Technical Report

Nr. TUD-CS-2012-0013
January 17th, 2012

CASED

On the Expressiveness of Parameterized Finite-state Runtime Monitors

Authors
Eric Bodden (EC SPRIDE, CASED)

On the Expressiveness of Parameterized Finite-state Runtime Monitors*

Eric Bodden
bodden@acm.org

Secure Software Engineering Group
European Center for Security and Privacy by Design
Technische Universität Darmstadt

Abstract. Many contemporary runtime-verification tools instrument a program under test with finite-state runtime monitors that can be parameterized through variable-to-object bindings. Often, such tools provide a specification formalism that is some form of syntactic extension to an aspect-oriented programming language. The tools then transform monitor specifications into aspects that bind the monitors' parameters through pointcuts.
In this work we show that the availability of certain, well-selected pointcuts and the variable bindings that these pointcuts expose can greatly enhance the expressive power of finite-state monitoring formalisms, often going clearly beyond the expressiveness of regular languages. Formally, this effect can be explained by a correspondence between finite-state machines and Weak Monadic Second-Order Logic (MSO). For this logic it is known that it can exactly express the regular languages when combined with a successor relation over string positions—adding other relations may increase its expressiveness. In the conceptual framework of this logic, such adjunct relations correspond directly to primitive pointcuts in the underlying aspect language.

1 Introduction

In the past, researchers have developed many tools to generate runtime monitors from high-level specifications. Many of these tools [1–5] generate monitors in the form of so-called *aspects* in the programming language AspectJ [6]. Using AspectJ, a runtime monitor can use *pointcuts* to define which types of events to process and *pieces of advice* to define which action should be taken at each type of event.

All monitoring tools mentioned above generate monitors from a high-level specification written in some formal specification language, such as linear temporal logic, regular expressions, or context-free grammars. The expressiveness of these formalisms has been extensively studied and is nowadays taught in many

* This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

undergraduate classes on language theory. In this work, however, we show that the expressiveness of the monitoring formalism may increase when combined with pointcuts that match certain, well-defined patterns of the execution trace, or expose certain, well-defined context information.

In particular, this paper presents the following original contributions:

– We show that a "startup" pointcut allows tracematches [1] to express precedence properties, making them equally expressive as the extended-regular-expression (ERE) plugin of JavaMOP [3].
– We show that a "shutdown" pointcut allows both JavaMOP's ERE plugin and tracematches to monitor bounded liveness properties.
– We show that a "cflowdepth" pointcut allows finite-state monitoring tools such as tracematches to express patterns of matching method entry and exit, although such patterns typically require a stack or counter for recognition and therefore normally cannot be expressed by finite-state systems.
– We present a "let" pointcut that allows programmers to expose arbitrary context information. As we show, runtime monitors can use this context information to project the execution trace onto ground traces which are then easier to recognize.

Further, we show how a well-known correspondence between finite-state machines and Weak Monadic Second-Order Logic (MSO) can be used to reason about the effect of adding pointcuts. For this logic is is known that it can exactly express the regular languages when combined with a successor relation over string positions. Adding other relations, on the other hand, may increase its expressiveness. Adding pointcuts to a finite-state monitoring language can be formalized by adding corresponding adjunct predicates to MSO. While a formal treatment of the pointcuts that we propose here goes beyond the scope of this paper, we expect this correspondence to yield interesting results in future work.

The remainder of this paper is organized as follows. In Section 2 we explain the correspondence between finite-state machines and Weak Monadic Second-Order Logic (MSO). In Section 3 we show the effect of adding new primitive, parameter-less pointcuts to a monitoring language. Section 4 lifts these results to pointcuts with parameters. We discuss related work in Section 5 and conclude in Section 6.

2 Monadic second-order logic

In first-order logic, formulas can quantify over primitive values. First-order logic over words is exactly as expressive as future-time linear temporal logic (LTL), i.e., can be used to describe any regular language whose recognition does not require modulo counting. (One language that does require such counting, and is not expressible in LTL is the language $(a\ a)^*$ of all words of even length.)

