Iterator, no event $\mathsf{createI}\langle c_2, i_1\rangle$ will occur in the subsequent execution since an iterator can be associated to only one collection. Hence, the monitor for $\langle m_2, c_2, i_1\rangle$ will never reach the validation state and we do not need to create it from the beginning. However, such semantic information about the program is very difficult to infer automatically. Below, we show a simpler yet effective solution to avoid unnecessary monitor creations by analyzing the specification to monitor.

When monitoring a program against a specific property, usually only a certain subset of property categories, ($\mathcal{C}$ in Def. 2), is checked. For example, in the UnsafeMapIterator property in Fig. 11, the regular expression specifies a defective interaction among related Map, Collection and Iterator objects. To find an error in the program using monitoring is thus to detect matches of the specified pattern during the execution. In other words, we are only interested in the validation category of the specified pattern. Obviously, to match the pattern, for

a parameter instance of parameter set {m, c, i}, createC and createI should be observed before useI is encountered for the first time in monitoring. Otherwise, the trace slice for {m, c, i} will never match the pattern. Based on this information, we next show that creating the monitor state for $\langle m_2, c_2, i_1\rangle$ in Fig. 13 is not needed. When event useI$\langle i_1\rangle$ is encountered, if the monitor state for a parameter instance $\langle m_2, c_2\rangle$ exists without the monitor state for $\langle m_2, c_2, i_1\rangle$, like in Fig. 13, it can be inferred that in the trace slice for $\langle m_2, c_2, i_1\rangle$, only events createC and/or updateM occur before useI because, otherwise, if createI also occurred before useI, the monitor state for $\langle m_2, c_2, i_1\rangle$ should have been created. Therefore, we can infer, when event useI$\langle i_1\rangle$ is observed and before the execution continues, that no match of the specified pattern can be reached by the trace slice for $\langle m_2, c_2, i_1\rangle$, that is to say, the monitor for $\langle m_2, c_2, i_1\rangle$ will never reach the validation state.

This observation shows that the knowledge about the specified property can be applied to avoid unnecessary creation of monitor instances. This way, the sizes of Δ and $\mathbb{U}$ can be reduced, reducing the monitoring overhead. We next formalize the information needed for the optimization and argue that it is not specific to the underlying specification formalism. How this information is used is discussed in Section 4.2.3.

Definition 10. *Given $w, w_1, w_2, w_3 \in \mathcal{E}^*$ and $e, e' \in \mathcal{E}$, for a trace $w = w_1\ e'\ w_2\ e\ w_3$,* i.e.*, a trace w where e' occurs before an occurrence of e, we denote the relationship between e and e' with respect to w as $e' \rightsquigarrow_w e$. Let the **trace enable set** of $e \in \mathcal{E}$ be the function* $\mathsf{enable}_w : \mathcal{E} \to \mathcal{P}_f(\mathcal{E})$*, defined as:* $\mathsf{enable}_w(e) = \{e' \mid e' \rightsquigarrow_w e\}$.

Note that if $e \notin w$ then $\mathsf{enable}_w(e) = \emptyset$. The trace enable set can be used to examine whether the execution under observation may generate a particular trace of interest, or not: if event e is encountered during monitoring but some event $e' \in \mathsf{enable}_w(e)$ has not been observed, then the (incomplete) execution being monitored will *not* produce the trace w when it finishes. This observation can be extended to check, before an execution finishes, whether the execution can generate a trace belonging to some designated property categories. The designated categories are called the *goal* of the monitoring.

Definition 11. *Given $P : \mathcal{E}^* \to \mathcal{C}$ and a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, the **property enable set** is defined as a function* $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ *with* $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) = \{\mathsf{enable}_w(e) \mid P(w) \in \mathcal{G}\}$.

Intuitively, if event e is encountered during monitoring but none of event sets $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e)$ has been completely observed, the (incomplete) execution being monitoring will not produce a trace w s.t. $P(w) \in \mathcal{G}$. For example, given the regular expression specifying the UnsafeMapIterator property in Fig. 11, where $\mathcal{G} = \{\mathsf{match}\}$, the second column in Fig. 14 shows the property enable sets of events in UnsafeMapIterator.

The property enable set provides a sound and fast way to decide whether an incomplete trace slice has the possibility of reaching the desired categories by looking at the events that have already occurred. In the above example, if a trace slice starts with createC useI, it will never reach the match category, because $\{\mathsf{createC}\} \notin \mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(\mathsf{useI})$. In such case, no monitor state need be created even when the newly observed event may lead to new parameter instances. For example, suppose that the observed (incomplete) trace is createC useI from before. At the second event, useI, a new parameter instance can be constructed, namely, $\langle m_1, c_1, i_1\rangle$, and a monitor state s will be created for $\langle m_1, c_1, i_1\rangle$ if algorithm $\mathbb{C}\langle X\rangle$ is applied. However, since the trace slice for s is createC useI, we immediately know that s cannot reach state match. So there is no need to create and maintain s during monitoring if match is the goal.

Event	$\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$	$\mathsf{enable}^{X}_{\mathcal{G}}$
createC	{∅}	{∅}
createI	{{createC}, {createC, updateM}}	{{m, c}}
useI	{{createC, createI}, {createC, createI, updateM}}	{{m, c, i}}
updateM	{{createC}, {createC, createI}, {createC, createI, useI}}	{{m, c}, {m, c, i}}

Fig. 14. Property and Parameter Enable Sets for UnsafeMapIterator.

A direct application of the above idea to optimize $\mathbb{C}\langle X\rangle$ requires maintaining observed events for every created monitor and comparing event sets when a new parameter instance is found, reducing the performance. Therefore, we adapt the notion of the enable set to be based on parameter sets instead of event sets.

Definition 12. *Given a property $P : \mathcal{E}^* \to \mathcal{C}$, a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, a set of parameters X and a function $\mathcal{D}_{\mathcal{E}} : \mathcal{E} \to \mathcal{P}_f(X)$ mapping an event to its parameters, the **property parameter enable set** of event $e \in \mathcal{E}$ is defined as a function* $\mathsf{enable}^{X}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$ *as follows:* $\mathsf{enable}^{X}_{\mathcal{G}}(e) = \{\cup\{\mathcal{D}_{\mathcal{E}}(e') \mid e' \in \mathsf{enable}_w(e)\} \mid P(w) \in \mathcal{G}\}$.

From now on, we use "enable set" to refer to "property parameter enable set" for simplicity. For example, given the regular pattern for the UnsafeMapIterator property in Fig. 11 and $\mathcal{G} = \{\mathsf{match}\}$; the third column in Fig. 14 shows the parameter enable sets of events in UnsafeMapIterator. Then, given again the trace createC$\langle m_1, c_1\rangle$ useI$\langle i_1\rangle$, no monitor state need be created at the second event for $\langle m_1, c_1, i_1\rangle$ since the parameter instance used to initialize the new monitor state, namely, $\langle m_1, c_1\rangle$, is not in $\mathsf{enable}^{X}_{\mathcal{G}}(\mathsf{useI})$. In other words, one may simply compare the parameter instance used to initialize the new parameter instance with the enable set of the observed event to decide whether a new monitor state is needed or not. Note that in JavaMOP, the property parameter enable sets are generated from the property enable sets provided by the formalism plugin. This allows the plugins to remain totally parameter agnostic.

4.2.3 *Algorithm* $\mathbb{D}\langle X\rangle$

We next integrate the concept of enable sets with algorithm $\mathbb{C}\langle X\rangle$, to improve performance and memory usage [21].

	$\mathsf{e}_1\langle p_1\rangle$	$\mathsf{e}_2\langle q_1\rangle$	$\mathsf{e}_3\langle p_1, q_1\rangle$
Δ	$\langle p_1\rangle : \sigma(\iota, \mathsf{e}_1)$	$\langle p_1\rangle : \sigma(\iota, \mathsf{e}_1)$	$\langle p_1\rangle : \sigma(\iota, \mathsf{e}_1)$ $\langle p_1, q_1\rangle : \sigma(\sigma(\iota, \mathsf{e}_1), \mathsf{e}_3)$

Fig. 15. Unsound Usage of the Enable Set.

	$\mathsf{e}_1\langle p_1\rangle$	$\mathsf{e}_2\langle q_1\rangle$	$\mathsf{e}_3\langle p_1, q_1\rangle$
Δ	$\langle p_1\rangle : \sigma(\iota, \mathsf{e}_1)$	$\langle p_1\rangle : \sigma(\iota, \mathsf{e}_1)$	$\langle p_1\rangle : \sigma(\iota, \mathsf{e}_1)$ $\langle p_1, q_1\rangle : \sigma(\sigma(\iota, \mathsf{e}_1), \mathsf{e}_3)$
$\mathcal{T}$	$\langle p_1\rangle : 1$	$\langle p_1\rangle : 1$	$\langle p_1\rangle : 1$
disable	$\langle p_1\rangle : 2$	$\langle p_1\rangle : 2$ $\langle q_1\rangle : 3$	$\langle p_1\rangle : 2$ $\langle q_1\rangle : 3$ $\langle p_1, q_1\rangle : 4$

Fig. 16. Sound Monitoring Using Timestamps.

Given a set of desired verdict categories $\mathcal{G}$, we are guaranteed that we can optimize the monitoring process by omitting creating monitor states for certain parameter instances when an event is received using the enable set without missing any trace belonging to $\mathcal{G}$. However, skipping the creation of monitor states may result in false alarms, i.e., a trace that is not in $\mathcal{G}$ can be reported to belong to $\mathcal{G}$. Let us consider the following example. We monitor to find matching of a regular pattern $\mathsf{e}_1\mathsf{e}_3$. Relevant events and their parameters are $\mathsf{e}_1(p), \mathsf{e}_2(q), \mathsf{e}_3(p, q)$. The observed trace is $\mathsf{e}_1\langle p_1\rangle\mathsf{e}_2\langle q_1\rangle\mathsf{e}_3\langle p_1, q_1\rangle$. Also, suppose e_1 is the only creation event. Obviously, the trace does not match the pattern. Fig. 15 shows the run using the enable set optimization (i.e., not creating monitor states for parameter instances disallowed by the enable sets). Only the content of Δ is given for simplicity. At e_1, a monitor state is created for $\langle p_1\rangle$ since it is the creation event. At e_2, no action is taken since $\mathsf{enable}^X_{\mathcal{G}}(\mathsf{e}_2) = \emptyset$. At e_3, a monitor state will be created for $\langle p_1, q_1\rangle$ using the monitor state for $\langle p \mapsto p_1\rangle$ since $\mathsf{enable}^X_{\mathcal{G}}(\mathsf{e}_3) = \{\{p\}\}$. This way, e_2 is forgotten and a match of the pattern is reported incorrectly.

To avoid unsoundness, we introduce the notion of disable stamps of events. $\mathsf{disable} : [[X \rightarrow V] \rightarrow \mathsf{integer}]$ maps a parameter instance to an integer timestamp. $\mathsf{disable}(\theta)$ gives the time when the last event with θ was received. We maintain timestamps for monitors using a mapping $\mathcal{T} : [[X \rightarrow V] \rightarrow \mathsf{integer}]$. $\mathcal{T}$ maps a parameter instance for which a monitor state is defined to the time when the original monitor state is created from a creation event. Specifically, if a monitor state for θ is created using the initial state when a creation event is received (i.e., using the defineNew function in algorithm $\mathbb{C}\langle X\rangle$), $\mathcal{T}(\theta)$ is set to the time of creation; if a monitor state for θ is created from the monitor state for θ', $\mathcal{T}(\theta')$ is passed to $\mathcal{T}(\theta)$. Fig. 16 shows the evolution of disable and $\mathcal{T}$ while processing the trace in Fig. 15.

disable and $\mathcal{T}$ can be used together to track "skipped events": when a monitor state for θ is created using the monitor state for θ', if there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ then the trace slice for θ does not belong to the desired verdict categories $\mathcal{G}$. Intuitively, $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ implies that an event $e\langle\theta''\rangle$ has been encountered after the monitor state for θ' was created. But θ'' was not taken into account ($\theta'' \not\sqsubseteq \theta'$). The only possibility is that e is omitted due to the enable set and thus the trace slice for θ does not belong to $\mathcal{G}$ according to the definition of

```
Algorithm 𝔻⟨X⟩(M = (S, ℰ, 𝒞, ι, σ, γ))

Input : mapping enable^X_𝒢 : [ℰ→𝒫_f(𝒫_f(X))]

Globals : mapping Δ : [[X→V]→S]
          mapping 𝒯 : [[X→V]→integer]
          mapping 𝕌 : [X→V] → 𝒫_f([X→V])
          mapping disable : [[X→V]→integer]
          integer timestamp

Initialization : 𝒯 ← ⟨⟩, 𝕌 ← ⟨⟩, disable ← ⟨⟩
                 disable(θ) ← 0 for any θ
                 𝕌(θ) ← ∅ for any θ, timestamp ← 0

function main(e⟨θ⟩)
 1 if Δ(θ) undefined then
 2 : createNewMonitorState(e⟨θ⟩)
 3 : if Δ(θ) undefined and e is a creation event then
 4 : : defineNew(θ)
 5 : endif
 6 : disable(θ) ← timestamp; timestamp ← timestamp + 1
 7 endif
 8 foreach θ' ∈ {θ} ∪ 𝕌(θ) s.t. Δ(θ') defined do
 9 : Δ(θ') ← σ(Δ(θ'), e)
10 endfor

function createNewMonitorStates(e⟨θ⟩)
 1 foreach X_e ∈ enable^X_𝒢(e) (in reversed topological order) do
 2 : if Dom(θ) ⋤ X_e then
 3 : : θ_m ← θ' s.t. θ' ⊏ θ and Dom(θ') = Dom(θ) ∩ X_e
 4 : : foreach θ'' ∈ 𝕌(θ_m) ∪ {θ_m} s.t. Dom(θ'') = X_e do
 5 : : : if Δ(θ'') defined and Δ(θ'' ⊔ θ) undefined then
 6 : : : : defineTo(θ'' ⊔ θ, θ'')
 7 : : : endif
 8 : : endfor
 9 : endif
10 endfor

function defineNew(θ)
 1 foreach θ'' ⊏ θ do
 2 : if Δ(θ'') defined then return endif
 3 endfor
 4 Δ(θ) ← ι; 𝒯(θ) ← timestamp; timestamp ← timestamp + 1
 5 foreach θ'' ⊏ θ do 𝕌(θ'') ← 𝕌(θ'') ∪ {θ} endfor

function defineTo(θ, θ')
 1 foreach θ'' ⊑ θ s.t. θ'' ⋢ θ' do
 2 : if disable(θ'') > 𝒯(θ') or 𝒯(θ'') < 𝒯(θ') then
 3 : : return
 4 : endif
 5 endfor
 6 Δ(θ) ← Δ(θ'); 𝒯(θ) ← 𝒯(θ')
 7 foreach θ'' ⊏ θ do 𝕌(θ'') ← 𝕌(θ'') ∪ {θ} endfor
```

Fig. 17. Optimized Monitoring Algorithm $\mathbb{D}\langle X\rangle$.

the enable set. Therefore, in Fig. 16, no monitor instance is created for $\langle p_1, q_1\rangle$ at e_3 because $\mathsf{disable}(\langle q_1\rangle) > \mathcal{T}(\langle p_1\rangle)$.

