

Event-Based Runtime Verification of Java Programs

Marcelo d'Amorim^{* †}
Department of Computer Science
University of Illinois Urbana-Champaign, USA
damorim@cs.uiuc.edu

Klaus Havelund
Kestrel Technology
Palo Alto, USA
havelund@kestreltechnology.com

ABSTRACT

We introduce the temporal logic HAWK and its supporting tool for runtime verification of Java programs. A monitor for a HAWK formula checks if a finite trace of program events satisfies the formula. HAWK is a programming-oriented extension of the rule-based EAGLE logic that has been shown capable of defining and implementing a range of finite trace monitoring logics, including future and past time temporal logic, metric (real-time) temporal logics, interval logics, forms of quantified temporal logics, extended regular expressions, state machines, and others. Monitoring is achieved on a state-by-state basis avoiding any need to store the input trace. HAWK extends EAGLE with constructs for capturing parameterized program events such as method calls and method returns. Parameters can be executing thread, the objects that methods are called upon, arguments to methods, and return values. HAWK allows one to refer to these in formulae. The tool synthesizes monitors from formulae and automates program instrumentation.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specification—*languages*; D.2.5 [Software Engineering]: Testing and Debugging—*monitors*; D.2.4 [Software Engineering]: Software/Program Verification—*assertion checkers, reliability*

General Terms

Runtime verification, temporal logic, event versus state predicates, Java, program instrumentation, aspect oriented programming.

1. Introduction

Model Checking [7], Theorem Proving [16] and Static Analysis [18] are techniques aiming at *static* program verification. The first

^{*}CAPES grant# 15021917.

[†]This author is grateful for the support received from MCT while participating in the Summer Student Research Program at the NASA Ames Research Center.

is concerned with checking if all possible traces derived from a program (or abstract model) satisfy a property of interest. The state-space explosion is known to be an issue when considering concurrency and unbounded types. Additional model abstraction, such as partial-order reduction, can reduce the model size considerably but *scalability* is still an issue when checking properties of programs in general. Theorem proving relies on the language semantics and a proof system in order to come up with a proof that the program will behave correctly for all possible inputs. Unfortunately, this technique cannot be fully automated for undecidability reasons. As yet another technique, static analysis is concerned with analyzing the program offline to determine program properties from the program structure. Usually such static analyzers can only detect a limited set of generic errors, such as array-bound violations, deadlock potentials etc, and they often yield false positives. However, they usually scale fairly well, and they often are guaranteed to detect designated problems if they exist.

In contrast to these techniques, this paper describes a logic and tool that employs *dynamic* analysis to detect bugs in software during execution. Runtime Verification (RV) [1] is concerned with checking a single trace of events generated from the program run against properties described in some logic. When a property is violated or validated, the program can take actions to deal with it. The technique scales since just one model of computation is considered, rather than the entire state space as in model checking. The technique can be used both for testing before deployment of the software, and for monitoring after deployment. In the first case, one must come up with test cases [3] that might uncover a bug. In this setting, RV is considered as an auxiliary tool to automate the creation of oracles that detect errors. In the second case, RV is used to monitor a program during its execution so to take actions in response to the violation of properties. With this perspective in mind, an RV tool may be used to define how the program reacts to bugs, possibly steering it to the correct behavior [10].

This paper describes a logic and its tools, named HAWK, for runtime verification of JAVA programs. By using a subset of JAVA's expression language as propositions, the user can describe temporal properties relating different points in the program and their accessible objects, and verify the program against these properties during runtime. For instance, one can state a requirement that if a file is ever written, it must have been opened before. This is written in HAWK as follows:

```
Always([file?.write(*)] Previously({file.open()} true))
```

where `file` denotes a file object, the operator `Always` means always in the future, and `Previously` means eventually in the past. This example has the sole purpose of giving early in this paper

“some” intuition on the role of objects, instrumentation, and how these concepts integrate in HAWK. We will go over the details of the language in later sections. In particular, we will give the denotation of the square and angled brackets which enclose a description for a program event.

HAWK is defined on top of EAGLE which is more expressive than several logics [4]. EAGLE not only allows one to state temporal and interval properties but also to define new logics. We will present briefly the main concepts of the EAGLE logic.

Instrumentation is acknowledged as an issue that runtime verification tools have to face in order to monitor programs [14, 11, 6]. Some tools provide no support for mechanical instrumentation, others use annotations in the source program to check against verification formulae. We understand that automated instrumentation is part of the problem we want to solve. Therefore, a tight integration between the logic and the source language will not only simplify the task of writing and reasoning about properties but also give opportunity to mechanical instrumentation.