Second-order logic is an extension to first-order logic that further allows for the quantification over *relations* of primitive values. In the following we will consider *monadic* second-order logic (MSO), in which all quantified relations must be monadic, i.e., must represent sets of primitive values. In 1960, Büchi, Elgot, and Trakhtenbrot independently showed that a certain class of monadic second-order formulas is exactly as expressive as the regular languages. Their proof is constructive: from every finite-state machine accepting a regular language $\mathcal{L}$ one can construct an MSO formula whose models are exactly all elements of this language. The same also holds the other way around, but we will not require the other direction (from logic to finite-state machines) in this paper. In this section we briefly introduce monadic second-order logic and sketch the original equivalence proof.

Monadic second-order logic over words over a finite alphabet Σ uses formulas constructed of the following elements:

Position variables $x, y, z \dots$ denoting positions in a finite input string that is itself an element of Σ^*,
Constant positions *min* and *max*, denoting the first and last position in the string,
Quantified set variables $X, Y, Z \dots$ denoting finite sets of positions,
Quantifiers $\exists$ and $\forall$ over position and set variables,
Boolean connectives $\wedge, \vee, \neg, \dots$

Any MSO logic can be enhanced with additional *adjunct relationships*. As we will see later in this paper, the exact selection of adjunct relationships is a crucial factor of the logic's expressiveness. To obtain a logic that is exactly as expressive as regular languages, one needs to provide exactly the following adjunct relationships:

Successor relationship $S(x, y)$ over positions, and
Label relationships L_a for every $a \in \Sigma$. For example, the predicate $L_a(x)$ describes that the symbol at the x-th position of the input string is an a.

Note that, because we restrict ourselves to *monadic* logic all polyadic relations (n-ary for $n > 1$) must be fixed and pre-defined. In other words, one may only quantify over monadic relations, not polyadic relations.

Example 1. The following MSO formula expresses the language $(\Sigma\Sigma) \cdot (\Sigma\Sigma)^*$ of all non-empty words of even length:

$$\exists X(X(min) \wedge \neg X(max) \wedge \forall y \forall z(S(y, z) \rightarrow (X(y) \leftrightarrow \neg X(z))))$$

The formula demands that there exists a set of positions X, containing the first position of the string but not the last, and that the set contains every second position: if z is a successor of y then y is in X, i.e., $X(y)$ holds, if and only if $\neg X(z)$.

One can visualize the relationship between the input word and the quantified set X as shown in Figure 1. In the figure, every position that is a member of X is checked (✓).

input word	e	v	e	n	l	e	n	g	t	h
X	✓		✓		✓		✓		✓	

Fig. 1: Input word "evenlength" and quantified set X

Expressing finite-state machines through MSO formulas

As Büchi [7], Elgot [8] and Trakhtenbrot [9] independently showed, one can express every finite-state machine as an MSO formula that only uses the adjunct relationships $S(x, y)$ and L_a that we mentioned above. Because of the authors' last names, this is frequently called the "BET theorem." For every state s of the finite-state machine one quantifies over a fresh set X_s of positions.

Let $\mathcal{M} = (Q, \Sigma, 0, \Delta, F)$ be a finite-state machine with $Q = \{0, .., n\}$. Then we define the formula $\phi_{\mathcal{M}}$ as:

$$\begin{aligned}
&\exists X_0 \ldots \exists X_n : \\
&X_0(min) \\
&\wedge \forall x \forall y (S(x, y) \rightarrow \bigvee_{(i,a,j)\in\Delta} X_i(x) \wedge L_a(x) \wedge X_j(y)) \\
&\wedge \bigvee_{(i,a,j)\in\Delta, j\in F} X_i(max) \wedge L_a(max)
\end{aligned}$$

The BET theorem shows that for every $w \in \Sigma^*$:

$$w \in \mathcal{L}(\mathcal{M}) \Leftrightarrow w \models \phi_{\mathcal{M}},$$

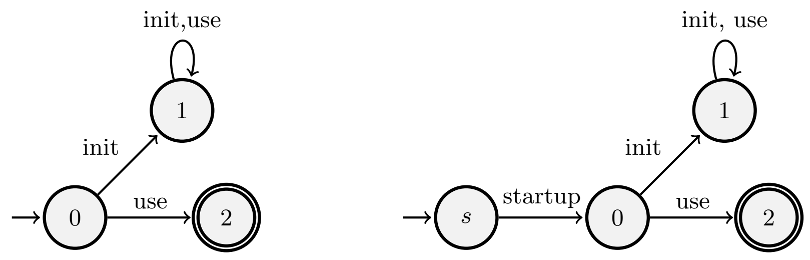
In other words, a finite-state machine accepts w exactly if w (and the label relations that it induces) satisfy $\phi_{\mathcal{M}}$.