The above discussion applies when the skipped event occurs after the initial creation of the monitor state. The other case, i.e., an event is omitted before the initial monitor state is created, can also be handled using timestamps. If the skipped event is not a creation event, it does not affect the soundness of the algorithm because of the definition of creation events. In the above example, if the observed trace is $\mathsf{e}_2\langle q_1\rangle\mathsf{e}_1\langle p_1\rangle\mathsf{e}_3\langle p_1, q_1\rangle$, we will ignore e_2 and report the matching at e_3 since e_1 is the only creation event. It is more sophisticated (but not much different) when the skipped event is a creation event.

Based on the above discussion, we develop a new parametric monitoring algorithm that optimizes algorithm $\mathbb{C}\langle X\rangle$ using the enable set and timestamps, as shown in Fig. 17. This algorithm makes use of the mappings discussed above, namely, $\mathsf{enable}^X_{\mathcal{G}}$, Δ, $\mathbb{U}$, disable, and $\mathcal{T}$, and maintains an integer variable to track the timestamp. Similar to algorithm $\mathbb{C}\langle X\rangle$, when event $e\langle\theta\rangle$ is received, algorithm $\mathbb{D}\langle X\rangle$ first checks whether $\Delta(\theta)$ is defined or not (line 1 in main). If not, monitor states may be generated for new encountered parameter instances, which is achieved by function createNewMonitorStates in algorithm $\mathbb{D}\langle X\rangle$. Unlike in algorithm $\mathbb{C}\langle X\rangle$, where all the parameter instances less informative than θ are searched to find all the compatible parameter instances using $\mathbb{U}$, createNewMonitorStates enumerates parameter sets in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and looks for parameter instances whose domains are in $\mathsf{enable}^X_{\mathcal{G}}(e)$ and which are compatible with θ, also using $\mathbb{U}$. The inclusion check at line 2 in createNewMonitorStates is to omit unnecessary search since if $\mathsf{Dom}(\theta) \subseteq X_e$ then no new parameter instance will be created from θ. This way, createNewMonitorStates creates all the parameter instances from θ whenever the enable set of e is satisfied using fewer lists in $\mathbb{U}$.

If e is a creation event then a monitor state for θ is initialized (lines 3 - 5 in main). Note that $\Delta(\theta)$ can be defined in function createNewMonitorStates if $\Delta(\theta')$ has been defined for some $\theta' \sqsubset \theta$. $\mathsf{disable}(\theta)$ is set to the current timestamp after all the creations and the timestamp is increased (line 6 in main). The rest of function main in $\mathbb{D}\langle X\rangle$ is the same as in $\mathbb{C}\langle X\rangle$: all the relevant monitor states are updated according to e. Function defineNew in $\mathbb{D}\langle X\rangle$ first searches for a defined sub-instance of θ. If such instance exists, θ should be defined using it; otherwise, $\Delta(\theta)$ is set to the initial state. Then $\mathcal{T}(\theta)$ is set to the current timestamp, and the timestamp is incremented. Function defineTo in $\mathbb{D}\langle X\rangle$ checks disable and $\mathcal{T}$ as discussed above to decide whether $\Delta(\theta)$ can be defined using $\Delta(\theta')$. If $\Delta(\theta)$ is defined using $\Delta(\theta')$, $\mathcal{T}(\theta)$ is set to $\mathcal{T}(\theta')$. Both functions then add θ to the sets in table $\mathbb{U}$ for the bindings less informative than θ, as in $\mathbb{C}\langle X\rangle$.

In all of our tested cases $\mathbb{D}\langle X\rangle$ performs better than $\mathbb{C}\langle X\rangle$; in most cases that $\mathbb{C}\langle X\rangle$ or $\mathbb{D}\langle X\rangle$ caused notable monitoring overhead, the efficiency of $\mathbb{D}\langle X\rangle$ is significantly better (see Section 4.5).

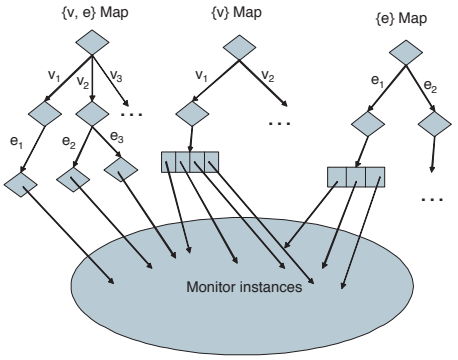


Fig. 18. Centralized indexing for the JavaMOP spec in Fig. 2

4.3 Indexing

Algorithm $\mathbb{D}\langle X\rangle$ gives an efficient algorithm for monitoring parametric specifications, but it tells us nothing about the data structures used to map parameter instances to their given monitor states. The following techniques have been adapted from [23] to algorithm $\mathbb{D}\langle X\rangle$. $\mathbb{D}\langle X\rangle$ is more powerful than the monitoring algorithm used in [23].

4.3.1 Centralized Indexing

Efficient monitor lookup is crucial to reduce the runtime overhead. The major requirement here is to quickly locate all related monitor instances given a parameter instance. Recall that different events can have different sets of parameters: e.g., in Fig. 11, all four events declare different parameter subsets. Our centralized indexing algorithm constructs multiple indexing trees according to the event definitions to avoid inefficient traversal of the indexes; more specifically, for every distinct set of event parameters found in the specification, an indexing tree is created to map the set of parameters directly into the list of corresponding monitors.

The number and structure of indexing trees needed for a specification can be determined by a simple static analysis of event parameter declarations. For example, from our original parametric specification in Fig. 2, since there are three different sets of event parameters, namely $\langle v, e\rangle$, $\langle v\rangle$, and $\langle e\rangle$, three indexing trees will be created to index monitors, as illustrated in Fig. 18: the first tree uses a pair of v and e to find the corresponding monitor, while the other two map v and, respectively, e to the list of related monitors.

We use hash maps in JavaMOP to construct the indexing tree. Fig. 19 shows the generated monitor lookup code for the updateV event in Fig. 2. This code is inserted at the end of every call to a Vector.add method[6] or to Vector.remove, according to the event definition. One parameter is associated to this event, namely, the vector v on which we invoke the method. A map, SafeEnum_v_map, is created to store the indexing information for v, i.e., the $\{v\}$Map in Fig. 18. When such a method call is encountered during the execution, a concrete vector object will be bound to v and the monitoring code

[6] The '*' tells AspectJ to weave at any method that begins with add.

```
Map SafeEnum_v_map = makeMap();
pointcut SafeEnum_updateV0(Vector v) :
       (call(* Vector.add*(..))||call(* Vector.remove(..)))&& target(v);
after (Vector v) : SafeEnum_updateV0(v) {
  Map m = SafeEnum_v_map;
  Object obj = null;
  obj = m.get(v);
  if(obj != null){
    Iterator monitors = ((List)obj).iterator();
    while (monitors.hasNext()) {
      SafeEnumMonitor monitor =
      (SafeEnumMonitor)monitors.next();
      monitor.updateV(v);
      if(monitor.MOP_fail()) {
        //fail handler
      }
    }//end of while
  }//end of if
}
```

Fig. 19. Centralized indexing monitoring code generated by JavaMOP for updateV (from the JavaMOP spec in Fig. 2)

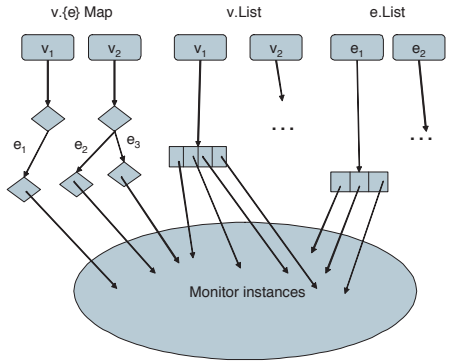


Fig. 20. Decentralized indexing for the JavaMOP spec in Fig. 2

will be triggered to fetch the list of related monitors using SafeEnum_v_map. Then all the monitors in the list will be invoked to process the event.

A performance-related concern in our implementation of JavaMOP is to avoid memory leaks caused by hash maps. The values of parameters are stored in hash maps as key values. When these values are objects in the system, this might prevent the Java garbage collector from removing them even when the original program has released all references to them. We use weakly referenced hash maps in JavaMOP. The weakly referenced hash map only maintains weak references to key values; hence, when an object that is a key in the hash map dies in the original program, it can be garbage collected and the corresponding key-value pair will also be removed from the hash map. This way, once a monitor instance becomes unreachable, it can also be garbage collected and its allocated memory released. Note that a monitor instance will be destroyed *only* when it will never be triggered in the future because the monitor is not destroyed unless all parameters are garbage collected. If a future event can ever trigger monitor instance m, then m is not garbage collectible. This guarantees the soundness of our usage of weak references.

```
List Vector.SafeEnum_v_List = null;
pointcut SafeEnum_updateV0(Vector v) :
       (call(* Vector.add*(..))||call(* Vector.remove(..)))&& target(v);
 after (Vector v) : SafeEnum_updateV0(v) {
   if (v.SafeEnum_v_List != null) {
     Iterator monitors =
      (v.SafeEnum_v_List).iterator();
     while (monitors.hasNext()) {
       SafeEnumMonitor monitor =
        (SafeEnumMonitor)monitors.next();
       monitor.updateV(v);
       if(monitor.MOP_fail()) {
         //fail handler
       }
     }//end of while
   }//end of if
}
```

Fig. 21. Decentralized indexing monitoring code automatically generated by JavaMOP for updateV

4.3.2 Optimization: Decentralized Indexing

The centralized-indexing approach above can be regarded as a centralized database of monitors. This solution proves to be acceptable with respect to runtime overhead in many of the experiments that we have carried out. However, reducing runtime overhead is and will always be a concern in runtime verification. We next propose a further optimization based on decentralizing the indexing. This optimization is also implemented in JavaMOP using the decentralized keyword. Theoretically, decentralized indexing will not *always* outperform centralized indexing, but in our experiments it has.

In *decentralized indexing*, the indexing trees are piggybacked into states of objects to reduce the lookup overhead. For every distinct subset of parameters that appear as a parameter of some event, JavaMOP automatically chooses one of the parameters as the *master parameter* and uses the other parameters, if any, to build the indexing tree using hash maps as before; the resulting map will then be declared as a new field of the master parameter. For example, for the updateV event in Fig. 2, since it has only the $\langle$v$\rangle$ parameter, v is selected as master parameter and a new field will be added to its Vector class to accommodate the list of related monitor instances at runtime. Fig. 20 shows the decentralized version of the centralized indexing example in Fig. 18.

Comparing Figs. 21 and 19, one can see that the major difference between the centralized and the decentralized indexing approaches is that the list of monitors related to v can be directly retrieved from v when using decentralized indexing; in centralized indexing we need to look up the list from a hash map. Decentralized indexing thus scatters the indexing over objects in the system and avoids unnecessary lookup operations, reducing both runtime overhead and memory usage. It is worth noting that decentralized indexing does *not* affect the behavior of disposing unnecessary monitor instances as discussed in the previous section: when an object is disposed, all the references to monitor instances based on this object will also be discarded, no matter whether they are stored in maps using weak references or whether they are embedded as fields of the object.

On the negative side, decentralized indexing involves more instrumentation than the centralized approach, sometimes beyond the boundaries of the monitored program, since it needs to modify the original signature of the master parameter: for the monitoring code in Fig. 21, the Java library class Vector has to be instrumented (add a new field). This is usually acceptable for testing/debugging purposes, but may not be appropriate if we use MOP as a development paradigm and thus want to leave monitors as part of the released program. If that is the case, then one should use centralized indexing instead, by not using the modifier decentralized.

The choice of the master parameter may significantly affect the runtime overhead. In the specification in Fig. 2, since there is a one-to-many relationship between Collections and Iterators, it would be more effective to choose the Iterator as the master parameter of the create event. Presently, JavaMOP picks the first parameter encountered in the analysis of the MOP specification as the master parameter for each set of event parameters. Hence, the user can control the choice of the master parameter by putting, for each set of parameters P, the desired master parameter first in the list of parameters of the first event parametric over P.

4.4 Binding Modes and Connectedness

There are three parameter binding modes available in JavaMOP, as well as the concept of connectedness, which may be used in conjunction with any binding mode. These modes and connectedness determine which monitor instances are allowed to report verdict categories (e.g., match or fail). This allows a user to essentially apply a filter on the number of results they want from their monitors. For each binding mode we will consider the pattern:

$$\Lambda a, b \,.\, \mathsf{e}_1\langle\rangle(\mathsf{e}_2\langle a, b\rangle \mid \mathsf{e}_3\langle b\rangle)^*$$

And the following trace:

$$\mathsf{e}_1\langle\rangle \;\; \mathsf{e}_2\langle a_1, b_1\rangle \;\; \mathsf{e}_3\langle b_1\rangle$$

The default binding mode is the any-binding mode. In this mode any instance monitor is allowed to report categories. When the above trace is monitored using any-binding four matches are reported, one on each of the first two events as they arrive and two for the last event, e_3. Two matches are reported when e_3 arrives because one is reported from the monitor instance for $\langle a_1, b_1\rangle$ and one is reported from the monitor instance for $\langle b_1\rangle$, and categories can be reported from any monitor instance. The trace slices for each monitor instance after all three events can be seen in Fig. 22. Note that each one matches the pattern, and that the one for $\langle a_1, b_1\rangle$ has *two* prefixes that match the pattern ($\mathsf{e}_1\ \mathsf{e}_2$ and $\mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3$), resulting in four total matches, as expected.