We claim that this goal can be achieved by augmenting the EAGLE language with a simple construct that allows one to bind data values from parameterized program events. This construct is the *event expression* and has been mainly influenced by aspect languages [13] and process algebras [17].

It is worth mentioning that we do not address in this paper the important problem of runtime overhead introduced by monitors. This is an orthogonal problem that requires further investigation.

We present related work in the following section. In section 3 EAGLE is described. Section 4 presents HAWK as a specialization of EAGLE for JAVA. It extends EAGLE with event expressions. This section describes the tool and the syntax of the language, presents monitor examples, and then discusses implementation. Section 5 concludes the paper.

2. Related Work

There is currently an increasing amount of work being performed in this field, as documented for example in [1]. Here we shall only mention part of this work. At close range, HAWK is built on top of EAGLE. EAGLE [4] is a language-independent runtime verification tool and logic. It requires the user to create a projection of the actual program state that is being monitored. User-defined formulae are evaluated with respect to this projected state. The EAGLE language essentially extends the μ -calculus with data parameterization and past time logic. HAWK supports automated instrumentation and object reasoning at the expense of making the language specific to Java. The notation and semantics of the data-binding construct is similar to those used in modal logics like the π -calculus [17]. These works influenced the integration of programming and logic as well as the notation and semantics of event expressions.

JAVA MAC [14] defines an event-based language to describe monitors. It is comprised of two specification languages, PEDL and MEDL. The first is tightly integrated to the programming language and defines events that might occur during the program execution. A MEDL specification, on the other hand, makes use of these events in order to state high-level requirements. MAC also supports the declaration variables of primitive types, that can be updated by user-defined assignment statements upon arrival of new events. These variables can be referred to in formulae. HAWK can

define MAC properties as rules, and variables can be cast as either variables in the projected EAGLE state, or as formal parameters in rules following a functional programming paradigm. In contrast to MAC that also allows variable accesses as primitive events, HAWK event expressions currently only concern method calls and returns. However, the HAWK event construct is designed to be extensible so to allow one to reason about other program events. In addition, HAWK supports data binding and object reasoning which we believe to be an essential feature of object-oriented program monitoring. To the best of our knowledge MAC does not support them. A newer version of JAVA MAC supports extended regular expressions. HAWK also supports such.

JASS [5] is a JAVA tool providing a trace-assertion checker in addition to a language for describing pre and post conditions for methods, loop variants (used to assure loop termination) and invariants, and class invariants which are predicates about the state of objects of a particular class. These are defined in a similar fashion as in Eiffel [2] which follows a design-by-contract methodology. The JASS sub-language of trace-assertions is similar to CSP and is strongly related to the logic we present in this paper. Trace assertions are defined as class invariants in the form of annotations in the class file. The distinction between JASS and HAWK is essentially that the first is a process algebraic “programming” language while the latter is a logic. For example, HAWK does not provide built-in operators for external choice, parallel composition and hiding.

TEMPORAL ROVER [9] is a commercial tool that allows the user to specify future and past time metric temporal logic requirements to be checked during runtime. Also, data can be captured and related over a trace. Furthermore, a recent interesting extension combines temporal logic with state charts, allowing edges/transitions in state charts to be guarded by temporal conditions. EAGLE also allows these features, but tries to do so using an integrated single notation rather than the combination of several notations. The user of TEMPORAL ROVER needs to manually instrument the program in order to emit events to the checker. In contrast, HAWK provides automated program instrumentation.

MOP [6] is a methodology and framework for building program monitors. In MOP the crafting of a monitoring tool is divided into building a logic engine and a logic plugin. The first is concerned with generating a software artifact that will check the trace. The later is concerned with the integration of the target program and the logic engine. Instrumentation and IDE integration are supported by the engine. Several plugins have been created in this line already including those for extended regular expressions and linear temporal logic. We believe HAWK could be defined in MOP as well.