Example 2. For example, consider the finite-state machine in Figure 2a. According to the BET theorem, the following formula is equivalent to this finite-state machine:

$$\begin{aligned}
&\exists X_0 \exists X_1 \exists X_2 : X_0(min) \wedge \forall x \forall y \big(\\
&\quad (X_0(x) \wedge L_{init}(x) \wedge X_1(y)) \vee \\
&\quad (X_0(x) \wedge L_{use}(x) \wedge X_2(y)) \vee \\
&\quad (X_1(x) \wedge L_{init}(x) \wedge X_1(x)) \vee \\
&\quad (X_1(x) \wedge L_{use}(x) \wedge X_1(x)) \,\big) \;\wedge \\
&X_0(max) \wedge L_{use}(max)
\end{aligned}$$

Note that the clause $X_0(max) \wedge L_{use}(max)$ denotes that we reach an accepting state when, just before the end, being in state 0 and reading a "*use*."

The BET theorem is important to the remainder of this paper, because it helps us explain some of our observations. In remainder of this paper we will



(a) State machine for JavaMOP (b) State machine for Tracematches

Fig. 2: State machines for pattern "no use before init"

show that adding primitive pointcuts to a monitoring formalism may enhance this formalism's expressiveness. On the level of language theory and finite-state machines, this added expressiveness may be hard to quantify. The BET theorem allows us, though, to think in terms of formulae instead of languages or state machines. In this model, pointcuts are just other adjunct n-ary predicates, similar to the successor predicate from above. In the field of logics it is well known that adding such predicates to MSO makes the logic more expressive. Reasoning about pointcuts this way hence appears natural.

3 Adding pointcuts without parameters

In this section, we will discuss the effect of adding two different pointcuts to a monitoring logic: "startup" and "shutdown." Both pointcuts take no parameters, i.e., they expose no context information. Nevertheless, they manage to enhance a formalism's expressiveness through the well-defined position at which they occur in the execution trace (startup at the beginning and shutdown at the end).

3.1 A "startup" predicate to turn suffix matching into full-string matching

Two prominent examples of parameterized logics for runtime verification are Extended Regular Expressions (ERE), as implemented by the JavaMOP tool [3], and tracematches [1], implemented as an extension to the AspectBench Compiler [10]. Both tools offer a similar abstract syntax: regular expressions are not defined over ground symbols but over symbols that may have parameters. A word is only accepted if it instantiates equal parameters with equal concrete bindings. For instance, consider the following regular expression:

$$\mathit{create}(c,i) \quad \mathit{next}(i)^* \quad \mathit{update}(c)^+ \quad \mathit{next}(i)$$

This expression matches execution traces that violate the "FailSafeIter" property [11]: after creating an iterator i for a collection c, possibly iterating a few times, but then updating the underlying collection c, it is then an error to iterate using i again. A runtime monitor generated from this regular expression would, for concrete objects c_1, c_2, and i_1, match the execution trace

$$\mathit{create}(c_1, i_1) \quad \mathit{update}(c_1) \quad \mathit{next}(i_1)$$

but it would not match the trace

$$\mathit{create}(c_1, i_1) \quad \mathit{update}(c_2) \quad \mathit{next}(i_1)$$

as this trace contains a conflicting binding, associating c both with c_1 and c_2.

Despite their similarities, tracematches and ERE from JavaMOP have a quite different matching semantics. Tracematches use so-called *suffix matching*: a tracematch whose regular expression defines the regular language $\mathcal{L}$ accepts a word w if any *suffix* of w is in $\mathcal{L}$. For instance, a tracematch with the regular expression "b" would match the input "ab", ignoring the initial prefix "a" of the input. This is different from ERE, which uses *full-string matching*: JavaMOP would not accept the input "ab" for the same regular expression "b."

Returning to our correspondence to MSO, the tracematch semantics effectively demand that for all positions x it holds that $X_0(x)$: the finite-state machine always remains (also) in the initial state, so that it can start a match at any point, irrespective of any prefix it may already have read.