The other extreme, full-binding is allowing only those instance monitors that correspond to fully instantiated parameter instances to report categories. This is similar to the semantics of Tracematches [4, 8, 18]. Looking at our example trace, two matches will be reported because two prefixes of the trace $\mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3$, which comes from the only instance we consider ($\langle a_1, b_1\rangle$), match (see Fig. 22). To implement this mode, during monitor generation we count the number of parameters in the parameter list of the monitor. Whenever we check a monitor instance for a category (such as the check for the match category in Figs. 19 and 21) we compare the number of bound parameters of the monitor instance to the number of parameters of the specification. If the numbers of parameters do not match, the monitor instance's output is ignored.

Instance	Trace
$\langle\rangle$	e_1
$\langle b_1\rangle$	$\mathsf{e}_1\ \mathsf{e}_3$
$\langle a_1, b_1\rangle$	$\mathsf{e}_1\ \mathsf{e}_2\ \mathsf{e}_3$

Fig. 22. Trace Slices for Binding Mode Example Trace

In between these two extremes is maximal-binding. The "less informative or as informative as" relation "$\sqsubseteq$" in Def. 4 induces a partial order over parameter instances. In this mode we only report verdict categories from those instances that are *currently* maximal in that partial order. Considering again our example, this is a bit more complex. When event e_1 arrives, the instance $\langle\rangle$ *is maximal*, so a match is reported. When e_2 arrives the new maximal instance is $\langle a_1, b_1\rangle$, and a match is reported. When e_3 arrives, $\langle a_1, b_1\rangle$ is still larger than $\langle b_1\rangle$ so only the instance $\langle a_1, b_1\rangle$ is allowed to report a match, thus only three matches are reported, unlike the four from any-binding. To implement this binding mode, all monitor instances contain flags; when defineTo in Algorithm $\mathbb{D}\langle X\rangle$ (Fig. 17) defines a new monitor from a less informative monitor the flag in the less informative monitor is set to false. Additionally, when a new instance is created we must check if there is a monitor instance for a more informative parameter instance already in existence, as happens in our example, and set the flag to false if there is. Results from the monitor instance are only reported when the flag is true.

Connectedness, which may be used to augment any binding mode, filters out all those monitor instances for which the parameters are not connected to each other by some event. For example, if events $\mathsf{e}_1\langle p_1\rangle$ and $\mathsf{e}_2\langle q_1\rangle$ are the only events that have been seen that are sent to the $\langle p_1, q_1\rangle$ instance, no categories will be reported from that instance until some event such as $\mathsf{e}_3\langle p_1, q_1\rangle$ occurs. For connectedness we will consider again the SafeEnum property of Fig. 2, and we will consider the trace:

$$\mathsf{updateV}\langle v_1\rangle\ \mathsf{createE}\langle v_1, e_1\rangle\ \mathsf{updateV}\langle v_2\rangle\ \mathsf{useE}\langle e_1\rangle$$

Nothing violating the SafeEnum condition of not using an Enumeration created from a Vector that has been modified occurs in this pattern, however, because of the generic parametric trace slicing algorithm, some monitor instances will be generated that will flag failures, as can be seen in Fig. 23. With connectedness the results of extraneous monitor instances such as $\langle v_2, e_1\rangle$, which would signal an undesired fail verdict, can be filtered out. Note that the instance for $\langle v_2, e_1\rangle$ must be created by the generic parametric monitoring algorithm, because it has no way to know that $\langle v_2, e_1\rangle$ cannot be connected at

Instance	Trace
$\langle v_1 \rangle$	updateV
$\langle e_1 \rangle$	useE
$\langle v_2 \rangle$	updateV
$\langle v_1, e_1 \rangle$	updateV createE useE
$\langle v_2, e_1 \rangle$	updateV useE

Fig. 23. Trace Slices for Connectedness Example Trace

some time in the future, as it has no semantic knowledge of the createE event. Connectedness can be added to a specification via the connected keyword shown in Fig. 7, and is implemented using a union-find data structure in which each set represents parameter objects which have been connected by a given event. When an event arrives, all of its associated parameter objects are unioned in the union-find data structure. When a monitor instance attempts to report a verdict category, the union-find data structure is queried to ensure that all parameter objects of the instance are in the same set of the union-find.

4.5 JavaMOP Evaluation

The evaluation presented here is a summary of the evaluation of JavaMOP that we performed in the summer of 2009 and described in [21]. Algorithms $\mathbb{C}\langle X \rangle$ and $\mathbb{D}\langle X \rangle$ described in Section 4.2 were both used to showcase the effectiveness of using enable sets to reduce extraneous monitor creations. Results from Tracematches [4,8,18] are also presented because it is the most efficient Java monitoring system of which we are aware, other than JavaMOP.

Experimental Settings.

Our experiments were performed on a machine with 2GB RAM and a Pentium 4 2.66GHz processor using Ubuntu Linux 7.10. We used version 2006-10 of the DaCapo benchmark suite [15]. The default input for DaCapo was used, and we use the -converge option to ensure the validity of our test by running each test multiple times, until the execution time converges. After convergence, the runtime is stabilized within 3%, thus numbers in Fig. 24 should be interpreted as "$\pm$3%". Additional code introduced by the AspectJ weaving process changes the program structure in DaCapo, sometimes causing the benchmark to run slightly faster due to better instruction cache layout. Both of these facts account for the negative overheads.

Properties. These properties, borrowed from [17, 18, 47] and specified using formalisms discussed in Section 6, were used in the evaluation of JavaMOP.

- UnsafeMapIterator: Do not update a Map when using the Iterator interface to iterate its values or its keys;
- UnsafeSyncCollection: If a Collection is synchronized, then its iterator also should be accessed synchronously;
- UnsafeSyncMap: If a Collection is synchronized, then its iterators on values and keys also should be accessed in a synchronized manner;
- UnsafeIterator: Do not update a Collection when using the Iterator interface to iterate its elements;
- UnsafeFileWriter: Do not write to a FileWriter after closing.

UnsafeMapIterator and UnsafeIterator specify properties that catch conditions in which the Java Virtual Machine will throw an exception. However, the exception will not always be properly thrown when the Map or Collection in question is modified in a separate thread from the thread iterating over the Map or Collection. Also, each of these properties is relevant to most of our benchmarks, and show large overheads that are useful for comparisons between Tracematches and JavaMOP, which is the goal of this evaluation.

UnsafeFileWriter cannot be expressed in Tracematches because it is a context-free property.

Results and Discussion. Figs. 24 and 25 summarize the results of the experiments from [21]. Fig. 24 shows the percent overheads of $\mathbb{C}\langle X \rangle$, $\mathbb{D}\langle X \rangle$, and Tracematches. All the properties were heavily monitored in the experiments. Millions of parameter instances were observed for some properties under monitoring, e.g., UnsafeIterator, putting a critical test on the generated monitoring code. Note that Soot [59, 61], the underlying bytecode engine for Tracematches, cannot handle the DaCapo benchmark properly, resulting in fewer instrumentation points in the pmd program. Accordingly, our specifications are modified to have the same scope of instrumentation in pmd for a fair comparison. All three systems generated low runtime overhead in most experiments, showing their efficiency. For $\mathbb{D}\langle X \rangle$, only 7 out of 66 cases caused more than 10% runtime overhead. The numbers for $\mathbb{C}\langle X \rangle$ and Tracematches are 9 out of 66 and 15 out of 44, respectively. Fig. 25 shows the comparison among three systems using 7 cases where significant numbers of monitors were created in monitoring. Fig. 25 (A) compares runtime overhead. In all cases, $\mathbb{D}\langle X \rangle$ outperformed the other two and $\mathbb{C}\langle X \rangle$ is better than Tracematches. This shows that JavaMOP provides an efficient solution to monitor parametric specifications despite its genericity in terms of specification formalisms. The results also illustrate the effectiveness of the enable set based optimization: on average, the overhead of $\mathbb{D}\langle X \rangle$ is about 20% less than $\mathbb{C}\langle X \rangle$. Moreover, when the property to monitor becomes more complicated, the improvement achieved by the optimization is more significant. In the two extreme cases, namely, bloat-UnsafeMapIterator and pmd-UnsafeMapIterator, where both the non-optimized JavaMOP and Tracematches ran out of memory, the optimized JavaMOP managed to finish the executions with overheads that are reasonable for many applications, such as testing and debugging.

Fig. 25 (B) shows the maximum memory usages of our experiments in log_{10} scale, in Megabytes. It shows that the enable set optimization does not always reduce *peak* memory usage. In two out of five pictured cases, algorithm $\mathbb{D}\langle X \rangle$ produces significantly lower peak memory usage than $\mathbb{C}\langle X \rangle$. In fact, in bloat-UnsafeMapIterator and pmd-UnsafeMapIterator, where $\mathbb{C}\langle X \rangle$ ran out of memory, $\mathbb{D}\langle X \rangle$ managed to complete. In general, peak memory usage with $\mathbb{D}\langle X \rangle$ was never observed to be higher than $\mathbb{C}\langle X \rangle$ by any significant amount. Wherever

	UnsafeMapIterator			UnsafeSyncCollection			UnsafeSyncMap			UnsafeIterator			UnsafeFileWriter	
	TM	$\mathbb{C}\langle X\rangle$	$\mathbb{D}\langle X\rangle$	TM	$\mathbb{C}\langle X\rangle$	$\mathbb{D}\langle X\rangle$	TM	$\mathbb{C}\langle X\rangle$	$\mathbb{D}\langle X\rangle$	TM	$\mathbb{C}\langle X\rangle$	$\mathbb{D}\langle X\rangle$	$\mathbb{C}\langle X\rangle$	$\mathbb{D}\langle X\rangle$
antlr	-2	5	2	-2	2	1	-3	2	1	0	0	0	2	5
bloat	OOM	OOM	935	1448	735	712	2267	858	660	11258	769	749	5	0
chart	-1	4	0	0	1	1	1	3	0	11	5	3	-1	0
eclipse	8	2	1	0	0	0	0	1	1	2	0	1	1	2
fop	11	-2	-3	-4	-3	0	16	-5	-3	5	4	1	-3	-5
hsqldb	29	0	0	24	0	0	22	-1	0	17	-1	0	0	-1
jython	57	11	7	6	-4	-4	8	-4	-5	16	-2	0	-3	-5
luindex	7	12	5	0	1	1	3	1	4	9	3	5	-1	-1
lusearch	9	1	-1	9	1	1	8	2	-1	34	4	2	-1	0
pmd	OOM	OOM	196	33	18	15	50	21	12	196	19	14	-2	-2
xalan	10	4	4	7	-1	1	6	0	0	10	9	8	2	1

Fig. 24. Average *Percent* Runtime Overhead for Tracematches(TM), $\mathbb{C}\langle X\rangle$, and $\mathbb{D}\langle X\rangle$ (convergence within 3%, OOM = Out of Memory). $\mathbb{D}\langle X\rangle$, at its worst, has less than an order of magnitude of overhead.

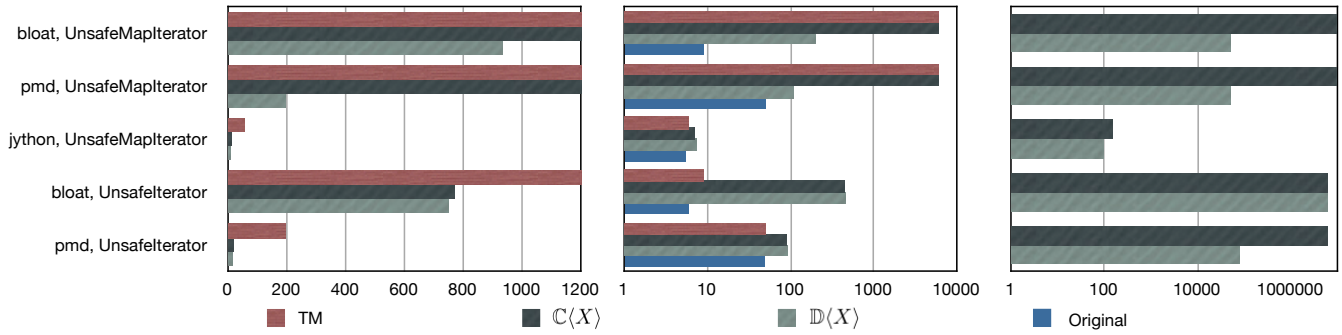


Fig. 25. Statistics. (A): Runtime Overhead. (B) Peak Memory Usage. (C) Number of Monitor Instances.

$\mathbb{C}\langle X\rangle$ was observed to have lower peak memory usage, it was at the expense of more garbage collections than with $\mathbb{D}\langle X\rangle$. For example, in pmd-UnsafeIterator, the $\mathbb{C}\langle X\rangle$ monitored program had 1361 young generation garbage collections while the $\mathbb{D}\langle X\rangle$ monitored program had 1167 collections. Of course, fewer garbage collection cycles contributes to the performance increase of the enable set optimization. This observation also applies to Tracematches: in three out of five cases, Tracematches caused less peak memory usage with more garbage collection but more runtime overhead.

Fig. 25 (C) shows the number of monitor instances generated by $\mathbb{C}\langle X\rangle$ and $\mathbb{D}\langle X\rangle$. Tracematches is absent from the graph because it does not produce monitor instances per se. In general fewer monitor instances are generated by $\mathbb{D}\langle X\rangle$.

5 BusMOP

Every MOP instance has issues specific to its domain. BusMOP must deal with interfacing with buses at a hardware level. There is also complexity in the mechanisms for recovery actions, which require specialized hardware modules. We first provide an introduction to the PCI bus, which is the currently supported bus architecture for BusMOP. While we intend to support more bus architectures in the future, the operation of the PCI bus was instrumental in guiding the design of BusMOP. We then discuss the design of the BusMOP monitoring device. Particular care must be taken to support certain features of the current MOP logical formalism semantics that expect serialized events. To use BusMOP, one writes a property for a specific peripheral, say p, that is plugged into the bus. The property is then synthesized onto an FPGA that is also plugged into the bus. A small program is used to write the proper value of the Base Access Register (BAR) for p to the FPGA (see below for more explanation on the BARs).

5.1 PCI Bus

While BusMOP [52] is a generic approach to generating hardware based monitors that can be adapted to any bus structure[7]. the current implementation is specific to the PCI Bus architecture. The idiosyncrasies of the PCI bus architecture guided the design of BusMOP.

The Peripheral Component Interconnect (PCI) is the current standard family of communication architectures for motherboard/peripheral interconnection in the personal computer market; it is also widely popular in the embedded domain [50]. The standard can be divided in two parts: a *logical* specification, which details how the CPU configures and accesses peripherals through the system controller, and a *physical* specification, which details how peripherals are connected to and communicate with the motherboard. While the logical specification has remained largely unaltered since the introduction

[7] As demonstrated in [53].

of the original PCI 1.0 standard in 1992, several different physical specifications have emerged since then.