Aspect Oriented Programming (AOP) [12] is a software development technique aiming at increasing modularity of orthogonal programming concerns. An aspect is a module that characterizes the behavior of “cross-cutting concerns”. It defines behavior that cross-cuts different abstractions of a program, avoiding scattering code that is related to a single concept at multiple places of the program, and as a consequence, protecting the encapsulation of modules. To some extent AOP can be seen as just a clean solution to the instrumentation of programs. For this purpose, we used extensively the ASPECTJ AOP tool. We believe, however, that a natural extension of this work is the introduction of temporal advices, which could be integrated in an AOP tool. In contrast to the usual aspect advices, temporal advices can provide a means to define hooks for code to be executed upon validation or violation of a finite-trace requirement.

3. The EAGLE Logic

This section serves to give some background on the finite-trace monitoring logic EAGLE.

EAGLE offers a succinct but powerful set of primitives, essentially supporting recursive parameterized equations, with a minimal or maximal fix-point semantics together with three temporal operators: next-time, previous-time, and concatenation. The next-time and previous-time operators can be used for defining future and past time logics on top of EAGLE. The concatenation operator can be used to define interval logics and extended regular expressions. Rules can be parameterized with formulas and data, which allows the definition of new combinators and contexts to be captured in different points in time.

Atomic propositions are boolean expressions over a *user-defined Java object* denoting the current state of the program. This state is a projection of the actual state being monitored and the user has to provide this mapping. This decision allows one to monitor programs written in different languages with reduced effort. That is, one needs to define such a state object in JAVA, which is the EAGLE implementation language, and send events to it from the application being monitored in order to keep it updated. The logic is introduced informally by means of two examples.

3.1. EAGLE by example

This sub-section is a modification of [4], of which the second author is a co-author.

Assume we want to state a property about a program P , which contains the declaration of two integer variables x and y . We want to state that whenever x is positive then eventually y becomes positive. The property can be written as follows in classical future time LTL: $\Box(x > 0 \rightarrow \Diamond y > 0)$. The formulas $\Box F$ (meaning “always F ”) and $\Diamond F$ (meaning “eventually F ”), for some property F , usually satisfy the following congruences [16], where the temporal operator $\bigcirc F$ stands for *next* F (meaning “in next state F ”):

$$\Box F \equiv F \wedge \bigcirc(\Box F) \quad \Diamond F \equiv F \vee \bigcirc(\Diamond F)$$

One can, for example, show that $\Box F$ is a solution to the recursive equation $X = F \wedge \bigcirc X$; in fact it is the maximal solution¹. A fundamental idea in EAGLE is to support this kind of recursive definition, and to enable users to define their own temporal combinators using equations similar to those above. In this framework one can write the following definitions for the combinators *Always* and *Eventually*, and the formula to be monitored (M_1):

$$\begin{aligned} \max \text{Always}(\text{Term } F) &= F \wedge \bigcirc \text{Always}(F) \\ \min \text{Eventually}(\text{Term } F) &= F \vee \bigcirc \text{Eventually}(F) \\ \text{mon } M_1 &= \text{Always}(x > 0 \rightarrow \text{Eventually}(y > 0)) \end{aligned}$$

The *Always* operator is defined as having a maximal fix-point interpretation. That is, if by the end of the trace the property was not yet violated it is assumed to be validated and will evaluate to true. On the other hand, the *Eventually* operator is defined as having a minimal interpretation. If by the end of the trace the formula was not yet validated the eventuality is considered violated and will evaluate to false. Put differently, maximal rules define safety properties (nothing bad ever happens), while minimal rules define liveness properties (something good eventually happens). In EAGLE, the difference only becomes important when evaluating formulas at the boundaries of a trace.

¹Similarly, $\Diamond F$ is a *minimal* solution to the recursive equation $X = F \vee \bigcirc X$.

For completeness we provide remaining definitions of the future time LTL operators \mathcal{U} (until) and \mathcal{W} (unless) below, and also the past-time operator \mathcal{S} (since) used in an example later on. The logic provides a previous-time operator, which allows us to define such past time rules. Note how *Unless* is defined in terms of other operators. However, it could have been defined recursively.

$$\begin{aligned} \min \text{Until}(\text{Term } F_1, \text{Term } F_2) &= F_2 \vee (F_1 \wedge \bigcirc \text{Until}(F_1, F_2)) \\ \max \text{Unless}(\text{Term } F_1, \text{Term } F_2) &= \text{Until}(F_1, F_2) \vee \text{Always}(F_1) \\ \min \text{Since}(\text{Term } F_1, \text{Term } F_2) &= F_2 \vee (F_1 \wedge \bigcirc \text{Since}(F_1, F_2)) \end{aligned}$$

