

# Dynamic Analysis of Java Applications for MultiThreaded Antipatterns

S. Boroday  
CRIM

550 Sherbrooke West, 100  
Montreal, Canada  
1 (514) 840-1234

sboroday@crim.ca

A. Petrenko  
CRIM

550 Sherbrooke West, 100  
Montreal, Canada  
1 (514) 840-1234

apetrenk@crim.ca

J. Singh

Concordia University  
1455 Maisonneuve West  
Montreal, Canada  
+1 (514) 848-2424

j\_singh@ece.  
concordia.ca

H. Hallal  
CRIM

550 Sherbrooke West, 100  
Montreal, Canada  
1 (514) 840-1234

hhallal@crim.ca

## ABSTRACT

Formal verification is not always applicable to large industrial software systems due to scalability issues and difficulties in formal model and requirements specification. The scalability and model derivation problems could be alleviated by runtime trace analysis, which combines both testing and formal verification. We implement and compare an ad-hoc custom approach and a formal approach to detect common bug patterns in multithreaded Java software. We use the tracing platform of the Eclipse IDE and state-of-the-art model checker Spin.

## Categories and Subject Descriptors

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—*parallel programming*; D.2.4 [Software Engineering]: Software/Program Verification—*model checking*; D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids*; *monitors*; *tracing*; D.4.1 [Operating Systems]: Process Management—*concurrency*; *deadlock*; *multiprocessing/multiprogramming*; *synchronization*.

## General Terms

Algorithms, Reliability, Experimentation, Languages, Verification.

## Keywords

Antipatterns, bug patterns, Java, bytecode, instrumentation, multithreading.

## 1. INTRODUCTION

### 1.1 Multithreading

Java is a highly popular language that exploits the power of distributed computing and allows developing applications that run on different kinds of computer systems and other devices [27]. Among the strong features of Java is its support of multithreading.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop on Dynamic Analysis (WODA 2005) 17 May 2005, St. Louis, MO, USA.. Copyright 2005 ISBN # 1-59593-126-0 17/05/05 ... \$5.00

Java is the first mainstream language to support multithreading on the level of the language itself, rather than with external libraries. Threads are “lightweight” processes, which run concurrently but share common resources. Multithreading is a convenient way to decompose large programs into smaller tasks, so it adds to program organization. At the same time, it increases the overall efficiency; in particular, on truly multiprocessor or hyperthreaded architectures. Multithreading is heavily used in Java programming, especially on the server side, where one program must process multiple concurrent requests from numerous clients.

Being one of the most attractive features of Java, multithreading happens to be the source of difficulty not only for novice but also for experienced programmers. Unlike sequential programs, a multithreaded program can have several execution paths for the same input data because of the unpredictable thread scheduling and the inherent parallelism in the behavior of threads. Errors could occur only on some executions, which makes them hard to detect. Also, it may be difficult to reproduce such errors. Even mature programmers, including authors of textbooks and research papers, could make mistakes in multithreaded programs. For example, the widely cited and used double check design pattern is shown to be error prone in Java both theoretically and experimentally [3]. In order to enable efficient compiler implementation on different systems, Java specifications provide a liberal and weak memory model. Thus tricks, which work for certain languages and environments with a coherent memory model, could cause problems in Java applications, especially when executed on a multiprocessor system and compiled with a modern optimizing Java compiler. Even access to “atomic” data could result in data incoherence if the data was not protected appropriately.

Since multithreading errors are difficult to avoid, a practical means to ensure the quality of multithreaded applications is to apply the automatic bug detection. In this work, we implement and compare two, promising approaches to multithreaded bug detection: the first is based on custom detection of multithreaded errors, and the second relies on the theory of model checking. Both approaches target the dynamic analysis of multithreaded programs. The resulting tools analyze traces collected by the quality assurance framework of the popular open-source Java development platform Eclipse [6].