One of the main features of the logical layer is plug-and-play (automatic configuration) functionality. On start-up, the OS executes a PCI base driver which reads information from special configuration registers implemented by each PCI-compliant peripheral and uses them to configure the system. Of peculiar importance is a set of up to 6 Base Access Registers (BARs) (recall the baseN registers from Section 3.3.1). Each BAR represents a request by the peripheral for a block of addresses in either the I/O or memory space; the PCI base driver is responsible for accepting such requests, allocating address blocks and communicating back the chosen addresses to the peripheral, by writing in the BARs.

To communicate with the peripheral, the CPU can, then, issue write and read commands, called *transactions*, to either I/O or memory space; each peripheral is required to implement *bus slave* logic, which decodes and responds to transactions targeting all address spaces allocated to the peripheral. Typically, address spaces are used to implement either registers, which control and determine the logical status of the peripheral, or data buffers. Peripherals *can* also implement *bus master* logic: they can autonomously initiate read and write transactions to either main memory or the address space of another peripheral. Master mode is typically used by high-performance peripherals to perform a DMA transfer, i.e., transfer data from the peripheral to a buffer in main memory. The peripheral's driver can then read the data directly from memory, which is much faster than issuing a read transaction on the bus. Finally, each peripheral is provided with an interrupt line that can be used to send interrupts to the CPU.

There are two main flavors of physical architecture: both PCI and PCI-X are parallel, while PCI-E is serial but runs at much higher frequency (2.5Ghz against up to 133Mhz for PCI-X). We have focused on PCI/PCI-X,[8] which implements a shared bus architecture. The logical PCI tree is physically divided into bus segments, and most bus wires are shared among all peripherals connected to a single segment. We refer to [50] for detailed bus specifications. Each transaction seen on the bus consists of an address phase, which provides the initial address in either memory or I/O space, followed by one or more data phases, each of which carries up to 32 or 64 bits of data for PCI/PCI-X, respectively (individual bytes can be masked using *byte enables*). Since each bus segment is shared, arbitration is required to determine which master peripheral is allowed to transmit at any one time. Arbitration uses two active-low, point-to-point wires between the peripheral and the bus segment arbiter, REQ# and GNT#. A standard request-grant handshake is used, where the peripheral first lowers REQ# to request access to the bus, and the arbiter grants permission to start a new transaction by lowering GNT#.

[8] We also plan to extend our design to PCI-E, as mentioned in Section 3.3.1.

5.2 Monitoring Device

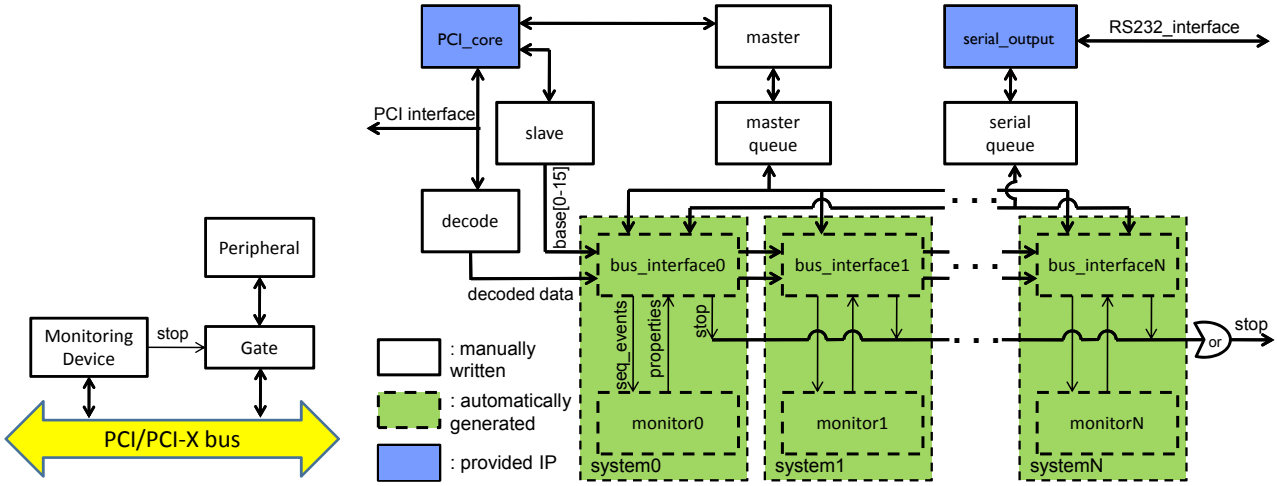
The current version of BusMOP is designed for the Xilinx ML455 board [62], but adapting the generated code to different FPGA boards takes minimal effort. The monitoring device uses a mixed VHDL/Verilog register transfer level (RTL) description. The board is outfitted with a Virtex-4 FPGA and it can be plugged into a standard 3.3V PCI/PCI-X socket. The FPGA implements both a slave and a master peripheral module, together with the monitoring modules. Events for the system are specified in terms of read/write data transfers on the bus and interrupt requests; the device continuously "sniffs" all ongoing activities on the bus, and is therefore able to monitor communication for all other peripherals located on the same bus segment. Whenever a failure to meet the specification is detected, the device can execute a recovery action using strategies based on the detected error.

For a vast category of errors that involve incorrect interaction between the peripheral and its software driver, it is often possible to recover from the failure by forcing the peripheral into a consistent state. The monitoring device implements a master module, and can therefore initiate transactions on the bus. For example, consider a common type of error, where the driver fails to validate some input from the user and as a result writes an invalid value to a register in the peripheral. We can recover by rewriting the register with a valid value. However, if the error is caused by a fault in the peripheral hardware, interacting with registers may not be enough to bring the peripheral to a consistent and safe state.

To handle peripherals that cannot be put into a consistent and safe state, a hardware device, the *peripheral gate* [51], is used that is able to force the REQ# signal from the peripheral to the bus arbiter to be high. Hence, the peripheral never receives the grant and it is prohibited from initiating any further transaction on the bus.[9] The peripheral gate is implemented based on a PCI extender card, i.e., a debug card that is interposed between the peripheral card and the bus and provides easy access to all signals. A clarifying picture for monitoring of a single peripheral is provided in Fig. 26(a). The monitoring device can output a *stop* signal, which closes the gate when active high; this can be achieved in a specification by setting the stop_reg described in Section 3.3.1 to '1'. Finally, sometimes the monitoring device cannot perform a suitable recovery action by itself, but there is a higher level actor, such as the OS or the system user, that can provide better recovery; examples include complex software operations such as restarting the driver or the whole PCI stack, and physically interacting with the peripheral. In this case, the best strategy is to communicate the failure to the chosen actor. Additionally, we implemented an RS-232 controller that can be used to send information to the user over a serial connection.

The reader should notice that the nature of our implementation is such that if a trace is seen, which does not conform

[9] While technically it is always possible for a faulty peripheral to disrupt the bus by altering the state of the signals, in practice the described approach is effective since access to the bus is mediated by three-state buffers enabled by GNT#.



(a) Gated Monitoring Device. (b) Block Diagram.

Fig. 26. Monitoring Device.

to a specification, as a consequence of a bus transaction, that specific bus transaction cannot be prevented from propagating to the rest of the system. For example, if a faulty peripheral performs a write transaction to an area in main memory which is not supposed to modify, we can detect the error, disconnect the peripheral, and report the failure to the OS/user. However, the information in the overwritten area will be lost. As part of our future work, we are working to implement an interposed monitoring device: by sitting between the bus and a peripheral, it will be able to buffer all transactions that target that specific peripheral or are initiated by it. If a property is validated/violated, it is then possible to take *preventive* measures (i.e., either discard or modify the transaction before propagating it). While this solution will provide a higher degree of reliability, there is a price to be paid in terms of increased communication delay due to buffering in the monitoring device.[10]

A simplified block diagram for the monitoring device is shown in Fig. 26(b). We distinguish three types of blocks: 1) blocks provided by Xilinx as proprietary intellectual properties (IPs); 2) manually coded RTL modules provided by BusMOP, which are independent of the peripheral specification; 3) automatically generated RTL modules, which are dependent on the specification (see Section 3.3.1 for specification syntax). PCI transaction signals are routed to two different modules: the PCI_core and the decode module.

5.2.1 The PCI_core Module

The PCI_core module is a hard IP[11] that implements all logic required to handle basic PCI functions such as plug-and-play. Bus slave and bus master logic is implemented by the slave and master modules, respectively. In particular, slave implements a set of 16 registers, base0 through base15. Since the OS configures the BAR registers at system boot, a peripheral cannot directly determine the location of address blocks used by another peripheral. Hence, the OS must also write the locations of the address blocks allocated to monitored peripherals in the base registers. The decode module is used to simplify event generation. It translates all transactions on the bus (except for those initiated by the monitoring device itself) into a series of I/O or memory reads/writes, one for each data phase, as well as the occurrence of an interrupt, and forwards the translated information to the monitoring logic.

5.2.2 The systemN Modules

The system0, . . ., systemN blocks implement the monitoring logic for each of N user specified properties. Each systemI block consists of two automatically generated modules: bus_interfaceI contains all logic that depends on the specific choice of communication interface (PCI bus), while monitorI contains all logic that depends on the formal language used to specify the property. This separation provides good modularity and facilitates code reuse. bus_interfaceI first receives as input the decoded bus signals and generates events, which are sequentialized by the events_sequentializer submodule (see Section 5.2.3), and then passed to monitorI using the seq_events wires. monitorI checks whenever the formula for the I-th property is validated/violated and passes the information back to bus_interfaceI, which can then execute three types of recovery: 1) disconnect a monitored peripheral from the bus using the stop signal; 2) send information to the user using the serial_output module, which implements a RS-232 transmitter; 3) start a write transaction on the bus using the master module. Finally, since it is possible for multiple systemI modules to initiate recovery at the same time, we provide queuing functions for serial_output and master in modules master_queue and serial_queue, respectively.

Notice that in the current implementation the time elapsed from any event that requires a handler to executing that corre-

[10] Note, however, that communication bandwidth will be unaffected.

[11] A hard IP is a module that is provided already synthesized rather than as an HDL or net list description (which are soft IPs). This means that it cannot be readily modified by users of the IP.

sponding handler is at most 4 clock cycles. This time is short enough to execute a recovery action before a faulty peripheral is allowed to start a new transaction, as PCI arbitration overhead prevents a peripheral from transmitting immediately.

5.2.3 The bus_interface Module

The code for the ⟨*BusMOP Declarations*⟩ (see Section 3.3.1), ⟨*BusMOP Action*⟩, and ⟨*BusMOP Handler*⟩ is copied verbatim into the VHDL module defining the bus_interface. The events are expanded to combinatorial statements implementing the specified logic. The output of the combinatorial statements is assigned to an events wire vector (not pictured), which is connected to the monitor module through an event_sequentializer submodule. Each index in the bus corresponds to the truth value of a specific event, numbered with the 0'th index as the first event, and the n'th index as the n'th event from top to bottom in the specification. This ordering is important, because it directs the event linearization performed by the event_sequentializer submodule.

The event_sequentializer is necessary because the logical formalisms expect linear, disjoint events. The event_sequentializer takes coincident events and sends them to the monitor in subsequent clock cycles, in ascending index order, using the seq_events wire vector (Fig. 26(b)). Therefore, if events(0) and events(3) occur in the same cycle, the monitor will see 0 followed by 3. To see why simultaneous events are possible, consider, again, Fig. 3 from Section 1. The cntrlMod event is asserted whenever the cntr_cntrl2 register (base1 + X"220") is written. Because both the countEnable and countDisable events require writes to the same address as the cntrlMod event, any time countEnable or countDisable are triggered, a cntrlMod is also triggered. As the property tries to enforce the policy that all modifications happen when the counter is not enabled, we must serialize events such that cntrlMod happens *after* a countDisable and *before* a countEnable. The ordering of events in Fig. 3, is consistent with this, because countDisable is listed before cntrlMod, which is listed before countDisable.

The @fail handler is placed in the module such that it is only executed if the monitor module denotes that the property has failed to match. The situation is similar for an @match handler, save that it is executed only when the formula or pattern is matched.[12] As can be seen in the Fig. 26(b), the monitor module reports the validation, violation, or neutral state of the monitored property, via the properties wire vector, to the bus_interface module. Several actions are available in ⟨*Property Handlers*⟩ as described under ⟨*BusMOP Handler*⟩ in Section 3.3.1. Aside from manipulating any local state of the monitor (such as the write to cntrlCurrent in Fig. 3), the bus_interface module makes available several registers which can be used used to execute the recovery actions detailed in Section 5.2. The registers are described under ⟨*BusMOP Handler*⟩ in Section 3.3.1.

[12] The same is true for @validation and @violation for PTLTL (Section 6.4).

5.2.4 The monitor Module

The monitor module is responsible for monitoring the property given serialized events. It encompasses the logic of the formula, and it is the only portion of our system dependent on the logical formalism used. The module is generated from the pseudocode monitor descriptions returned by the logic repository from Fig. 4. The parallel assignment algorithm for monitoring PTLTL described in Section 6.4 was originally designed for use in BusMOP monitors.

5.3 BusMOP Evaluation

In general, BusMOP imposes 0% runtime overhead on the system it monitors[13], therefore, our evaluation is a case study on the usefulness of BusMOP, first presented in [52]. We show how our runtime monitoring technique can be applied to a concrete case by providing specification and runtime experiments for a specific COTS peripheral, the PCI703A board [27]. PCI703A is a high performance Analog-to-Digital/Digital-to-Analog Conversion (ADC/DAC) peripheral for the PCI bus. In particular, it can perform high-speed, 14-bits precision ADC at a rate of up to 450,000 conversions/s, and transfer data to main memory in bus master mode. At the same time, the peripheral is simple enough that we were able to carefully check all provided hardware manuals and to manually inspect its Linux driver; specifying formal properties for a peripheral clearly requires a deep understanding of its inner working. In our proposed model, the peripheral's manufacturer is responsible for writing the runtime specification. In this sense, the formal specification can be thought of as a correctness certification provided by the manufacturer, as long as the user employs a monitoring device and recovery actions can be proved to restore the system to a safe state.