Data Parameters

We have seen how rules can be parameterized with formulae. Let us modify the above example to include data parameters. Suppose we want to state the property: “whenever at some point $x = k > 0$ for some k , then eventually $y = k$ ”. This can be expressed as follows in quantified LTL: $\Box(x > 0 \rightarrow \exists k.(x = k \wedge \Diamond y = k))$. We use a parameterized rule to state this property, capturing the value of x when $x > 0$ as a rule parameter.

$$\begin{aligned} \min R(\text{int } k) &= \text{Eventually}(y = k) \\ \text{mon } M_2 &= \text{Always}(x > 0 \rightarrow R(x)) \end{aligned}$$

Rule R is parameterized with an integer k , and is instantiated in M_2 when $x > 0$, hence capturing the value of x at that moment. Rule R replaces the existential quantifier. Data parameterization is also used to elegantly model real-time logics. See [4] for more details on EAGLE and how to encode LTL, MTL, etc. in the language. The textual notations for \bigcirc and \bigcirc in EAGLE are respectively $@$ and $\#$.

3.2. The EAGLE tool

EAGLE monitors and rules are specified in a text file. In order to verify the program against the stated properties, the programmer must instrument the application in points affecting any formula in the specification. In the example above, at any place where x and y are updated. In these points, the EAGLE state must be updated and then the formulae verified as Figure 1 shows.

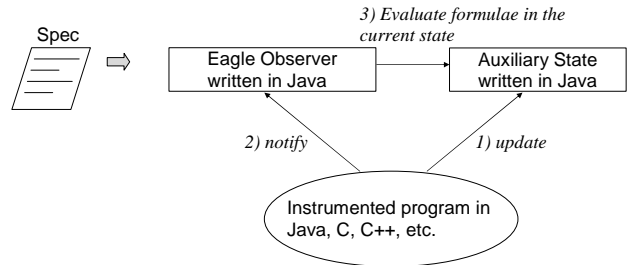


Figure 1: Eagle architecture

When an instrumentation point is hit during program execution, the EAGLE state is updated (1). Then, the observer corresponding to the specified properties (Spec) is notified (2). In response, the observer evaluates the formulae in the current state (3) and derives new obligations for the future which are stored in its internal state.

The observer component “updates” logical requirements upon notification of state updates. It accesses the projected state, as shown by the arrow connecting the boxes, via boolean tests. That is, the observer tests the validity of atomic propositions included in some of its verification rules. In practice, these propositions take the form of (assumed pure) method calls on the projected state and is accomplished via the JAVA reflection mechanism. Note that, these method calls can take parameters since the context in which that method was called (from an EAGLE rule) can be accessed. See, for instance, rule R used in monitor M_2 .

4. HAWK

HAWK is a logic and tool for runtime verification of JAVA programs. It is built on top of EAGLE. HAWK specifications are ultimately translated to EAGLE monitors.

HAWK follows an *event-based* approach to runtime verification in contrast to EAGLE which is *state-based*. It allows one to refer in specification formulae to events that may be emitted by the program. For instance, one can specify that some event must eventually occur after another. In principle, any kind of program event can be defined and used in HAWK formulae. However, currently we support only a limited number, namely, method calls and returns. In addition, these events can be parameterized. It is possible to refer to actual parameters, to the calling thread, and to the return value of an event corresponding to a method call, for instance. As a consequence, HAWK formulae might refer to these values in contexts where they are visible.

The HAWK tool instruments the program in order to track events referred in formulae. In addition, it generates and compiles the EAGLE specification and auxiliary state corresponding to the HAWK specification. In the following, we show the syntax and semantics of HAWK, then two examples of its use, and finally a brief discussion on its implementation.

4.1. Syntax and a Semantics Fragment

The syntax of HAWK in EBNF form follows. The usual symbols $*$, $+$ might be qualified with a terminal denoting a separator:

<i>Observer</i>	::=	<u>observer</u> <i>Id</i> { <i>Vardecl</i> *; <i>Monitor</i> + }
<i>Vardecl</i>	::=	<u>var</u> <i>Id</i> <i>Id</i>
<i>Monitor</i>	::=	<u>mon</u> <i>Id</i> \equiv <i>Hawk_proposition</i> $_$
<i>Hawk_proposition</i>	::=	\langle <i>Event</i> \rangle <i>Hawk_proposition</i> \llbracket <i>Event</i> \rrbracket <i>Hawk_proposition</i> "Java boolean expression" "Eagle formula extended with Hawk propositions"
<i>Event</i>	::=	\llbracket <i>IdQ</i> $_$ \rrbracket <i>Method_expression</i> \llbracket <u>returns</u> \llbracket <i>IdQ</i> \rrbracket
<i>Method_expression</i>	::=	<i>IdQ</i> $_$ <i>Id</i> (<i>IdQ</i> *)
<i>IdQ</i>	::=	"Java identifier" \llbracket $_$ \rrbracket $_$ *