This semantics of tracematches lead to the fact that it is impossible to express *precedence properties* [12] such as "no use before init" as a tracematch. After all, a tracematch for this property would have to accept the string $w_1 =$"use" (because it violates the property) but at the same time would have to reject the string $w_2 =$"init use" (because this sequence constitutes a valid use). Since w_1 is a suffix of w_2, no such tracematch can exist: any tracematch that accepts w_1 would automatically also accept w_2.

Using MSO we can easily conduct a formal proof to show this deficiency. Assume there exists a tracematch tm accepting the string "use". Because tm accepts "use", we know that the following is a sub-formula of tm's MSO representation: $X_0(max) \wedge L_{use}(max)$. According to the tracematch semantics, we have $\forall x : X_0(x)$. Therefore, tm cannot possibly reject the trace "init use": after reading the prefix "init", $X_0(x)$ still holds, and therefore at this position, $X_0(max) \wedge L_{use}(max)$ will evaluate to "true". $\square$

Adjunct predicates to the rescue. As we will show in the following, one can enhance the expressiveness of a monitoring logic such as tracematches by adjoining additional primitive predicates that have certain well-defined properties. In the particular case of precedence properties, the problem is that tracematches match against every suffix of the input and therefore "forget" any prefix that probably should have prevented a match.

Assume now, that we add a new primitive predicate *startup*, which only holds at program startup time, i.e., at the first state of the execution trace. If such a primitive predicate exists, then we can define a tracematch of the form

$$\mathit{startup} \quad \mathit{use}$$

over the alphabet $\{startup, use, init\}$. Figure 2b shows the appropriate finite-state machine for this tracematch. Because the symbol *startup* is known to occur only once, at the beginning of the trace, it does not matter that the tracematch automaton also always remains in the initial state (now s): the automaton can enter state 0 only once, after reading the initial *startup* symbol. Hence, this tracematch effectively accepts the same traces as the ERE implementation of JavaMOP with the extended regular expression "*use*."

Returning to MSO, the adjunct predicate *startup* can be visualized as in Figure 3: the predicate defines a singleton set of positions that only contains the very first position.

Again we can use MSO to reason about the correctness of this construction. Because "startup" occurs at the beginning of the trace, and only there, we obtain:

$$L_{startup}(min) \wedge (\forall x : x > min \rightarrow \neg L_{startup}(x))$$

Using this formula, it is easy to show that the state machine from Figure 2b accepts the trace "use" while rejecting the trace "init use." We leave the formal proof as an exercise to the reader.

A startup predicate can be implemented using different techniques. For languages such as Java, the predicate is actually relatively hard to define, since there may be multiple ways to bootstrap a Java program. It is however easily possible to capture the general case, in which a Java program is started by executing its `main` method. The following AspectJ advice captures this event:

```
before(): execution(public static void main(String[])) {
  //send startup event to monitor
}
```

3.2 A "shutdown" predicate to monitor bounded liveness properties

A *liveness property* is a property stating that some event a must happen at some point in the future. The canonical example of a liveness property expressed in linear temporal logic is $F\ a$, stating that a must happen "finally", i.e., at some point in time. Using runtime monitoring it is generally impossible to detect that a program violates a liveness property: there can be no prefix of the execution trace (the prefix monitored so far) after which the monitor can decide that a will never happen in the future [13].

A *bounded* liveness property is a liveness property stating that a must happen at some point in the future but before some other event b. For our purposes we will only consider special bounded liveness properties where a must occur before

input word	use	use	init
startup	✓		

Fig. 3: Input word "use use init" and adjunct predicate *startup*

the program shuts down, i.e., before the end of the program run. To be able to reason about the end of the program run, we hence define an adjunct predicate *shutdown* which matches program shutdown, conversely to the *startup* predicate from above.

Using the *shutdown* predicate, one can monitor violations of the property "finally a" simply through the regular expression "*shutdown*" over the alphabet $\{shutdown, a\}$: the monitor will signal a property violation if *shutdown* is observed without having observed a before.

Note that monitoring such bounded liveness properties would be impossible without a *shutdown* predicate, regardless of whether ERE from JavaMOP or tracematches or any other monitoring logic is used. Without being notified of program shutdown there would be no way to determine the end of the execution trace online.