To better mimic what we think would be a typical process for a COTS manufacturer, we produced a requirement specification for the PCI703A in two steps. First, we prepared a detailed description of the communication behavior of the peripheral in plain English. Then, we converted this informal description into a formal set of events and formulae for BusMOP. Inspection of the driver revealed two software faults, both of which can cause errors that are detected and recovered by the monitoring device. While in this case we could have prevented errors by simply removing the faults, we argue that drivers for more complex peripherals can be thousands of lines long and neither code inspection nor testing is sufficient to remove all bugs. We further injected additional faults in the driver to test all written formal properties. It would have been nice to also show recovery for hardware faults, but we did not find any in the tested peripheral and injecting faults in the hardware is difficult. In what follows, we first provide an overview of PCI703A and then we detail properties for an example subsystem, a counter used in the ADC process. The

[13] Because the monitor is implemented on a peripheral card, overhead from the users perspective can only occur by increased bus traffic caused by specification handlers. Specification handlers typically run infrequently, and do not add much traffic to the PCI bus even when they do run.

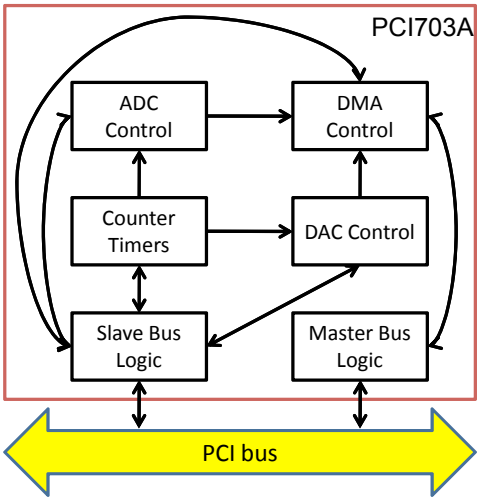


Fig. 27. PCI703A Diagram.

```
pci InterruptFix{
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event cntrlMod : memory write address in base1 + X"220"
                   {
                     cntrlOld <= cntrlCurrent;
                     cntrlCurrent <= value(15 downto 0);
                   }
  event setBit4 : memory write
                  address = base1 + X"220"
                  dbyte value(4) in '1'

  ere: setBit4

  @match {
    mem_reg <= '1';
    address_reg <= base1 + X"220";
    -- roll back to the previous cntr_cntrl2 value
    value_reg(15 downto 0) <= cntrlOld;
    cntrlCurrent <= cntrlOld;
    enable_reg <= "0011";
  }
}
```

Fig. 28. InterruptFix Specification

example is particularly instructive as we show how a small but representative set of properties is able to catch one of the aforementioned driver bugs.

A block diagram for the PCI703A is shown in Fig. 27. The bus slave logic implements two memory address blocks in BAR0 and BAR1, used for conversion data and control registers, respectively; the corresponding base addresses are written in base0 and base1 in the monitoring device. The ADC Control and DAC Control blocks control the ADC/DAC operations and write/read data into internal FIFOs. The DMA Control block can be programmed to move data between each FIFO and main memory using bus master functionality. Finally, the Counter Timers block implements four counters. Counter 0 and 1 are user programmable and can be used either for debugging purposes or to trigger a DA conversion. Counter 3 is also user programmable and produces an external output. Finally, Counter 2 is not meant to be user programmable; it is to be used exclusively to generate the clock for AD conversions. The C user library provided with the driver exports an ADConfig function used to configure ADC Control and the associated Counter 2. The library also provides a CTConfig function to be used to configure the user counters; unfortunately, under Linux the function can also be used to change the configuration of Counter 2. This is a problem, as any user in the system could erroneously or maliciously change Counter 2 while an ADC is in progress.

Three 16-bit control registers are relevant to our discussion: cntr_cntrl2 (at hexadecimal location 220 relative to BAR1), cntr_divr2 (228), and adc_cntrl (300). Bit 0 of cntr_cntrl2 determines whether Counter 2 is enabled, and bits 2-1 determine its clock source (either 20Mhz or 100Khz); when the counter is enabled, it first loads the content of cntr_divr2 and then starts counting down at the selected frequency. When it reaches zero, the value of cntr_divr2 is reloaded, a clock signal is sent to ADC Control, and finally if bit 4 of cntr_cntrl2 is set, an interrupt is generated. Register adc_cntrl controls the behavior of ADC Control; in particular, bit 0 enables/disables the ADC process and bits 2-1 determine the clock source, with a value of "00" indicating that Counter 2 is used.

We express three requirements:

Requirement 1 *Bit 4 of cntr_cntrl2 should never be set. While the functionality is relevant for Counters 0,1, in the case of Counter 2 setting bit 4 would cause the generation of spurious interrupts that increase load on the driver.*

Requirement 2 *If the ADC is using Counter 2, and the clock source for Counter 2 is set to 20 Mhz, then the value of cntr_divr2 must be at least 45 to avoid violating the maximum conversion speed of the peripheral.*

Requirement 3 *If the ADC is active and using Counter 2, then Counter 2 must also be active; furthermore, while Counter 2 is active no change to the counter configuration is allowed.*

Requirements 1-3 are able to catch the driver bug in the sense that an invalid counter configuration cannot be set before starting the ADC, and furthermore while the ADC is active no counter modification is allowed. We wrote four (five including the example from Section 1) formal properties to capture the requirements:

InterruptFix. The InterruptFix specification is the formalization of Requirement 1, and can be seen in Fig. 28. Because we do not want the 4th bit set, we simply monitor the pattern setBit4, an event which corresponds to setting the 4th bit. We perform recovery when the pattern is validated by overwriting cntr_cntrl2 with the last valid value, similarly to SafeCounterModify in Fig. 3.

SafeConversionSpeed. The SafeConversionSpeed specification is the formalization of Requirement 2, and can be seen in Fig. 29. For this property we chose to show how event side effects can be used in handlers as part of checking that a property has been validated/violated. When the clkSrcSet or srcSet events are triggered, meaning that the cntr_cntrl2

```
pci SafeConversionSpeed{
  signal clkSrc : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal src : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event divrBad: memory write address = base1 + X"228"
                 dbyte value in 0,44
  event divrGood: memory write address = base1 + X"228"
                  dbyte value in 45,65535
  event clkSrcSet : memory write address in base1 + X"300"
                    { clkSrc <= value(15 downto 0); }
  event srcSet :  memory write address in base1 + X"220"
                  { src <= value(15 downto 0); }
  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'

  ere : (divrBad (clkSrcSet + srcSet)* countEnable)*

  @match {
    if (clkSrc(2 downto 1) = "01") and (src(2 downto 1) = "00") then
      mem_reg <= '1';
      address_reg <= base1 + X"228";
      --set cntr_divr2 to 45
      value_reg(15 downto 0) <= X"2D";
      enable_reg <= "0011";
    end if;
  }
}
```

Fig. 29. SafeConversionSpeed Specification

or adc_cntrl registers have been modified, respectively, we store the value written to the register in monitor local registers (e.g., src <= value(15 downto 0)). The pattern specifies that the cntr_divr2 be set to a bad value (less than 45), followed by any number of updates to cntr_cntrl2 or adc_cntrl, followed by the enabling of the counter. If cntr_divr2 is set to a value larger than 44, the pattern will be violated, and the monitor will be reset. This means that the validation handler will be executed only when the value of cntr_divr2 is too low for safe conversion, but regardless of whether or not the board is actually using Counter 2. The handler then checks that it is, in fact using Counter 2, and that Counter 2 is using the 20Mhz source, before performing the recovery: setting cntr_divr2 to a valid value (45).

NoDisableWhileConverting. The NoDisableWhileConverting specification is the formalization of part of Requirement 3, and can be seen in Fig. 30. This could have been written in a similar manner to SafeConversionSpeed, i.e., using event side effects to store current register values and checking them in the handler. We decided to use a fully formal specification, that defines events for setting the registers to good or bad values. The formula itself specifies that, if the ADC is enabled, and clkSrc2 is good, meaning that Counter 2 is being used to time the ADC, then Counter 2 must be enabled. The part of the formula before the implies keyword, states that the ADC is enabled and the ADC clock source is Counter 2, the second half of the formula is the requirement that Counter 2 not be disabled. The formula is true when correct behavior is exhibited, so we use a violation handler for the recovery action, which again is simply to set cntr_cntrl2 to the last valid value.

SafeDivrModify. The SafeDivrModify specification is the formalization of part of Requirement 3, and can be seen in Fig. 31. In conjunction with NoDisableWhileConverting and SafeCounterModify (from Section 1), all of requirement 3 is covered.

```
pci NoDisableWhileConverting{
  signal cntrlCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal cntrlOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'
                      {
                        cntrlOld <= cntrlCurrent;
                        cntrlCurrent <= value(15 downto 0);
                      }
  event countDisable : memory write address = base1 + X"220"
                       dbyte value(0) in '0'
                       {
                         cntrlOld <= cntrlCurrent;
                         cntrlCurrent <= value(15 downto 0);
                       }
  event clkSrc2Good : memory write address = base1 + X"300"
                      dbyte value(2 downto 1) in "01"
  event clkSrc2Bad : memory write address = base1 + X"300"
                     dbyte value(2 downto 1) not in "01"
  event adcEnable : memory write address = base1 + X"300"
                    dbyte value(0) in '1'
  event adcDisable : memory write address = base1 + X"300"
                     dbyte value(0) in '0'

  ptltl :  ( ((not adcDisable) S adcEnable) and
             ((not clkSrc2Bad) S clkSrc2Good) )
           implies
           ((not countDisable) S countEnable)

  @violation {
    mem_reg <= '1';
    address_reg <= base1 + X"220";
    -- roll back to the previous cntr_cntrl2 value
    value_reg(15 downto 0) <= cntrlOld;
    cntrlCurrent <= cntrlOld;
    enable_reg <= "0011";
  }
}
```

Fig. 30. NoDisableWhileConverting Specification

```
pci SafeDivrModify{
  signal divrCurrent : STD_LOGIC_VECTOR(15 downto 0) := X"0000";
  signal divrOld : STD_LOGIC_VECTOR(15 downto 0) := X"0000";

  event countDisable : memory write address = base1 + X"220"
                       dbyte value(0) in '0'
  event divrMod : memory write address in base1 + X"228"
                  {
                    divrOld <= divrCurrent;
                    divrCurrent <= value(15 downto 0);
                  }
  event countEnable : memory write address = base1 + X"220"
                      dbyte value(0) in '1'

  ptltl: (divrMod) and (*)((not countDisable) S countEnable)

  @validation {
    mem_reg <= '1';
    address_reg <= base1 + X"228";
    -- roll back to the previous cntr_divr2 value
    value_reg(15 downto 0) <= divrOld;
    divrCurrent <= divrOld;
    enable_reg <= "0011";
  }
}
```

Fig. 31. SafeDivrModify Specification

This specification ensures that cntr_divr2 is not modified while Counter 2 is enabled. This property is the same as SafeCounterModify from Fig. 3, save that we are ensuring that cntr_divr2 is not modified, rather than cntr_cntrl2. We also used PTLTL rather than ERE, to show how two very similar properties look in both logics. These could be collapsed into one specification, but it would make recovery more complicated, because we

only want to roll back the register that was actually modified (cntr_cntrl2 or cntr_divr2). The formula itself states that if cntr_divr2 has been modified and the counter has not been disabled since the last time it was enabled, than we must recover. Unlike SafeCounterModify we use a validation rather than a violation handler, because the formula was easier to express with recovery being on validation.

As a final consideration, note that the handlers of SafeCounterModify, InterruptFix and NoDisableWhileConverting can be invoked simultaneously if an incorrect value is written to cntr_cntrl2, which results in the execution of multiple bus writes. However, this causes no problem since all handlers overwrite cntr_cntrl2 with the same valid value.

6 Logic Plugins

Each logical formalism provided by the MOP framework is implemented as a program called a *logic plugin*, as mentioned in Section 2. The individual logic plugins are controlled by the logic repository as can be seen in Fig. 4.

For each plugin we first discuss the syntax of specifications using the formalism. Each plugin syntax instantiates the generic ⟨*Logic Name*⟩, ⟨*Logic Syntax*⟩, ⟨*Logic State*⟩ non-terminals described in Section 3.1 (see also Fig. 6). Internally, aside from the specified syntax, each plugin is also given the set of events used in the property by the instance client (either JavaMOP or BusMOP). The instance client simply drops the instance-specific event definition and actions, which are irrelevant to the plugin, and sends only the event names. For simplicity, in what follows, the event declarations are dropped from the logic plugin's syntax. To clarify the syntax we show an example property in each logic, which also does not include event descriptions. Because only the property is shown, without any event definitions, the parameters, which are part of event definitions in JavaMOP, are absent. One should be aware, if one wishes to use the logic repository described in Section 2 without using one of the two pre-defined instance clients, that one should also provide the event names with the property.[14]

We then discuss some issues specific to the particular plugin as well as how monitor pseudocode suitable for conversion to Java and HDL is generated. The pseudocode generated from each example property is shown to make the explanation of monitor pseudocode generation more concrete.

For every plugin we also describe how enable sets are generated. Recall that enable sets distill information about which prior events must be seen for a given event to create a new monitor instance (see Defs. 10, 11, 12 and the surrounding text in Section 4.2 for a complete explanation of enable sets).

6.1 Finite State Machines

The finite state machine (FSM) plugin is one of the most important plugins for MOP. Not only is it a useful logical formalism in itself, but it is used as a backend for all logics reducible to finite automata. Currently, all logic plugins except for the context-free grammar (CFG; Section 6.5) and past time linear temporal logic with calls and returns (PTCaRet; Section 6.6) generate FSM output. This allows for a strong separation of concerns. For instance, minimization need occur only once, and it allows us to use one enable set generation algorithm for all of these plugins. Additionally, each instance of MOP need only know how to translate the pseudocode for FSMs, CFGs, and PTCaRet. Some MOP instances, such as BusMOP, may even opt to support only finite state monitors, in which case they only need to provide support for translating FSM pseudocode.[15]

⟨*FSM Name*⟩	::=	"*fsm*"
⟨*FSM Syntax*⟩	::=	{⟨*Item*⟩}{⟨*Alias*⟩}
⟨*Item*⟩	::=	⟨*State Name*⟩ "[" { ⟨*Transition*⟩ [","]} "]"
⟨*Transition*⟩	::=	⟨*Event Name*⟩ "–>" ⟨*State Name*⟩
	|	"*default*" ⟨*State Name*⟩
⟨*Alias*⟩	::=	"*alias*" ⟨*Group Name*⟩ "="
		{ ⟨*State Name*⟩ "," } ⟨*State Name*⟩
⟨*FSM State*⟩	::=	⟨*Group Name*⟩ | ⟨*State Name*⟩ | "*fail*"

Fig. 32. FSM Syntax

```
start [
  default start
  next -> unsafe
  hasnext -> safe
]
safe [
  next -> start
  hasnext -> safe
  dummy -> safe
]
unsafe [
  next -> unsafe
  hasnext -> safe
]
alias all_states = start, safe, unsafe
alias safe_states = start, safe
```

Fig. 33. FSM Example

Fig. 32 shows the syntax for FSM properties. An FSM property is a series of ⟨*Item*⟩s followed by ⟨*Alias*⟩s. An ⟨*Item*⟩ is essentially a state in the finite state machine, and the different transitions to take on a given input (⟨*Transition*⟩). The ⟨*Alias*⟩ allows for giving a name to a *set* of states. This is invaluable, because the ⟨*FSM State*⟩ non-terminal, which defines what categories may trigger handlers, both ⟨*Group Name*⟩s, which are the names associated to sets of states in ⟨*Alias*⟩s, and ⟨*State Name*⟩s may be associated with handlers. This allows one to write a property that triggers actions when any state in a given ⟨*Alias*⟩ is entered.