A HAWK specification is given in Figure 2. The initial (root) syntactic category of this grammar is *Observer*. A HAWK specification is thus an observer comprised of possibly many monitors. Event expressions correspond to the first two productions of the syntactic category *Hawk_proposition*. That is, they can take either forms:

\langle event \rangle proposition \llbracket event \rrbracket proposition

Here, event corresponds to a method being called or returned from and may bind free variables in proposition. Question marks are used in event to extend the environment with such new bindings. For instance, an event expression of the form:

\langle o?.method-name(p?) returns r? \rangle P

can bind variables o , p , and r in the scope of P . So the HAWK proposition P can use these objects in its definition, possibly involving temporal operators and relating other objects captured in the body of P .

The construct \langle $_$ \rangle has a conjunctive semantics. It means the entire proposition will evaluate to true iff the associated event occurs *and* the proposition argument evaluates to true in the current state.

The construct \llbracket $_$ \rrbracket , on the other hand, has an implicative semantics meaning that the proposition will only be checked *if* the event denoted by event occurs. Therefore, \langle event \rangle true means that event must occur while \llbracket event \rrbracket false means that it cannot occur.

More formally, let the semantic function for HAWK propositions have the following signature:

\llbracket $_$ \rrbracket : *Hawk_proposition* \rightarrow (*Id* \rightarrow *Object*) \rightarrow *Hawk_proposition*

It takes a formula, a state, and returns a new formula, being true, false, or a new temporal proposition that has to be satisfied in the next state. The semantic function is defined as follows for the \langle $_$ \rangle -construct:

$$\llbracket \langle \text{ev} \rangle \text{prop} \rrbracket \Sigma = \begin{cases} \text{false} & \text{if } \llbracket \text{ev} \rrbracket \Sigma = \text{false} \\ \llbracket \text{prop} \rrbracket \Sigma' & , \text{otherwise} \end{cases}$$

and for the \llbracket $_$ \rrbracket -construct:

$$\llbracket \llbracket \text{ev} \rrbracket \text{prop} \rrbracket \Sigma = \begin{cases} \text{true} & \text{if } \llbracket \text{ev} \rrbracket \Sigma = \text{false} \\ \llbracket \text{prop} \rrbracket \Sigma' & , \text{otherwise} \end{cases}$$

for all events *ev*, HAWK propositions *prop*, and environment maps Σ . The semantic function on events returns a boolean denoting if the event has occurred or not in the current state. Note that the generation of events from the run of a program leads to a notion of pattern-matching of variables in our specifications. In particular, the state² Σ' denotes the extension of Σ with the binding from variables, occurring in the pattern matching of *ev*, to their corresponding program values. Therefore, in addition to only returning a boolean the semantic function of *Event* should also return an environment for the match which captures the variables in the corresponding event. Due to this we say these events are *parameterized*. We omitted the definition of this function for the sake of brevity.

4.2. HAWK by Example

We show in the following examples of *liveness* and *safety* [15] properties specified as HAWK specifications. Intuitively, a liveness property states that something good must eventually happen while a safety property states what should never happen. It is perhaps surprising that liveness properties can be monitorable [8]. This is because eventualities may not hold in a finite trace and still hold in the future. The finite trace semantics of EAGLE, however, makes a simplifying assumption that if the program terminates, and an eventuality has not been satisfied, the system will emit a warning even though the eventuality perhaps could be satisfied if the program continued.

Temporal Buffer Requirements

Figure 2 illustrates the format of a logic observer specification in HAWK.