## 1.2 Antipattern: Dark Side of Patterns

The patterns of erroneous code, design, or behavior often repeated in various applications, are known as antipatterns [8]. So far, 38 most often cited antipatterns that relate to multithreading problems are documented [8]. While many of documented antipatterns are easy to detect by static analysis [8], many others are more naturally addressed by dynamic analysis. Several antipatterns relate to the well-known classical problems of data race, deadlock, and livelock. Some antipatterns are related to the specifics of the weak memory model of Java, such as the infamous double checking antipattern [3], spin wait, or unprotected “atomic” variables. Others, such as double *start()* call, are Java specific. Antipatterns are not solely restricted to correctness problems, but also reflect inefficiencies and poor programming styles. While the notion of antipattern originates from design pattern; here, we use it in a broader sense: covering error and bug patterns. Bug patterns [1] refer to behavior patterns which are associated with underlying bugs and are closer to our case. However, we do not restrict ourselves to bug detection and would like to cover various deficiencies.

## 1.3 Locating the Problems

Various approaches and tools are used to detect multithreading problems [12]. Static approaches require only source or compiled code, while dynamic approaches require actual execution of the compiled code. Static approaches are more useful, since code could be analyzed in parallel with development, ideally while the code is typed. A drawback of static approaches is over-abstraction, which sometimes results in an overwhelming number of false alarms [8]. For example, most of the available static tools perform analysis on the level of classes, rather than objects; in this case, some antipatterns such as double start, could not be detected reliably [8]. On the contrary, with dynamic approaches, objects are visible. So, sophisticated methods of dynamic analysis, based on partial order analysis or formal methods are capable of detecting potential problems even if they do not appear explicitly in the recorded execution [25]. However, a dynamic approach cannot infer all the possible executions of the program. In the case of some classic problems, such as deadlock and race condition detection, efficient predictive algorithms with a reasonable false alarm rate, such as Eraser [24], are developed and successfully implemented in Java analyzers like JProbe [20] and VisualThreads [13]. A promising approach is based on model-checking [4], as it could efficiently analyze relatively complex models for deadlocks, livelocks, and other properties defined in a temporal logic. Currently, few model checkers, such as NASA’s experimental Java Path Finder 2 [11] with a home-grown Java Virtual Machine, are capable to analyze the Java code directly. To alleviate the difficulties associated with model formalization and ensure the scalability, two related approaches emerged: automatic model extraction and runtime analysis of execution traces. The majority of runtime tools detect only apparent non-multithreaded bugs, but more advanced tools could detect multithreaded problems using causality and partial order techniques [20], classical distributed trace analysis [16], and dedicated algorithms for detection of classical multithreaded problems, such as Eraser [24]. While there is some research on comparison and experimental assessment of various static analyzers and

methodologies [22], [7], rather little has been done in comparing and benchmarking dynamic methods and tools [12].

Here we address runtime analysis of Java application; we present our preliminary results in the implementation -in the context of popular Eclipse [6] tracing platform Hyades- and comparative analysis of two promising approaches: ad-hoc custom analysis and model checking of the formal model of a collected trace. Probably, the closest research on the analysis of multithreaded Java applications is related to the JMPaX analyzer [25], which adapts a classical distributed trace analysis framework, where only events that happen on the same thread or object are ordered. Unlike this work, we are not concerned with functional user-defined temporal properties on variable values, but with antipattern detection; this results in a slightly different model of a trace and a less heavy instrumentation. Also we opt for post-mortem analysis, which reduces overhead on the application and allows the use of an off-the-shelf model checker Spin. Detection of certain multithreading antipatterns is sensitive to the order in which events occur in the execution, while for others it is not. Some antipatterns are thread-local. Our custom detectors do not address the problem of distortions in the event order by a tracing tool, while in many cases it is possible to do so, e.g., by analyzing causal dependencies.

The paper is organized as follows. In Section 2, workflows and descriptions of the custom and formal approaches to antipattern detection are presented. Java trace collection and instrumentation are explained in Section 3. The formalization of antipatterns in temporal logic is discussed in Section 4. Section 5 presents our implementation of the custom approach. Implementation of the model checking approach is discussed in Section 6. Experimental evaluation of the resulting tools is reported in Section 7. Section 8 concludes the paper with a summary of results and future work.