Fig. 33 shows an example FSM property. In this example, three events: next, hasnext and dummy, and three states: start, safe and unsafe are defined. Two state aliases are declared:

[14] Additionally, one must use the logic repository XML syntax, which distinguishes the events from the property.

[15] Though note that in the case of BusMOP, finite state machines are not used for PTLTL in favor of using parallel assignments (see Section 6.4).

all_states represents all the states in the state machine and safe_states includes the start state and the safe state. The fail category is reported whenever an event occurs that is not specified for the current state. For example, the state machine will go into fail when the dummy event is seen in the unsafe state. The default transition in the start state covers any event not specified in the transition. Because of this, any state with a default transition cannot lead to a fail category for any input. As mentioned in Fig. 32, handlers may be associated with any state (e.g., start) or group name (e.g., all_states).

In the interest of keeping runtime monitoring as efficient as possible, we wish to use minimized finite state machines for monitors. Because of the ability to trigger handlers from ⟨*State Name*⟩s and ⟨*Group Name*⟩s, MOP FSM properties are *multicategory* finite state machines (finite state machines that recognize more than one language; essentially equivalent to Moore machines). This requires a small change to the normal Hopcroft FSM minimization algorithm [39].

The Hopcroft algorithm works by assuming the largest possible equivalence class of states, and then partitioning the equivalence classes into smaller classes if necessary. The way the algorithm determines that it is necessary to split is by considering two equivalence classes C_1 and C_2 and an input, e. For each state s in C_1, if s goes to a state in C_2 on e then it goes into class C_{11}, otherwise it goes to class C_{12}. Classes are continuously split by other classes until a fixed point is reached. When a fixed point is reached, each equivalence class becomes a state in the final machine.

The way our algorithm differs is in the initial partition. The normal algorithm partitions the states into two classes, those states that are final states and those which are not. We, however, have multiple categories. The particularly interesting feature, is that categories may overlap on states. If two categories C_1 and C_2 overlap, they must have three equivalence classes: those states in $C_1 - C_2$, those in $C_2 - C_1$, and those in $C_1 \cap C_2$. The naive algorithm would be to compute the intersections between all the categories, but that is quadratic in nature. A better algorithm, which we use, is to find the set of categories each state belongs to. This takes time linear in the number of states. Those states that have the same set of categories are placed in the same initial equivalence class.

The monitor pseudocode for an FSM property appears the same as the input code, except that it will be minimized, whereas the input code may not be minimal. Because of this, we omit the output of the example in Fig. 33. Each MOP instance that wants to support the finite state machine-based logics must convert the FSM pseudocode into executable code.

The algorithm in Fig. 34 computes the property enable sets for a finite state machine [21]. We use this algorithm to compute the enable sets for any logic that is reducible to a finite state machine, including ERE, LTL, and PTLTL. The algorithm assumes a finite state machine, defined as $FSM = (\mathcal{E}, S, s_0 \in S, \delta : S \times \mathcal{E} \rightharpoonup S)$. $\mathcal{E}$ is the alphabet, traditionally listed as Σ but changed for consistency, because the alphabets of our FSMs are event sets. s_0 is the start state, corresponding to 1 in the definition of a monitor. δ is the transition partial function, taking a state and an event and potentially mapping to a next state for the machine. Note that we can extend δ to be consistent with σ in Def. 7 by simply completing the function by adding an undefined state, *undef*, and making all non-existent transitions point to *undef*. This is how FSMs are handled in the JavaMOP instance. We assume that all states not reachable from the initial state and not coreachable from the states of interest (states of interest being those states s such that $\gamma(s) \in \mathcal{G}$, where $\mathcal{G}$ is the goal category; see Def. 11) are pruned from the FSM before running the algorithm, leaving the transitions that pointed to them undefined. $\mathcal{V}_\mu$ is a mapping from states to sets of events; it is used to check for algorithm termination. $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ is the output property enable set, which is converted into a parameter enable set by the language instance client, discussed in Section 2.

```
Algorithm EN_fsm(FSM = (E, S, s_0, δ))
Globals: mapping V_μ : S → P_f(P_f(E))
         mapping enable^E_G : E → P_f(P_f(E))
Initialization: V_μ(s) ← ∅ for any s ∈ s
                enable^E_G(e) ← ∅ for any e ∈ E
function main()
 1 compute_enables(s_0, ∅)
function compute_enables(s, μ)
 1 foreach defined δ(s, e) do
 2 : enable^E_G(e) ← enable^E_G(e) ∪ {μ}
 3 : let μ' ← μ ∪ {e}
 4 : if μ' ∉ V_μ(s)
 5 : : V_μ(s) ← V_μ(s) ∪ {μ'}
 6 : : compute_enables(δ(s, e), μ')
 7 : endif
 8 endfor
```

Fig. 34. FSM $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ Computation Algorithm [21].

Function compute_enables is first called from main with $\mu = \emptyset$ and the initial state s_0. If we think of the FSM as a graph, μ represents the set of edges we have seen at least once in a given traversal path. For each defined $\delta(s, e)$ (line 1), we add the current μ to the $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e)$ (line 2) because this means we have seen a viable prefix set (as all non-viable paths in the machine have been pruned). This follows from the definition of $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$. Line 3 begins the recursive step of the algorithm. We let $\mu' = \mu \cup \{e\}$, because we have traversed another edge, and that edge is labeled as e. The map $\mathcal{V}_\mu$ tells us which μ have been seen in previous recursive steps, in a given state. If a μ has been seen before, in a state, taking a recursive step can add no new information. Because of this, line 4 ensures that we only call the recursive step on line 6, if new information can be added. Line 5 keeps $\mathcal{V}$ consistent. Thus the algorithm terminates only when every viable μ has been seen in every reachable state, effectively computing a fixed point. Thus, the algorithm is bounded by the number of one cycle paths through the graph (and is faster in practice, because most paths will have repeated events).

⟨*ERE Name*⟩	::=	"*ere*"
⟨*ERE Syntax*⟩	::=	"*empty*" | "*epsilon*"
	|	⟨*Event Name*⟩
	|	⟨*ERE Syntax*⟩ "*"
	|	⟨*ERE Syntax*⟩ "+"
	|	"~" ⟨*ERE Syntax*⟩
	|	⟨*ERE Syntax*⟩ "|" ⟨*ERE Syntax*⟩
	|	⟨*ERE Syntax*⟩ "&" ⟨*ERE Syntax*⟩
	|	⟨*ERE Syntax*⟩ ⟨*ERE Syntax*⟩
	|	"(" ⟨*ERE Syntax*⟩ ")"
⟨*ERE State*⟩	::=	"*match*" | "*fail* | "*?*"

Fig. 35. ERE Syntax

$$
\begin{aligned}
(R_1|R_2)\{e\} &= R_1\{e\}|R_2\{e\} \\
(R_1 R_2)\{e\} &= (R_1\{e\})\ R_2 \\
&\quad |\ \text{if}(\mathit{epsilon}\ \text{in}\ R_1)\ \text{then}\ R_2\{e\}\ \text{else}\ \mathit{empty} \\
R*\{e\} &= (R\{e\})R* \\
\tilde{}R\{e\} &= \tilde{}(R\{e\}) \\
e_1\{e_2\} &= \text{if}(e_1 = e_2)\ \text{then}\ \mathit{epsilon}\ \text{else}\ \mathit{empty} \\
\mathit{epsilon}\{e\} &= \mathit{empty} \\
\mathit{empty}\{e\} &= \mathit{empty}
\end{aligned}
$$

Fig. 36. ERE Derivative Equations

```
s0[
  create_coll -> s1
]
s2[
  use_iter -> s4
  update_map -> s3
]
s1[
  createIter -> s2
  update_map -> s1
]
s4[
  use_iter -> s4
  update_map -> s3
]
s3[
  use_iter -> s5
  update_map -> s3
]
s5[
]
alias match = s4, s5
```

Fig. 37. ERE Example Output

6.2 Extended Regular Expressions

Regular expressions can be easily understood by the average software engineer or programmer, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for monitoring requirements, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must not occur during an execution. Complementation gives one the power to express patterns on strings non-elementarily more compactly. Also, one important observation about the use of ERE in the context of runtime verification is that ERE patterns are often used to describe buggy patterns instead of desired properties.

Fig. 35 shows the syntax for ERE properties. The operators are standard for regular expressions, except that "~" is the language complement of an ERE, and "&" is language intersection. While "*epsilon*" is the empty string, as is normal, "*empty*" refers to the empty language.

Here is an example ERE property for the UnsafeMapIterator property previously shown in Fig. 11:

```
create_coll update_map* create_iter
use_iter* update_map+ use_iter
```

Recall that in this property the sequence of actions of importance is the creation of an Iterator from a Collection that was created from a Map, which is updated between the creation of the Iterator and its use.

FSMs are generated from EREs using coinductive techniques [57]. 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 e produce another extended regular expression, denoted $R\{e\}$, with the property that for any trace w, trace $e\ w$ is in $\mathcal{L}(R)$ (i.e., the language denoted by R) iff w is in $\mathcal{L}(R\{e\})$. Fig. 36 defines this derivative semantics recursively on the structure of regular expressions; the operators without equations can be defined in terms of operators that do have equations specified. In the equations "|" refers to the ERE operator "|". The generated FSM is not minimal; the minimization algorithm of the FSM plugin (Section 6.1) is used to make it minimal. A deterministic automaton is produced, saving memory and time (in contrast to the more conventional Thompson approach [60], which operates by first producing a non-deterministic automaton, and then using a determinization algorithm to produce a deterministic automaton).

Fig. 37 shows the output FSM for the UnsafeMapIterator example ERE above. Note that the alias match is assigned so that ERE properties properly allow match as a verdict category. This was the initial motivation for aliases in the FSM plugin. Also note that the states do not have fully specified input, so fail is a possible output category.

As mentioned in Section 6.1, the property enable sets for EREs are computed by using the algorithm in Fig. 34 after the ERE has been converted to an FSM.

6.3 (Future Time) Linear Temporal Logic

Linear temporal logic (LTL) [54] is often used to specify properties in model checking. LTL formulae allow one to express concepts such as the occurrence of an event requiring

⟨*LTL Name*⟩	::=	"*ltl*"
⟨*LTL Syntax*⟩	::=	"*true*" | "*false*"
	|	⟨*Event Name*⟩
	|	"*not*" ⟨*LTL Syntax*⟩
	|	⟨*LTL Syntax*⟩"*and*" ⟨*LTL Syntax*⟩
	|	⟨*LTL Syntax*⟩"*or*" ⟨*LTL Syntax*⟩
	|	⟨*LTL Syntax*⟩"*xor*" ⟨*LTL Syntax*⟩
	|	⟨*LTL Syntax*⟩"*implies*" ⟨*LTL Syntax*⟩
	|	"[]" ⟨*LTL Syntax*⟩
	|	"<>" ⟨*LTL Syntax*⟩
	|	"o" ⟨*LTL Syntax*⟩
	|	⟨*LTL Syntax*⟩ "U" ⟨*LTL Syntax*⟩
	|	⟨*LTL Syntax*⟩ "R" ⟨*LTL Syntax*⟩
⟨*LTL State*⟩	::=	"*validation*" | "*violation*" | "*?*"

Fig. 38. LTL Syntax

$(F_1\ and\ F_2)\{e\}$	=	$F_1\{e\}\ and\ F_2\{e\}$
$(F_1\ or\ F_2)\{e\}$	=	$F_1\{e\}\ or\ F_2\{e\}$
$(F_1\ xor\ F_2)\{e\}$	=	$F_1\{e\}\ xor\ F_2\{e\}$
$not\ F\{e\}$	=	$not\ (F\{e\})$
$(F_1\ \mathrm{U}\ F_2)\{e\}$	=	$F_2\{e\}\ or\ F_1\{e\}\ and\ F_1\ \mathrm{U}\ F_2$
$(F_1\ \mathrm{R}\ F_2)\{e\}$	=	$F_2\{e\}\ and\ (F_1\{e\}\ or\ F_1\ \mathrm{R}\ F_2)$
$\mathrm{o}\ F\{e\}$	=	F
$e_1\{e_2\}$	=	if$(e_1 = e_2)$ then $true$ else $false$
$true\{e\}$	=	$true$
$false\{e\}$	=	$false$

Fig. 39. LTL Derivative Equations

```
n0[
  end -> validation
  grant -> n0
  request -> n1
]
n1[
  end -> violation
  grant -> n0
  request -> violation
]
validation[default: validation]
violation[default: violation]
```

Fig. 40. LTL Example Output

that another event happen in the future. Note that runtime monitoring cannot guarantee the correctness of a safety or liveness property. Even though the properties might hold for a given execution of the system, they can only be proved to hold, in general, by exploring every possible state of the program. LTL specifications must be used with this in mind.

Fig. 38 shows the complete syntax for LTL supported by our plugin. The operators "*not*", "*and*", "*or*" , and "*implies*" are the standard boolean connectives. The operator "[]" stands for "always", meaning the formula following it must hold at all times, while "<>" stands for "eventually", meaning that the formula following it must eventually hold in the future. The operator "o" means "next": the formula following it, say F, must hold in the next time step; in terms of MOP, this means that F must hold when the next event occurs. The operators "U" and "R" are duals of each other. "U" is "until": "F_1 U F_2" means "either F_2 must hold now, or F_1 must hold until F_2 eventually holds". "<> F" can be defined as "*true* U F". The operator "R" means "release". "F_1 R F_2" means "once F_1 holds, F_2 can be released in the next time step;" F_2 must hold at all periods before F_1 holds, and it must hold during the first time step in which F_1 holds. "[] F" can be defined in terms of "R" as "*false* R F".