We omitted in the syntax the declaration of configuration attributes associated to the compilation, distribution, and execution of the observer. Here we declare `classpath`, `targetPath`, and `termina`

²We use the terms environment and state indistinctly in this definition.

```

observer BufferObserver {
  classPath = C:/downloads/src
  targetPath = C:/downloads/src
  terminationMethod = bufferexample.Barrier.end()

  var Buffer b ;
  var Object o ;
  var Object k ;

  mon B =
    Always ( [b?.put(o?)]
      Eventually ( <b.get() returns k?> (o == k) ) ) .
}

```

Figure 2: HAWK observer for Buffer requirement

tionMethod. The first serves to access the class definitions associated to the types of the declared variables. In this case, Buffer. The attribute targetPath indicates where the generated files will be stored. Finally, terminationMethod indicates a static method that will trigger the end of the monitoring session. This method must be called from the program to allow monitors to finish. Variables are typed and can be used in monitor definitions.

In this figure, monitor B states a property that whenever an object o is inserted into a buffer b, eventually it is taken out from that buffer. Note that this specification is not sensible to duplicates. That is, it will be satisfied if the trace consists only of two put on the same object and only one later get on that object. The eventuality of B can be alternatively expressed as:

$\langle b.get() \text{ returns } o \rangle \text{ true.}$

Note that we were able to capture the actual parameter of the method put and the return of method get. In addition, we assume events to be disjoint – they do not occur simultaneously. In other words, events have an interleaving semantics.

Strict Alternation in Acquisition and Release of Locks

The monitors F1 and F2 in Figure 3 state properties about the way locks are acquired and released via the methods acquireLock and releaseLock, which can be seen as part of a user-defined thread synchronization JAVA package for a File System implementation. We want to check if the file system correctly follows the lock protocol.

F1 states that there should not be an acquire of a lock without a future release by the same thread, and no other acquisitions by any thread can occur in between. F1 states the dual property about lock releases. The term “t?:” qualifies the event description with the thread from which the event was sent. Note that these properties are more restrictive than those of Java’s standard reentrant locking discipline, which allows nested lock acquisitions.

4.3. The compiler

The compilation of the specification shown in Figure 2 produces the following EAGLE monitor and rules:

```

observer FileSystemObserver {
  ...
  var Thread t ;
  var FileSystem fs ;
  var int l ;

  mon F1 =
    Always ([t?:fs?.acquireLock(l?) returns]
      @ ( Until( [*:fs.acquireLock(l) returns]false,
        <t:fs.releaseLock(l)>true))) .

  mon F2 =
    Always ( [t?:fs?.releaseLock(l?)]
      # ( Since( [*:fs.releaseLock(l)]false ,
        <t:fs.acquireLock(l) returns>true))) .
}

```

Figure 3: HAWK observer for Lock requirement

```

max R2(Object o, Object k) = compare_references(o,k) .
max R1(Object b, Object o) =
  Eventually(get_(b)^R2(o,getValue(ht,`return`))) .
mon B = Always(put_(() →
  R1(getValue(ht,`caller`),getValue(ht,`arg1`))) .

```

The methods get_, put_, and compare_references are declared in the EAGLE state. The variable ht is a hash table stored in the state and carried over to access the parameters of the last event.

We show below a fragment of the ASPECTJ aspect the compiler generates. This aspect is in charge of instrumenting the program to track the events and call the observer:

```

public aspect BufferObserverAspect {

  BufferObserverState state = new BufferObserverState();
  Observer observer =
    new Observer(RuleBase.parse(...spec-file));
  Object lock = new Object();

  pointcut put_(bufferexample.Buffer caller, Object arg0) :
    target(caller) && args(arg0) &&
    execution(* bufferexample.Buffer.put(Object));

  before(bufferexample.Buffer caller, Object arg0) returning :
  put_(caller, arg0){
    synchronized (lock) {
      MethodCall mcall = new MethodCall("caller", caller,
        new EagleMethod("bufferexample.Buffer", "put",
          new String[]{"Object"}));
      mcall.addActualParameter("arg0", arg0);
      state.setCurrentEvent(mcall);
      state.eventMessage();
      observer.handle(state);
    }
  }
  ...
}

```

This aspect includes a before execution advice on the method put of class Buffer. When this advice is triggered we create a representation of the call and update the EAGLE state denoted by an instance of the class BufferObserverState. The observer is notified in the sequence with a call to the method handle.

The generated files corresponding to the Buffer specification are presented in Appendix A.

4.4. Implementation

A parser for HAWK was built using JLEX and JAVA CUP. The tool has 6500 lines of Java source code. The compiler works by transforming HAWK sentences written in a specification file into equivalent EAGLE specifications, and AspectJ aspects.

The user must provide the name of a method - `terminationMethod` - that must be called when the program terminates. HAWK tracks this call and informs EAGLE to finish observation. This is necessary to determine whether eventualities occurring logic formulae hold.