## 2. THE WORKFLOWS OF THE MODEL CHECKING AND CUSTOM APPROACHES

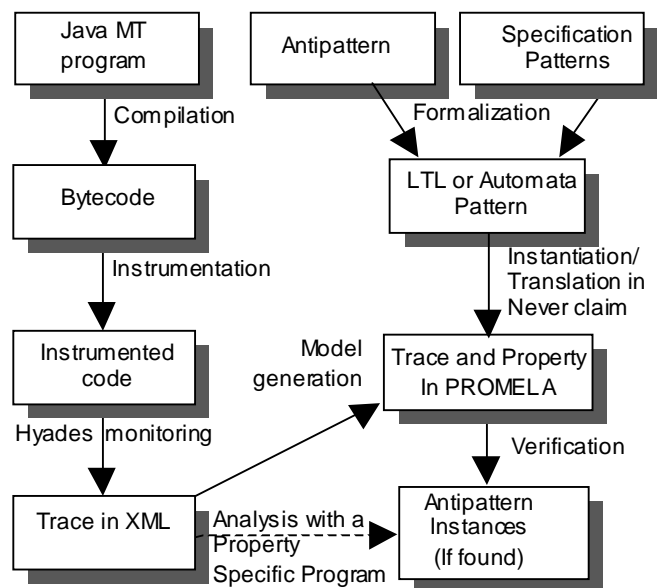


Figure 1. The workflows of the two approaches

To collect the distributed traces, we use Hyades [15], the monitoring framework of Eclipse, enhanced with our bytecode instrumentation tool to handle more thread related events. Analysis is later performed either with custom detectors or using the model checker Spin, which is one of the most popular, mature, and efficient open-source verification tools [14].

In the model checker based trace analysis approach (Figure 1), an execution trace obtained from the Hyades framework is translated into a PROMELA model (a PROMELA model is not needed in the case of the custom detection approach as indicated by the dashed arrow in Figure 1). The PROMELA model is verified using Spin against the multithreaded antipatterns specified in the linear temporal logic, LTL [14].

In the custom detection approach, the antipatterns are coded in Java. Formal methods could still be used here, e.g., LTL or automata formalization of antipatterns could help the programmer better understand antipatterns. We implement the automata descriptions in Java manually, though; automated code generation is possible [2]. In the future, we may also investigate the possibility of using UML Java code generators.

## 3. TRACE COLLECTION

### 3.1 Instrumentation Levels

Instrumentation, which is required to obtain an execution trace, could be performed at the level of:

- source code,
- bytecode,
- virtual machine (including standardized instrumentations such as debugging (JVMDI) and profiling (JVMPI) interfaces[27] ), or
- operating system.

Regardless of the instrumentation used, most public Java tracing tools are limited mainly to recording method invocations. However, such information is rarely sufficient, and better means are needed to instrument the application. Thus, we propose to reuse an existing industrial-strength method level tracing tool (Hyades) and enhance it with a light-weight instrumentation technique. Unlike usual full-scale bytecode instrumentation, there is often no need to add code that actually logs events, it is sufficient to insert designated method calls near events of interest. In many cases, such method could be dummy, thus creating minimal probe effect and overhead. Note that it is always possible to develop a home-grown tracing tool from scratch, where the instrumentation could depend on sophisticated methods of causality tracking [16]. However, since the foundations for theoretically sound instrumentation [23] are not yet completely established, heavy and full-scale code instrumentation is a rather tedious and error-prone task.

Hyades provides a JVMPI agent that records the most basic (but not all) events, such as data allocation and method invocation. In order to record all the relevant events, we instrument the bytecode of the application with calls to the special dummy “marking method” near important events such as lock acquisition and

release, variable access and update. While Hyades collects method calls of the instrumented applications, some of the instrumented method calls indicate operations with locks and variables. With our complimentary instrumentation, Hyades is capable of collecting traces detailed enough to detect several multithreaded problems.