While implementing a *shutdown* predicate may be hard in C-like languages, it is comparatively simple in Java. Java provides a mechanism called *shutdown hooks* [14]. Such hooks are un-started threads that the virtual machine executes automatically when the application shuts down, no matter how the shutdown was initiated.[1] A possible implementation of a *shutdown* predicate could hence be implemented as follows:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() { /* send shutdown event to monitor */ }
});
```

Through a dummy method, the event can even be exposed as an AspectJ pointcut:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() { __shutdown(); }
});

pointcut shutdown(): call(* __shutdown());
```

In the runtime verification tool J-LO [2] programmers define properties using a special linear temporal logic that has finite-trace semantics. In this semantics, a formula $F\ a$ means that a must occur before shutdown. J-LO uses shutdown hooks to implement the runtime verification of such formulas. This automatically implies that J-LO's monitoring logic can express properties that JavaMOP and tracematches could not express if no *shutdown* predicate were added.

4 Adjunct predicates with parameters

There are other properties, which finite-state based parameterized runtime verification formalisms cannot usually express, due to the fact that their monitors can only use a finite set of states. For instance, consider the property that a

[1] Shutdown hooks are only not executed if an application is forced to quit, e.g. with the command "`kill -9`."

resource like a file handle should be closed in the same method invocation that allocated the resource. To recognize matching method entries and exits in the case of recursive method calls, one generally has to use a push-down automaton or a counter—a finite number of states is insufficient in general. In JavaMOP, programmers can use the context-free grammar formalism to yield a push-down automaton which is able to monitor the resource-allocation property. In tracematches, however, which are bound to finite-state machines, it is unclear how such a monitor could be implemented.

However, also here adjunct predicates help to overcome this limitation of tracematches. In previous work [15], Avgustinov et al. present the pointcut `cflowdepth(i,pc)`, which takes the following two parameters:

– `i`: an "out" parameter binding `i` to the number of enclosing joinpoints (including the current one) that match `pc`, with
– `pc`: any pointcut.

Similarly, the pointcut `cflowbelowdepth(i,pc)` tests for strictly enclosing joinpoints (excluding the current one).

For instance, consider the tracematch from Figure 4. This tracematch defines two symbols `beforefunc` and `afterfunc`, which use the pointcut `anyfunc` (defined elsewhere) to match before, respectively after the execution of any method. In both symbols, `cflowdepth(i, anyfunc())` binds `i` to the depth of the current control flow (again with respect to any method). The symbols `alloc` and `release` match on the allocation and release of any `ResourceReference`. Those symbols use `cflowbelowdepth(i, anyfunc())`. The effect of this pointcut becomes clear when considering the tracematch pattern "`beforefunc alloc afterfunc`": the tracematch triggers when entering a method execution, allocating a resource and then exiting the same method execution without having seen a matching release. Because we used `cflowbelowdepth`, the tracematch will only trigger for such executions where the allocation occurred directly below the last monitored `beforefunc`. The tracematch will therefore issue its error message immediately when the innermost violating method execution finishes.

```
perthread tracematch(ResourceReference r, int i) {
    sym beforefunc before : anyfunc() && cflowdepth(i, anyfunc());
    sym afterfunc after : anyfunc() && cflowdepth(i, anyfunc());
    sym alloc after : alloc(r) && cflowbelowdepth(i, anyfunc());
    sym release after : release(r) && cflowbelowdepth(i, anyfunc());

    beforefunc alloc afterfunc {
            throw new RuntimeException("Unmatched resource allocation!");
    }
}
```

Fig. 4: Tracematch monitoring unmatched resource allocations

Note that this adjunct predicate differs from the former in that it binds some of its arguments: it exposes a context value, the current stack depth, as a variable binding. Because parameterized runtime monitoring systems associate a different monitor with each variable binding, this allows a finite-state runtime monitoring tool like tracematches to obtain an expressiveness beyond regular languages. In Figure 5, we show how the `cflowdepth` pointcut enriches the program's execution trace: the new adjunct pointcut virtually adds a new dimension to the trace, which can then be projected against.

In the above example, parameters are used to separate different matches from each other: because each complete match refers to one and the same stack depth `i`, this match refers to one stack frame only. We find that this is a recurring theme in parameterized runtime monitoring formalisms: parameters exist to partition the monitoring space into smaller units, which are then easier to reason about. In some cases like the above, this may mean that the smaller unit may then be described by a regular language, while the complete property is clearly not regular.