During program execution the generated EAGLE state contains information about the most recent event emitted which is also declared in the specification. We create ASPECTJ aspects [13] to track events that occur in the formulae and update this state.

Methods declared in the state check if an event has occurred. These methods are called from the observer to decide if the state satisfies the (event) guard of an event expression. In practice, whenever a program point of interest is hit, the EAGLE state gets updated and the formulae are checked by the observer.

5. Conclusion

We described a logic and tool, HAWK, that generates observers to monitor temporal properties of JAVA programs. The tool translates HAWK specifications into EAGLE. The contribution of this work is twofold. It integrates a very powerful temporal logic with automated aspect oriented program instrumentation. Second, the logic HAWK allows to state properties about Java objects. Further work includes enhancing the tool to support additional events, eventually all events supported by ASPECTJ.

In the current implementation ASPECTJ is hidden under the surface to support the connection to Java. One can, however, consider an even tighter integration of a system like Eagle with an AOP system like AspectJ by supporting temporal cutpoints: temporal EAGLE formulae now become part of the ASPECTJ cutpoint language, and can function as triggers for actions to be executed. This can be used for developing fault tolerant programs that can change behavior when temporal properties are violated.

6. References

- [1] *1st, 2nd, 3rd, and 4th CAV Workshops on Runtime Verification (RV'01 - RV'04)*, volume 55(2), 70(4), 89(2), 113 of *ENTCS*. Elsevier Science: 2001, 2002, 2003, 2004.
- [2] Eiffel language, 2005. <http://www.eiffel.com/>.
- [3] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with Test Case Generation and Runtime Analysis. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines (ASM'03)*, volume 2589 of *LNCS*, pages 87–107. Springer, March 2003.
- [4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 55(2), 70(4), 89(2) of *LNCS*, Venice, Italy, Jan 2004. Springer.
- [5] D. Bartetzko, C. Fisher, M. Moller, and H. Wehrheim. Jass - Java with Assertions. In K. Havelund and G. Roşu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01)*, volume 55 of *ENTCS*, Paris, France, 2001. Elsevier Science.
- [6] F. Chen, M. d'Amorim, and G. Roşu. A Formal Monitoring-Based Framework for Software Development and Analysis. In *Proceedings of ICFEM'04*, volume 3308 of *LNCS*, pages 357–372, 2004.
- [7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [8] M. d'Amorim and G. Roşu. Efficient Monitoring of ω -languages. to appear in *Computer Aided Verification (CAV'05)*.
- [9] D. Drusinsky. The Temporal Rover and the ATG Rover. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000.
- [10] D. Gabbay. The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In *Proceedings of the 1st Conference on Temporal Logic in Specification, Altrincham, April 1987*, volume 398 of *LNCS*, pages 409–448, 1989.
- [11] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop on Runtime Verification (RV'01)* [1], pages 97–114. Extended version to appear in the journal: *Formal Methods in System Design*, Kluwer, 2004.
- [12] G. Kiczales and et al. Aspect-Oriented Programming. In *ECOOP*, volume 1241. Springer-Verlag, 1997.
- [13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th ECOOP*, Lecture Notes in Computer Science, pages 327–353. Springer-Verlag, 2001.
- [14] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a Run-time Assurance Tool for Java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *ENTCS*. Elsevier Science, 2001.
- [15] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
- [16] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, 1995.
- [17] R. Milner. *Communicating and Mobile Systems: The π -Calculus*. Cambridge University Press, New York, 1992.
- [18] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

Appendix A: Generated Files

We show the complete output produced from an input specification for the Buffer example shown in Figure 2.

Source HAWK specification

```
max Always(Term t) = t /\ @ Always(t) .
min Eventually(Term t) = t \/ @ Eventually(t) .

observer BufferObserver {
  classPath = C:/tests/eaglepp
  targetPath = C:/tests/eaglepp
  terminationMethod = bufferexample.Barrier.end()
  var bufferexample.Buffer b ;
  var Object o ;
  var Object k ;
  mon B = Always( [b?.put(o?)
                  Eventually (
                    <b.get() returns k?> ( o == k ) ) ] ) .
}
```