### 3.2 Hyades Tracing

Hyades [15] quality assurance platform comes with a reference JVMPI (Java Virtual Machine Profiling Interface) agent for tracing. While Hyades tracing, based on an experimental profiling interface, is not free from both overhead and correctness problems [27], [15], our choice was motivated by the following:

1. Hyades is a tool within the popular Java IDE Eclipse.
2. Hyades is a modular open source platform with various reference tools which could be modified or enhanced.
3. While current Hyades data collection engine is based on Java 1.4 experimental JVMPI instrumentation, upcoming versions of Hyades are likely to support modern Java features, including new profiling interfaces, such as JVMTI.
4. Events are well defined and could be exported into XML for online and post-mortem analysis.
5. In Hyades, events can be filtered on the level of packages, classes and method names. The framework also provides control for the level of details and other parameters.
6. Hyades provides a unified framework for quality assurance tools with a standardized data system and user interface. Traces are visualized with Class Interaction Diagrams (Sequence Charts) of UML 2.0. The visualization features of Hyades and Eclipse graphical tools could be used to improve traceability of the results.
7. There is work in progress adding new features to Hyades (e.g., variable value collection) [15].

Events are represented in XMI (XML Metadata Interchange) format. The main events are declarations of classes and methods, allocations of objects, and method invocations. Each thread, class, method, and object is assigned an integer identifier and referenced by this identifier, rather than with a full name. Each method entry/exit event contains various attributes such as references to identifiers of thread, object, and timestamp, etc. With a proper configuration of Hyades, the code line number related to events can be logged to enable traceability of the problem in the source code.

### 3.3 Enhancing Hyades

Detection of certain multithreading related antipatterns requires not only information on the performed method calls, but also on lock acquisition and variable access. To collect all the relevant events in the trace, we perform the following instrumentations.

We use the JTrek instrumentation package [5] developed at Digital Corporation (now Hewlett-Packard), to build an instrumentation tool to log the events related to the lock entry and exit and variable access. JTrek consists of the Trek class library, which enables Java developers to write Java applications that analyze and modify existing Java class files. JTrek reads Java class files and traverses the abstract syntax trees of a program while examining the content and inserts a new code. The inserted

code can access the contents of the method call-time stack at run time. JTrek was previously used as instrumentation tool in Java-Mac [21] and Java Path Explorer (JPaX), [10].

A thread can access a lock using bytecode instructions *monitorenter* and *monitorexit*. These instructions are not traced by Hyades. To log the lock entry and exit events, associated with these instructions, the *Object* class is instrumented with the empty methods *lockacquired*, *lockreleased*; then invocations of methods of the locked objects are inserted by our instrumentation tool before and after *monitorexit* and *monitorenter* bytecode instructions, respectively. These methods *lockacquired*, *lockreleased* are empty, their sole purpose is to indicate the occurrence of the corresponding events and locked objects. While the method bodies are empty and do not perform any useful tasks, *lockacquired* and *lockreleased* method entries are logged in the XML trace, which indicate locking and unlocking of objects.

Field accesses are identified with the insertion of the *variable\_write* and *variable\_read* methods, which log additional details, such as the variables modified and their values. These additional details are logged into a separate log file. Variable access events are needed mostly for the detection of data races or incoherencies, already targeted by other race detection tools; thus variable access instrumentation could be omitted if the user is not interested in data race detection.

We illustrate the instrumentation process with a sample instrumented bytecode and Hyades trace. The following instrumented bytecode shows first seven instructions of a synchronized method. After locking an object, four instructions (shown in bold) that invoke the *lockacquired* method of this object are inserted.

```

0:  aload_0
1:  getfield    #36 <Field Object[] convey>
4:  ildload_1
5:  aaload
6:  dup
7:  astore_2
8:  monitorenter
9:  aload_0
10: getfield    #36 <Field Object[] convey>
13: ildload_1
14: aaload
15: invokevirtual #130 <Method void
    Object.lockacquired(>

```

Below is a fragment of a trace generated by Hyades showing events relevant to lock entry. The first event associates a method identifier 6532 to the *lockacquired* method. Then, all method entry events with method Id 6532 indicate locking of the object, referenced with the corresponding attribute *ObjectIdRef*.

```

<methodDef
  Name           = "lockacquired"
  Signature      = "()V"
  startLineNumber= "87"
  endLineNumber  = "96"
  methodId       = "6532"
  classIdRef     = "6544" />

```

```

<methodEntry
  threadIdRef    = "2"
  time           = "1078172332.902561700"
  methodIdRef    = "6532"
  objIdRef       = "6562"
  classIdRef     = "6544"
  ticket         = "7630"
  stackDepth     = "3" />