Below is an example LTL property, which states that all requests of a resource must be immediately fulfilled (grant) until the end of a program:

```
(request implies o grant) U end
```

The property does not hold until end occurs, and it must be that any request is fulfilled until such a time as end occurs.

FSMs are generated from LTL formulae in much the same way as from EREs, following a technique described in [33, 55]. Derivations based on event consumption are again used. Fig. 39 shows the equations used to derive FSMs. Monitors generated by our current LTL plugin report violation when a state is reached where there is no way to reach validation, and vice versa. To handle this, we use an LTL satisfiability checker on each derived formula. Not only does this allow us to collapse unsatisfiable and tautological states, but it allows us to perform minimization on the fly (by comparing states to each other using "iff" in the SAT solver). This minimization could be handled by the FSM plugin, but since we must check for unsatisfiable and tautological states to correctly implement the LTL monitoring algorithm, we perform the minimization on the fly.

Fig. 40 shows the output FSM for the example given above. Note that the states validation and violation are named so that LTL properties properly allow validation and violation as output categories. Note that the validation and violation states cannot be left on any input.

As mentioned in Section 6.1, the property enable sets for LTL formulae are computed by using the algorithm in Fig. 34 after a formula has been converted to an FSM.

6.4 Past Time Linear Temporal Logic

Past time linear temporal logic is similar to LTL, except that all operators refer to the past. Some safety properties are more easily expressed in terms of the past than the future, for example the property that a user authentication be required before accessing some resource is most naturally expressed as "access *implies* <*> authenticate", e.g., that an access requires an authentication at some point in the past. Monitors generated from PTLTL formulae also have the quality of validating or violating on every event because the past is already known. This contrasts with LTL monitors, which can also be in an intermediate, ?, state. Additionally, once an LTL monitor validates or violates, it is always violated or validated, whereas PTLTL is allowed to change on each event. The earlier caveat

⟨*PTLTL Name*⟩	::=	"*ptltl*"
⟨*PTLTL Syntax*⟩	::=	"*true*" | "*false*"
	|	⟨*Event Name*⟩
	|	"*not*" ⟨*PTLTL Syntax*⟩
	|	⟨*PTLTL Syntax*⟩"*and*" ⟨*PTLTL Syntax*⟩
	|	⟨*PTLTL Syntax*⟩"*or*" ⟨*PTLTL Syntax*⟩
	|	⟨*PTLTL Syntax*⟩"*xor*" ⟨*PTLTL Syntax*⟩
	|	⟨*PTLTL Syntax*⟩"*implies*" ⟨*PTLTL Syntax*⟩
	|	"[*]" ⟨*PTLTL Syntax*⟩
	|	"<∗>" ⟨*PTLTL Syntax*⟩
	|	"(*)" ⟨*PTLTL Syntax*⟩
	|	⟨*PTLTL Syntax*⟩ "S" ⟨*PTLTL Syntax*⟩
⟨*PTLTL State*⟩	::=	"*validation*" | "*violation*"

Fig. 41. PTLTL Syntax

Formula	Assignment	Initial $b[n]$
$F_1\ S\ F_2$	$b[n] \leftarrow B(F_1)$ and $b[n]$ or $B(F_2)$	false
[*] F	$b[n] \leftarrow B(F)$ and $b[n]$	true
<∗> F	$b[n] \leftarrow B(F)$ or $b[n]$	false
(*) F	$b[n] \leftarrow B(F)$	false
Formula	Expression	
$B(F)$	F if F is a simple boolean formula, otherwise the $b[n]$ storing the value of F	

Fig. 42. PTLTL Assignment Equations

of LTL not guaranteeing that its formula holds for all program executions applies to PTLTL, as well.

Fig. 41 shows the syntax for our PTLTL plugin. The operators "*not*", "*and*", "*or*" , and "*implies*" are the standard boolean connectives. The operator "[*]" stands for "always in the past", meaning the formula following it must hold at all times in the past, while "<∗>" stands for "eventually in the past", meaning that the formula following it must either currently hold or it must have held somewhere previously in the trace. The operator "(*)" means "previously": the formula following it, say F, must hold in the previous time step; in terms of MOP, this means that F must have held when the previous event occurred. "S" means "since"; "F_1 S F_2" means "either F_2 must hold now, or F_2 must have held in the past and F_1 must have held since then."; "<∗> F" can be defined as "*true* S F."

Below is an example PTLTL property. In this property the goal is to ensure that next is never called on an Iterator without first calling hasNext:

```
next implies (*)hasNext
```

The event definition should make sure that the call to the hasNext method actually returns true, as well. Recall that (*) means previously, so the property states that the event preceding next must be hasNext.

The original algorithm for PTLTL monitor generation, as outlined in [36,37], works by using a bitvector to keep the state of each temporal operator in the formula. A series of sequential assignments updates the bitvector as each event arrives. For example, "*hold* S *acquire*" would need one bitvector index to monitor. The assignment for this bitvector index would be "`b[0] ← b[0] and hold or acquire`". Fig. 42 shows the assignments necessary for each PTLTL temporal operator. Note that if one of the operands to a temporal formula is itself a temporal formula, it will appear as a bitvector index in the assignment. It is, then, essential to generate the assignments in the proper order (depth-first).

In [52], it was determined that a parallel series of assignments would be more efficient for monitoring PTLTL properties on an FPGA. Sequential assignments are parallelized by back substitution of terms for the bitvector index they computed. This back substitution in an assignment to `b[n]` is only performed, however, for bitvector indices `b[m]` that are computed before the assignment to `b[n]` in the original sequential assignments. For example, consider the following sequential bitvector assignments:

```
b[0]  ←  b[0] or e1
b[1]  ←  b[0] and b[2] or e2
b[2]  ←  e3
```

When parallelized (we us $\leftharpoondown$ to denote parallel assignments and ← for sequential), the code becomes:

```
b[0]  ⇁  b[0] or e1
b[1]  ⇁  (b[0] or e1) and b[2] or e2
b[2]  ⇁  e3
```

Note how "`b[0] or e1`" was substituted for "`b[0]`" in the assignment to `b[1]` because the assignment to "`b[0]`" occurred before the assignment to `b[1]` in the sequential code, while the assignment to `b[2]` was *not* substituted, because `b[2]` was computed after `b[1]` in the sequential code.

By using the parallel assignments it also becomes straightforward to generate an FSM by exhaustively computing and enumerating the reachable bitvectors. This allows us to easily compute the property enable sets of a PTLTL formula by using the algorithm in Fig. 34 on the FSM generated from the formula. This is the strategy now used in JavaMOP, while BusMOP continues to use the parallel assignments.

Figs. 43(a) and 43(b) show the monitor pseudocode for the example above. Fig. 43(a) shows the parallel assignment format (which, in this case, is equivalent to the sequential code), while Fig. 43(b) shows the FSM output. Note that in the parallel assignments `b[0]` is initialized to false. In the parallel assignments the output statement outputs the actual category. If it evaluates to false a violation is reported, if it evaluates to true a validation is reported. The FSM output uses aliases for validation and violation because, unlike in LTL, multiple states can be validation or violation states due to the manner in which the FSM is generated (e.g., n1 and n2 are both in the validation alias).

As mentioned in Section 6.1, the enable sets for PTLTL are computed by generating an FSM and using the FSM enable set computation algorithm.

6.5 Context-Free Grammars

Context-free grammars (CFG) are nearly as widely adopted by the average programmer as are regular expressions. Numerous

```
b[0] ← hasNext
output(not next or b[0])
```

(a) Parallel Assignments

```
n0[
  default n2
  hasnext -> n1
  next -> n0
]
n1[
  default n2
  hasnext -> n1
  next -> n2
]
n2[
  default n2
  hasnext -> n1
  next -> n0
]
alias violation = n0
alias validation = n1,n2
```

(b) FSM

Fig. 43. PTLTL Example Output

⟨*CFG Name*⟩	::=	"*cfg*"
⟨*CFG Syntax*⟩	::=	⟨*Rule*⟩ {"," ⟨*Rule*⟩}
⟨*Rule*⟩	::=	⟨*Symbol*⟩ "–> " ⟨*Symbols*⟩ {"|" ⟨*Symbols*⟩}
⟨*Symbols*⟩	::=	⟨*Symbols*⟩ {⟨*Symbols*⟩}
	|	⟨*Event Name*⟩
	|	⟨*Non-terminal Name*⟩
⟨*Non-terminal Name*⟩	::=	⟨*Letter or "_"*⟩ {⟨*Letter, digit, or "_"*⟩}
⟨*CFG State*⟩	::=	"*match*" | "*fail*" | "*?*"

Fig. 44. CFG Syntax

context-free parser generators such as Bison [14] exist and are widely used. CFGs offer a level of expressibility greater than that of finite-monitor logics, and allow for the specification of properties that involve proper nesting and a notion of counting.

Fig. 44 shows the syntax for our CFG plugin, where "," separates rules and "|" multiple alternatives within one rule. The "–>" separates the non-terminal head of a production from its body. While a CFG can contain epsilon, the language it represents will have the empty string removed from it.

Below is an example CFG property.

```
S -> P endThread,
P -> P acquire P release | epsilon
```

In this example there are three events: acquire, release, and endThread, and one non-terminal, S. The fail category is reported whenever the program cannot have followed the proper lock nesting property, e.g., releasing a lock more times than it was acquired or not releasing it enough times before the end of the Thread. Additionally, we could handle match, which is reported whenever our locking discipline is faithfully completed.

The CFG plugin uses a generalized LR (GLR) parser. Specifically, when an ambiguity is encountered, instead of choosing one of the particular alternatives, all are tried in parallel. The parser accepts a string if any of its parallel parses

	Action				Goto	
	$	endThread	acquire	release	P	S
0	error	shift(11)	shift(8)	error	5	10
1	error	reduce(P,2)	reduce(P,2)	reduce(P,2)	error	error
2	error	reduce(P,3)	reduce(P,3)	reduce(P,3)	error	error
3	error	reduce(P,3)	reduce(P,3)	reduce(P,3)	error	error
4	error	reduce(P,4)	reduce(P,4)	reduce(P,4)	error	error
5	error	shift(12)	shift(9)	error	error	error
6	error	error	shift(9)	shift(2)	error	error
7	error	error	shift(9)	shift(4)	error	error
8	error	error	shift(8)	shift(1)	6	error
9	error	error	shift(8)	shift(3)	7	error
10	accept	error	error	error	error	error
11	reduce(S,1)	error	error	error	error	error
12	reduce(S,2)	error	error	error	error	error

Fig. 45. CFG Example Output

does and rejects it if there is no possible parse in any of the alternate parses. Thus, unlike normal LR(1) parsers, the GLR algorithm is capable of recognizing all context-free languages. This parsing algorithm is both online and the overhead relative to a normal LR parser is proportional to the amount of ambiguity in the grammar. The LR parser tables are generated using Knuth's LR(1) parser table generation algorithm as presented in [3]. The table, which constitutes the monitor pseudocode, used for monitoring our nested locking example can be seen in Fig. 45.[16]

As with the other logic plugins we have the problem of not knowing when the last event of a trace slice will be seen. Thus, the plugin classifies traces into {match, fail, ?} (see Section 4.1.1). Originally, in [47], this was done by cloning the state of the monitor and seeing if the copy would accept on an end of trace event. After examining how the LR tables were constructed we noticed in [48] that if our parser is able to reduce on seeing an end of trace at all, then it must accept after some number of reductions. Thus, we just need to check if we can reduce assuming we are at the end of the trace instead of actually preforming the reductions. We refer to this concept as *guaranteed acceptance*. As a result, we no longer need to copy our parser's state and can just check whether the current state is the member of the set of states that would reduce at the end of the trace. More information on how to monitor CFGs can be found in [48].[17]

It should be noted that guaranteed acceptance is specific to monitoring and is *not* suitable for use in most parser generators since they do not need to only verify that a string is in the language but also to assemble an abstract syntax tree or produce some other side effects. Since these side effects are performed as part of the reductions omitting them is not, in general, possible. There are a number of open optimization opportunities, e.g., those suggested in [3,5]. Additionally, there are GLR specific opportunities to share common segments of the copied stacks. Therefore, there is still room for

[16] This is actually an LALR table, which is smaller than the full LR table, see [48].

[17] Note that normal LR(1) and LALR(1) parsing are used in [48], while in the meanwhile, as presented here, we generalized the techniques in [48] to work with arbitrary context-free languages.

$$
\begin{aligned}
\mathrm{G}(\epsilon) &= \{\emptyset\} \\
\mathrm{G}(t) &= \{\{t\}\} \\
\mathrm{G}(A) &= \textstyle\bigcup_{A \to \beta} \mathrm{G}(\beta) \\
\mathrm{G}(\beta_1\beta_2) &= \{S \cup T \mid S \in \mathrm{G}(\beta_1), T \in \mathrm{G}(\beta_2)\} \\
\mathrm{P}(\gamma) &= \{S \cup T \mid A \to \beta_1\gamma\beta_2, S \in \mathrm{P}(A), T \in \mathrm{G}(\beta_1)\} \\
\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) &= \mathrm{P}(e)
\end{aligned}
$$

Fig. 46. CFG $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ Defining Equations

⟨*PTCaRet Name*⟩	::=	"*ptcaret*"
⟨*PTCaRet Syntax*⟩	::=	"*true*" | "*false*" | ⟨*Event Name*⟩
	|	⟨*Unary Operator*⟩ ⟨*PTCaRet Syntax*⟩
	|	⟨*PTCaRet Syntax*⟩ ⟨*Binary Operator*⟩ ⟨*PTCaRet Syntax*⟩
⟨*Unary Operator*⟩	::=	"*not*" | "[∗]" | "<∗>" | "(∗)"
	|	"[∗*a*]" | "<∗*a*>" | "(∗*a*)"
	|	"@*b*" | "@*c*"
	|	"[∗*s*@*b*]" | "[∗*s*@*c*]" | "[∗*s*@*bc*]"
	|	"<∗*s*@*b*>" | "<∗*s*@*c*>" | "<∗*s*@*bc*>"
⟨*Binary Operator*⟩	::=	"*and*" | "*or*" | "*implies*" | "S" | "Sa"
	|	"*S*s@*b*" | "*S*s@*c*" | "*S*s@*bc*"
⟨*PTCaRet State*⟩	::=	"*validation*" | "*violation*"

Fig. 47. PTCaRet Syntax

generating better CFG monitors, though the current ones were satisfactory in our practical experiments.