4.1 A word on performance

In this work we have shown that one can enhance a formalism's expressiveness by adding certain well-chosen adjunct predicates. But what is the runtime cost involved? Fortunately, previous work has shown that all predicates mentioned above can be implemented with negligible runtime cost. In particular, computing the truth value of any such predicate is in $O(1)$, i.e., takes only constant time. The most complex predicate we discussed is `cflowdepth`, and even this predicate amounts to just updating a counter at those positions at which the associated pointcut matches. Avgustinov et al. showed that such counters induce almost no perceivable overhead [16].

The let-pointcut

As we just showed, the exposure of the correct variable bindings can mean the difference between being and not being able to monitor a given property in practice. It is therefore crucial to have a means to expose context information for the purpose of generating such variable bindings. Because an execution trace may contain many different kinds of context information, the tracematch developers proposed [15] a unified way to expose context information in general: the pointcut "`let(a,expr)`." This pointcut expects two parameters:

– `a`: the variable to be bound by the pointcut, and

input trace	beforefunc	beforefunc	beforefunc	alloc	afterfunc	afterfunc	afterfunc
cflowdepth	0	1	2	2	2	1	0

Fig. 5: Effect of `cflowdepth` pointcut.

– `expr`: an expression whose return value will be assigned to `a`.

In the current implementation of the `let` pointcut, the expression can access all static members and the variable `thisJoinPoint`, which AspectJ implicitly declares.

For instance, consider the tracematch in Figure 6, which issues a runtime error message just before a certain null-pointer access is going to occur. In this example, the surrounding aspect first declares pointcuts `fieldSet`, `fieldSetToNull` and `fieldAccess` to monitor (1) field assignments in general, (2) assignments that assign `null`, and (3) reading field accesses. All pointcuts use the primitive `let` pointcut to expose the field's signature for matching. In addition, the second pointcut uses `let` to further expose the current `JoinPointStaticPart` to obtain debugging information. Exposing the `fieldID` helps the matching process to project the execution trace onto a trace related to each field. This then makes it quite easy to reason about each field in isolation: if a field is set to `null` (we monitor `fieldSetToNull`), and then it is read from (we monitor `fieldAccess`), and there is no intervening other assignment (we would have monitored `fieldSet`), then we certainly read a `null` value at this point. The tracematch's regular expression directly denotes this pattern. Note that, in this case, the parameter `jp` is only exposed to obtain enhanced debug information: the tracematch can print the source location of the assignment that wrote the `null` value. This can easily be seen because only one symbol, the symbol `fieldSetToNull`, exposes this value. This is different from the value `fieldID` which all symbols reference. A complete match hence has to agree on the variable binding at all the events matched by these symbols.

Connection to Monadic Second-order Logic

Such pointcuts that expose context values though variable bindings can be modeled using monadic second-order logic (MSO). We remind the reader that MSO is just as expressive as the regular languages when combined with monadic label predicates $L_a(x)$ and a single binary successor relation $S(x, y)$. We can then easily model additional pointcuts as predicates, as follows. Pointcuts that expose no context information are modeled as monadic (unary) predicates that only accept a position parameter as argument. For instance, *shutdown(x)* denotes that a shutdown event occurred at string position x. Likewise, pointcuts that expose n context values to the monitoring logic are modeled as $(n + 1)$-ary predicates, accepting one position argument and further n "variable arguments" that can bind context information. That way, one can model the pointcut `let(a,expr)` as $let_{expr}(x, a)$, where x is a variable denoting a string position and a a variable exposing a context value.

That way, we can use MSO to denote the tracematch from Figure 6 as follows:

```
public aspect NullPointerCheck {

    pointcut fieldSet(String fieldID ): set(* *.*)
     && let(fieldID, thisJoinPoint.getSignature().toString().intern ());

    pointcut fieldSetToNull(String fieldID, JoinPointStaticPart jp, Object arg):
      fieldSet ( fieldID ,arg) && let(jp, thisJoinPointStaticPart) && if(arg==null);

    pointcut fieldAccess(String fieldID ): get(* *.*) &&
      let( fieldID , thisJoinPoint.getSignature().toString().intern ());

    tracematch(String fieldID, JoinPointStaticPart jp) {
        sym fieldSetToNull after: fieldSetToNull( fieldID ,jp ,*);
        sym fieldSet after: fieldSet ( fieldID );
        sym fieldAccess before: fieldAccess( fieldID );

        fieldSetToNull /*no fieldSet in between */ fieldAccess {
            System.out.println("About to read null! "+
            "Guilty assignment in line : +"jp.getSourceLocation()); }
    }
}
```