Generated EAGLE Specification

```
max Always(Term t) = t /\ @ Always(t) .
min Eventually(Term t) = t \/ @ Eventually(t) .
max r_5(Object o, Object k) = m_7(htable, o, k) .
max r_1(Object o, Object b) =
  Eventually(m_4(htable, b) /\
             r_5(o, getValue(htable,c6) )) .
mon B = Always(m_0(htable) ->
              r_1(getValue(htable,c2) ,
                  getValue(htable,c3) )) .
```

Generated Instrumentation Aspects

```
package monitors; import ...

public aspect BufferObserverAspect {

  BufferObserverState state = new BufferObserverState();
  Observer observer =
    new Observer(RuleBase.parse(
      "C:/tests/eaglepp/bufferexample/buffer_compiled.spec"));
  Object lock = new Object();

  pointcut put_ ( bufferexample.Buffer caller , Object arg0 ) :
    target(caller) && args(arg0) &&
    execution(* bufferexample.Buffer.put(Object));

  before ( bufferexample.Buffer caller , Object arg0) returning :
  put_ ( caller , arg0 ){
    synchronized (lock) {
      MethodCall mcall = new MethodCall("caller", caller,
        new EagleMethod(
          "bufferexample.Buffer","put", new String[]{"Object"}));
      mcall.addActualParameter("arg0",arg0 );
      state.setCurrentEvent(mcall);
      state.eventMessage();
      observer.handle(state);
    }
  }

  pointcut get_ ( bufferexample.Buffer caller ) :
    target(caller) && execution(* bufferexample.Buffer.get() );

  before ( bufferexample.Buffer caller ) returning (Object result) :
  get_ ( caller ){
    synchronized(lock) {
      MethodReturn mret = new MethodReturn(caller, new
        EagleMethod(
          "bufferexample.Buffer","get", new String[]{} ) , result );
      state.setCurrentEvent(mret);
      state.eventMessage();
      observer.handle(state);
    }
  }
}
```

```
pointcut end_ () : call(* bufferexample.Barrier.end(..));

before() : end_(){
  state.terminate();
  observer.end();
}
}
```

Generated EAGLE State

```
package monitors;

import eaglepp.*; import java.util.*; import java.io.*;

public class BufferObserverState extends EaglePPState {

  public static boolean m_0(Hashtable htable) {
    return (((String)getValue(htable,"methodName")!=null &&
      ((String)getValue(htable,"methodName").equals("put")) &&
      (((String)getValue(htable,"targetType")!=null &&
      ((String)getValue(htable,"targetType").equals(
        "bufferexample.Buffer"))));

  public static boolean m_4(Hashtable htable, bufferexample.Buffer b) {
    return (((String)getValue(htable,"methodName")!=null &&
      ((String)getValue(htable,"methodName").equals("get")) &&
      (((String)getValue(htable,"targetType")!=null &&
      ((String)getValue(htable,"targetType").equals(
        "bufferexample.Buffer")) &&
      ( getValue(htable,"caller") == b );)

  public static boolean m_7(Hashtable htable, Object o, Object k) {
    return (o == k) ; }

  public static final String c2 = "arg0";
  public static final String c3 = "caller";
  public static final String c6 = "retObject";
  private static File logFile =
    new File("bufferexample/errors.BufferObserverState");
  private static StringBuffer errorMessages = new StringBuffer();
  private static StringBuffer errorWarningMonitors = new StringBuffer();
  private static StringBuffer warningMessages = new StringBuffer();

  public void eventMessage() {
    errorWarningMonitors.append(printEventAsString()+"\n");
  }

  public void error(String args) {
    errorWarningMonitors.append("error: " + args + " was violated\n");
    errorMessages.append("error: " + args + " was violated\n");
  }

  public void warning(String args) {
    errorWarningMonitors.append("warning : monitor " + args +
      " was not validated.\n");
    warningMessages.append("warning : monitor " + args +
      " was not validated.\n");
  }

  public static void terminate() {
    System.out.println("-----");
    System.out.println("SUMMARY FOR MONITORS");
    if (errorMessages.length(>0) {
      System.out.println(errorMessages.toString());
    } else {
      System.out.println(" no violation");
    }
    if (warningMessages.length(>0) {
      System.out.println(warningMessages.toString());
    } else {
      System.out.println(" eventualities validated");
    }
    System.out.println("-----");
    try {
      PrintWriter pwriter = new PrintWriter(new FileWriter(logFile));
      pwriter.print(errorWarningMonitors.toString());
      pwriter.flush();
      pwriter.close();
    } catch (IOException ioException) {
      System.err.println("Could not write to the file."); } } }
```