To find the enables sets of a CFG we find the least fixed point of the equations in Fig. 46. Here, informally, $\mathrm{G}(A)$ is the set of events generated by the CFG, if the symbol A were used as the start symbol of the CFG. The rule $\mathrm{G}(\beta_1\beta_2) = \{S \cup T \mid S \in \mathrm{G}(\beta_1), T \in \mathrm{G}(\beta_2)\}$ generalizes this notion to entire strings of symbols. P is the enable sets function generalized to strings that include both non-terminals and terminals. For example, the prefixes of $abTB$ for B would be $\{\{a, b\} \cup S \mid S \in G(T)\}$. For a rule, $A \to \beta_1 B \beta_2$, $\mathrm{P}(B)$ needs to cope with the fact that A has its own enables set of possible prefixes. Thus its definition unions possible prefixes of A with the sets of symbols that are generated by β_1. The rest of MOP only needs to know sets of prefixes for events so $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ is just the restriction of P to events.

6.6 Past Time Linear Temporal Logic with Calls and Returns

Past time linear temporal logic with calls and returns (PTCaRet) [56] is a specialization of CaRet [5] for safety properties and their monitoring. CaRet is an extension of LTL with calls and returns. Matching call/return events in traces allows one to express program trace properties not expressible using plain LTL. One can express properties related to the contents of the program execution stack, such as "function g is always called from within function f", or one can express properties that are allowed to be temporarily validated/violated, such as "a user u may never directly access a password file (but may access it through system procedures)" [56]. Motivated by practical reasons, PTCaRet distinguishes call and return points from begin and end points: the former take place in the method caller's context and the latter take place in the callee's context. This distinction allows more flexible and elegant expressions of properties.

```
enter_phase_2 implies (
        not (not enter_phase_1 Sa begin)
        and (not acquire Sa enter_phase_1
            or not(not release Sa acquire))
        and @c (has_phase_2_pass)
        and <*s@b>(safe_exec))
```

Fig. 48. PTCaRet Example

PTCaRet is a past time variant of CaRet. Essentially, it is PTLTL extended by adding abstract variants of temporal operators. Fig. 47 shows the syntax for our PTCaRet plugin. PTCaRet includes all operators from PTLTL: the standard boolean operators and the temporal operators. PTCaRet, as mentioned, also has *abstract* temporal operators. The semantics of abstract operators is defined exactly as the semantics of their concrete counterpart operators, but they operate on the abstract version of the trace from which all the intermediate events of terminated method or function executions deeper in the call stack are erased [56]. In other words, abstract operators refer only to the trace of the current call stack level. In the syntax of the *abstract* temporal operators, "$*$" and "S" are followed by "a", meaning that the operator is an abstract variant of the concrete counterpart operator. The operators "$[*a]$", "$<*a>$", "$(*a)$", and "Sa" stand for "abstract always in the past", "abstract eventually in the past", "abstract previously in the past", and "abstract since", respectively.

PTCaRet also includes several derived operators which are convenient in practice, both for temporal and for stack operators. The operators "@b" and "@c", read "at begin" and "at call" respectively, are derived temporal operators meaning that the formula they take as an argument must hold "at the Beginning of the execution of the current function" and "at the context when the current function was Called", respectively. The semantics of the derived stack operators are defined exactly as the semantics of their *abstract* counterpart operators, but they operate only on the begin/call points on the abstract version of the trace. For example, derived stack operators defined on "begin" operate on a trace where we have filtered out all events except events in "begin" contexts from the abstract trace. Similar to the abstract temporal operators, in the syntax of the derived stack operators, "*" and "S" are followed by "s", meaning that the operator is a derived stack variant of the concrete counterpart operator. In addition to this keyword, either of "@b", "@c", or "@bc" follows right after "s", to indicate that the derived stack operator is defined on "begins", "calls", or "both begins and calls", respectively. In particular, the operators "$[*s@b]$", "$<*s@b>$", and "$Ss@b$" are the derived stack operators on the beginnings of method calls, meaning "always on begin contexts on abstract traces", "eventually on begin contexts on abstract traces", and "since on begin contexts on abstract traces", respectively. The derived stack operators for "calls" and "begins" are defined similarly.

Fig. 48 shows an example PTCaRet property from [56], which states that a program carrying out a critical multi-phase

```
if begin then {
  push(beta)
  exit
}
if end then {
  pop(beta)
  exit
}
beta[0] ← beta[0] or begin and safe_exec
beta[1] ← enter_phase_1 or not acquire
          and beta[1]
beta[2] ← acquire or not release and beta[2]
beta[3] ← begin or beta[3] and
          (not alpha[3] or alpha[2])
beta[4] ← begin or not enter_phase_1 and beta[4]
output(not enter_phase_2 or not beta[4]
       and beta[0]
       and (begin or beta[3])
       and (not begin or alpha[0])
       and (not beta[2] or beta[1])))
alpha[3] ← begin
alpha[2] ← alpha[1]
alpha[1] ← has_phase_2_pass
alpha[0] ← has_phase_2_pass
```

Fig. 49. PTCaRet Example Output

task should satisfy the following safety properties when execution enters the second phase:

- Execution entered the first phase in the same procedure;
- Resources acquired within the same procedure since the first phase must be released;
- The caller of the current procedure must have had approval for the second phase;
- Task is executed directly or indirectly by the procedure safe_exec.

Since the operators "Sa", "@c", and "$<*s@b>$" are abstract temporal operators, the example abstracts out events that happened in the procedure calls from within the current procedure.

Our technique for generating monitors from PTCaRet formulae is inspired from and generalizes the synthesis from plain PTLTL formulae (Section 6.4). The difference is that two bitvectors are kept for PTCaRet monitors, alpha[] and beta[]. The former plays the same role as the bitvector b from PTLTL (Section 6.4). The other bitvector, beta[], stores the validity status of the subformulae corresponding to abstract temporal operators. When a new function or method is called, a copy of the abstract bitvector is pushed onto the top of a stack. When the function or method ends, the bitvector is popped from the stack, effectively erasing all updates that happened during the called function or method.

Fig. 49 shows the output for Fig. 48. PTCaRet uses the sequential assignments explained in Section 6.4. In this example, we use the bitvector names and roles from previously, alpha[] and beta[]. Note that all elements in alpha[] and beta[] are initialized to false. When a new function or method is called, a copy of beta[] is pushed onto the top of a stack and when the function or method returns, the bitvector is popped from the stack, replacing beta[], while alpha[] stays as it is. Updating bitvectors before `output` is related to "since" operators, processing inner "since" operators before the outer ones. Updating bitvectors after `output` is related to "previously" operators, processing outer "previously" operators before the inner ones. Thus, bitvector updates before `output` are in order and bitvector updates after `output` are in reverse order. We refer to [56] for detailed monitor synthesis algorithm.

7 Conclusion

In this article we presented an overview of the Monitoring Oriented Programming (MOP) framework. We detailed the organization of the framework, and presented the formal syntax of specifications. We presented an in depth discussion of the two current instances of MOP: JavaMOP and BusMOP.

JavaMOP is an MOP instance for monitoring Java programs. JavaMOP specifications are compiled into AspectJ [41] aspects, which can be weaved into a program a user wishes to monitor using any standard AspectJ compiler such as ajc [7]. The theory of parametric monitoring was presented in the context of JavaMOP (currently, the only parametric instance of MOP). We discussed both naive and optimized monitoring algorithms for parametric properties. One important optimization, the enable set optimization, was explained in detail. We explained the two different modes of indexing from JavaMOP (centralized and decentralized), and we introduced three binding modes, which can be used as a filter on the number of handlers invoked by a given monitor. An evaluation of JavaMOP shows that, in general, parametric monitoring of Java programs is efficient.

BusMOP is an instance for monitoring PCI bus peripherals. BusMOP specifications are compiled into hardware design language code suitable for implementation on an FPGA [52]. Using a monitor implemented on an FPGA, it is possible to ensure the proper interaction between a peripheral and the system to which it is attached. BusMOP, in general, adds no runtime overhead to the monitored system.

Each logic plugin currently implemented in the MOP framework was presented. For each logic plugin we explained the syntax of the implemented logical formalism, how to generate a monitor, as well as how to generate enable sets.

8 Acknowledgments

Special thanks to Rodolfo Pellizzoni and Marco Caccamo, coauthors on [52], without whom the work on BusMOP would never have been completed. Also thanks to Choonghwan Lee for creating the JavaMOP installer program, and Michael Pradel for discussions that lead to the inclusion of the full-binding and connectedness modifiers of JavaMOP. Thanks to Klaus Havelund for using JavaMOP in his class at Caltech and Matthew Dwyer for using it at the University of Nebraska, thus forcing us to iron out the numerous bugs in earlier implementations.

References

1. *ISO/IEC 14977:1996, Information technology – Syntactic metalanguage – Extended BNF*. ISO, Geneva, Switzerland.
2. P. Abercrombie and M. Karaorman. jContractor: Bytecode instrumentation techniques for implementing DBC in Java. In *Runtime Verification (RV'02)*, volume 70 of *ENTCS*. Elsevier, 2002.
3. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. pages 215–246.
4. C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 345–364. ACM, 2005.
5. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 467–481. Springer, 2004.
6. AspectC++. `http://www.aspectc.org/`.
7. AspectJ. `http://eclipse.org/aspectj/`.
8. P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 589–608. ACM, 2007.
9. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69. Springer, 2004.
10. H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors. *Runtime Verification (RV'05)*, volume 144 of *ENTCS*. Elsevier, 2005.
11. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57. Springer, 2004.
12. H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RULER. *J. Logic Computation*, November 2008.
13. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass-Java with Assertions. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 103–117. Elsevier, 2001.
14. Bison. `http://www.gnu.org/software/bison/`.
15. S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.
16. E. Bodden. J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.
17. E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Aspect-Oriented Software Development (AOSD'09)*, pages 3–14. ACM, 2009.
18. E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *LNCS*, pages 525–549. Springer, 2007.
19. S. Chaudhuri and R. Alur. Instumenting C programs with nested word monitors. In *Model Checking Software (SPIN'07)*, volume 4595 of *LNCS*, pages 279–283. Springer, 2007.
20. F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. In *International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 357–372. Springer, 2004.
21. F. Chen, P. Meredith, D. Jin, and G. Roşu. Efficient formalism-independent monitoring of parametric properties. In *Automated Software Engineering (ASE'09)*, pages 383–394. IEEE, 2009.
22. F. Chen and G. Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 108–127. Elsevier, 2003.
23. F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 569–588. ACM, 2007.
24. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261. Springer, 2009.
25. M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.
26. D. Drusinsky. The Temporal Rover and the ATG Rover. In *Model Checking and Software Verification (SPIN'00)*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
27. Eagle Technology. *PCI 703 Series User's Manual*. `http://www.eagledaq.com/display_product_36.htm`.
28. Eiffel Language. `http://www.eiffel.com/`.
29. S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, pages 385–402. ACM, 2005.
30. K. Havelund, M. Nunez, G. Roşu, and B. Wolff, editors. *Formal Approaches to Testing and Runtime Verification (FATES/RV'06)*, volume 4264 of *LNCS*. Springer, 2006.
31. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*, pages 97–114. Elsevier, 2001.
32. K. Havelund and G. Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier, 2001.
33. K. Havelund and G. Roşu. Monitoring programs using rewriting. In *Automated Software Engineering (ASE'01)*, pages 135–143. IEEE, 2001.
34. K. Havelund and G. Roşu, editors. *Runtime Verification (RV'02)*, volume 70 of *ENTCS*. Elsevier, 2002.
35. K. Havelund and G. Roşu, editors. *Runtime Verification (RV'04)*, volume 113 of *ENTCS*. Elsevier, 2004.
36. K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
37. K. Havelund and G. Rosu. Efficient monitoring of safety properties. *J. Software Tools for Technology Transfer*, 6(2):158–173, 2004.
38. C. Hoare. *Communicating Sequential Processes*. Prentice-Hall Intl., New York, 1985.
39. J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, 1971.

40. JBoss. http://www.jboss.org.
41. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
42. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.
43. M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Europoean Conference on Real-Time Systems (ECRTS'99)*, 1999.
44. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00)*, pages 105–106. ACM, 2000.
45. H. Lu and A. Forin. The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Technical Report MSR-TR-2007–99, Microsoft Research, 2007.
46. M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, pages 365–383. ACM, 2005.
47. P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157. IEEE, 2008.
48. P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *J. Automated Software Engineering*, pages 149–180, 2010.
49. B. Meyer. *Object-Oriented Software Construction,* 2^{nd} *edition*. Prentice Hall, New Jersey, 2000.
50. PCI SIG. *Conventional PCI 3.0, PCI-X 2.0 and PCI-E 2.0 Specifications*. http://www.pcisig.com.
51. R. Pellizzoni, B. D. Buy, M. Caccamo, and L. Sha. Coscheduling of real-time tasks and PCI bus transactions. Technical report, University of Illinois at Urbana-Champaign, 2008. Available at http://netfiles.uiuc.edu/rpelliz2/www/techreps/.
52. R. Pellizzoni, P. Meredith, M. Caccamo, and G. Roşu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time System Symposium (RTSS'08)*, pages 481–491. IEEE, 2008.
53. R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha. Handling mixed-criticality in soc-based real-time embedded systems. In *Embedded Software (Emsoft'09)*, pages 235–244, 2009.
54. A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE, 1977.
55. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *J. Automated Software Engineering*, 2004.
56. G. Roşu, F. Chen, and T. Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Runtime Verification (RV'08)*, volume 5289 of *LNCS*, pages 51–68. Springer, 2008.
57. K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. In *Runtime Verification (RV'03)*, volume 89 of *ENTCS*, pages 162–181. Elsevier, 2003.
58. O. Sokolsky and M. Viswanathan, editors. *Runtime Verification (RV'03)*, volume 89 of *ENTCS*. Elsevier, 2003.
59. Soot website. http://www.sable.mcgill.ca/soot/.
60. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.
61. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *IBM Centre for Advanced Studies Conference (CASCON'99)*, pages 125–135. ACM, 1999.
62. Xilinx, Inc. *Virtex-4 ML455 PCI/PCI-X Development Kit User Guide*. http://www.xilinx.com/support/documentation/boards_and_kits/ug084.pdf.