Fig. 6: Tracematch monitoring certain null-pointer accesses

$$\begin{aligned}
&\exists X_0 \exists X_1 \exists X_2 : X_0(min) \wedge \forall x \forall y \forall f \forall j \big(\\
&\quad (X_0(x) \wedge L_{\mathit{fieldSetToNull}}(x, f, j) \wedge X_1(y)) \vee \\
&\quad (X_1(x) \wedge L_{\mathit{fieldSet}}(x, f) \wedge X_2(y)) \big) \;\; \wedge \\
&X_1(max) \wedge L_{\mathit{fieldAccess}}(max, f)
\end{aligned}$$

Here, f denotes the `fieldID`, j denotes `jp` and the state with number 2 is a "sink state", which the automaton moves to when a `fieldSet` event is read.

5 Related Work

In the following, we discuss related work on parametric runtime monitoring.

5.1 Stolz and Huch

Stolz and Huch [17] present an approach to parametric runtime monitoring of concurrent Haskell programs. The authors specify program properties using linear temporal logic formulae. Such formulae are generally evaluated over a propositional event trace: a formula refers to a finite set of named propositions and any of the propositions can either hold or not hold at a given event. Stolz and

Huch implemented a runtime library that would generate a propositional event trace at runtime and update a linear temporal logic formula according to the monitored propositional values. The library reports a property violation when the formula reduces to "false." The formulas that Stolz and Huch allow for can be parameterized by different values.

5.2 J-LO

We ourselves developed J-LO, the Java Logical Observer [2], a tool for runtime-checking temporal assertions in Java programs. J-LO follows Stolz and Huch's approach in large parts, however the propositions in J-LO's temporal-logic formulae carry AspectJ pointcuts as propositions. The J-LO tool accepts linear temporal logic formulae with AspectJ pointcuts as input, and generates plain AspectJ code by modifying an abstract syntax tree. J-LO extends the AspectBench Compiler, which allows it to then subsequently weave the generated aspects into a program under test. Pointcuts in J-LO specifications can be parameterized by variable-to-object bindings.

5.3 Tracematches

Allan et al. [1] are the creators of tracematches. Tracematches share with J-LO the idea of generating a low-level AspectJ-based runtime monitor from a high-level specification that uses AspectJ pointcuts to denote events of interest, however tracematches offer a much more sophisticated and efficient implementation. Tracematches are implemented on top of the AspectBench Compiler [10] (abc), which offers implementations of all the pointcuts that we discussed in this paper. The use of an extensible compiler such as abc makes it easy to add further pointcuts as required.

5.4 Tracecuts

Walker and Viggers developed tracecuts [18], an approach that monitors programs with respect to a specification given as a context-free grammar over AspectJ pointcuts. Context-free grammars are strictly more expressive than the finite-state patterns that, for instance, tracematches can express. However, in this paper we showed that tracematches can express some important context-free patterns when a "cflowdepth" pointcut is added.

5.5 JavaMOP

JavaMOP provides an extensible logic framework for specification formalisms [3]. Via logic plug-ins, one can easily add new logics into JavaMOP and then use these logics within specifications. JavaMOP ships with several built-in specification formalisms, including extended regular expressions (ERE), past-time and future-time linear temporal logic (PTLTL/FTLTL), and context-free grammars. JavaMOP translates specifications into AspectJ aspects. Regardless of the specification formalism that is used, all specifications can be parameterized by variable-to-object bindings. one of the key contributions of the work on JavaMOP is a set of general algorithms that allow for efficient monitoring of parameterized properties irrespective of the monitoring formalism at hand.

5.6 PQL

The Program Query Language [19] by Martin at al. resembles tracematches in that it enables developers to specify properties of Java programs, where each property may bind free variables to runtime heap objects. PQL supports a richer specification language than tracematches: it uses stack automata rather than finite state machines, which yields a language slightly more expressive than context-free grammars.

5.7 PTQL

Goldsmith et al. [20] proposed PTQL, the Program Trace Query Language, which provides an SQL-like language for querying properties of program traces at runtime. The authors also provide "partiqle", a compiler for this language. The compiler instruments the program that is to be queried so that the program notifies monitoring code about the appropriate events at runtime. The monitor itself uses indexing trees to associate the monitor's internal state with the appropriate objects.

6 Conclusion

In this work we showed that the expressiveness of parametric runtime monitoring formalisms can be increased by adding further, well-defined predicates, for example by providing additional primitive AspectJ pointcuts. As we showed, in some cases this increase in expressiveness can even go so far as to make such languages monitorable by a finite-state runtime verification tool that are themselves clearly not regular languages. We explained this effect by showing up a correspondence between finite-state machines and monadic second-order logic. In this logic, pointcuts can be seen as special predicates. Pointcuts that expose context-values are polyadic predicates where one argument of the predicate refers to the current position in the trace and the other arguments refer to the context values that the pointcut exposes.

Using this conceptual framework, we were able to show that tracematches can monitor precedence properties when enhanced with a "startup" predicate, that parametric runtime monitoring tools in general can monitor bounded liveness properties when enhanced with a "shutdown" predicate and that finite-state monitors can use a "cflowdepth" predicate to monitor properties that require matching method entries with their exits.

Acknowledgements Many of the insights of this paper arose from discussions with the developers of tracematches, in particular Pavel Avgustinov, Julian Tibble and Oege de Moor. Thanks a lot for the fruitful discussions and for developing and maintaining tracematches in the first place! Much gratitude also goes to the reviewers of the 2011 NASA Formal Methods Symposium, who provided valuable feedback on an earlier version of this paper. This work was supported by CASED (www.cased.de).

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: OOPSLA, ACM Press (October 2005) 345–364
2. Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (November 2005)
3. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA, ACM Press (October 2007) 569–588
4. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: Symposium on the Foundations of Software Engineering (FSE), ACM Press (November 2006) 219–230
5. Krüger, I.H., Lee, G., Meisinger, M.: Automating software architecture exploration with M2Aspects. In: Workshop on Scenarios and state machines: models, algorithms, and tools (SCESM), ACM Press (May 2006) 51–58
6. : The AspectJ home page (2003)
7. Büchi, J.R.: Weak second-order arithmetic and finite automata. Zeitschrift für mathematische Logik und Grundlagen der Mathematik (6) (1960) 66–92
8. Elgot, C.C.: Decision problems of finite automata design and related arithmetics. Trans. Amer. Math. Soc. (98) (1961) 21–51
9. Trakhtenbrot, B.A.: Finite automata and the logic of one-place predicates. Siberian Math. J. (3) (1962) 103–131 English translation in: AMS Transl. 59 (1966) 2355.
10. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An extensible AspectJ compiler. In: AOSD, ACM Press (March 2005) 87–98
11. Bodden, E., Hendren, L.J., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: European Conference on Object-Oriented Programming (ECOOP). Volume 4609 of LNCS., Springer (2007) 525–549
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering (ICSE), ACM Press (May 1999) 411–420
13. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) **3**(1) (2000) 30–50
14. inc., O.: Design of the shutdown hooks api `http://download.oracle.com/javase/1.4.2/docs/guide/lang/hook-design.html`.
15. Avgustinov, P., Tibble, J., Bodden, E., Lhoták, O., Hendren, L., de Moor, O., Ongkingco, N., Sittampalam, G.: Efficient trace monitoring. Technical Report abc-2006-1, `http://www.aspectbench.org/` (03 2006)
16. Avgustinov, P., Tibble, J., Bodden, E., Lhoták, O., Hendren, L., de Moor, O., Ongkingco, N., Sittampalam, G.: Efficient trace monitoring. Technical Report abc-2006-1 (March 2006)

17. Stolz, V., Huch, F.: Runtime verification of concurrent haskell programs. Electronic Notes in Theoretical Computer Science (ENTCS) **113** (January 2005) 201–216
18. Walker, R., Viggers, K.: Implementing protocols via declarative event patterns. In: Symposium on the Foundations of Software Engineering (FSE), ACM Press (October 2004) 159–169
19. Martin, M., Livshits, B., Lam, M.S.: Finding application errors using PQL: a program query language. In: OOPSLA, ACM Press (October 2005) 365–383
20. Goldsmith, S., O'Callahan, R., Aiken, A.: Relational queries over program traces. In: OOPSLA, ACM Press (October 2005) 385–402