```

```

<methodExit
  threadIdRef    = "2"
  methodIdRef    = "6532"
  objIdRef       = "6562"
  classIdRef     = "6544"
  ticket         = "7630"
  time           = "1078172332.931166600"
  overhead       = "0.000077831" />

```

## 4. ANTIPATTERN FORMALIZATION

We describe the formalization of antipatterns in LTL using the example of the premature *join()* call antipattern. It consists in the invocation of the *join()* method of a thread, which is not yet started [18], [8]. Obviously, it is impossible to specify such a pattern in the classical Propositional LTL independently of the number of threads, which makes writing an LTL formula to describe the antipattern rather difficult and cumbersome. For simplicity, we consider the instantiation of the antipattern for one particular thread  $T_i$ , where *join()* to thread  $T_i$  is called before  $T_i$  starts. Actually, it is more convenient to formalize the absence of the antipattern, so a model checker could pinpoint the problem with a counterexample to the correctness claim. The violation of the absence of premature *join* implies that the application under test exhibits the problem. The formalization of this antipattern requires predicates which indicate invocations of the *join()* method,  $\text{Join}(T_i)$  and *start()* method,  $\text{Start}(T_i)$ . Since specifying in LTL could be difficult for non-logicians, to formalize the absence of premature *join()* call to the thread  $T_i$ , one may use the existing pattern specification system [25]. It includes an online repository of LTL (and other formalisms) patterns that cover the most common specification patterns in many system types. The most relevant pattern is precedence:  $S = \text{Start}(T_i)$  precedes  $P = \text{Join}(T_i)$ , which is mapped into LTL as  $!P \ W \ S = ! \text{Start}(T_i) \ W \ \text{Join}(T_i)$ , where  $W$  is the weak until operator. On a trace that contains  $n$  threads  $T_1, \dots, T_n$  instantiations of the antipattern for each thread could be either checked individually, for each thread  $T_i$ , or at once with all combined into one composite property,  $(! \text{Start}(T_1) \ W \ \text{Join}(T_1)) \ \& \ (! \text{Start}(T_2) \ W \ \text{Join}(T_2)) \ \& \ \dots \ \& \ (! \text{Start}(T_n) \ W \ \text{Join}(T_n))$ .

We find that using one composite property is more convenient, though model checking  $n$  properties one-by-one could provide a better diagnostics: it is immediately clear exactly which thread is involved in premature *join()*. Instantiation of predicates  $\text{Start}(T_i)$  and  $\text{Join}(T_i)$  is implementation dependent.

Similarly, several other antipatterns could be formalized using the specification pattern system. Sometimes, several specification patterns apply. Double *start()* absence could be formalized with “bounded existence” or “absence of  $P = \text{Start}(T_i)$  after  $Q = \text{Start}(T_i)$ ” specification pattern, slightly modified with the next operator  $X$  to represent a left-open “after  $Q$ ” scope.

The antipatterns could be formalized with automata using an LTL to automata transformation tool or directly in an automata specification system, such as in [9]. Automata specifications for these two antipatterns were previously developed for Flavors/Java static analysis tool [18].

## 5. CUSTOM ANTIPATTERN DETECTORS

The most light-weight and flexible approach to antipattern detection is custom implementation of detectors with mainstream programming languages and tools. Such an approach provides flexibility since the programmer is not bound by inconveniences of exotic system specification languages of model checkers. Yet, there is still a possibility to use formal methods. While, partially, this approach is inspired by the so-called state based test oracles [2], we did not develop a particular state based language, instead FSM (finite state machine) or extended [19] FSM diagrams are built simply to capture the meaning of patterns and easy manual coding of the detectors (oracles) in Java. For experimental purposes, we code few Java detectors (namely, double call of *start()* method, premature *join()*, and *wait()* stall). Java Architecture for XML Binding (JAXB) is used to parse the generated trace. Such an architecture does not scale well on very large traces (>25M), but makes possible processing XML data without knowledge of XML itself [27]. This means that while it may not be the most efficient tool, it perfectly suits the purpose of experimenting and prototyping.

## 6. MODEL CHECKING APPROACH

To benefit from the advantages of modern verification technology, we implemented a tool based on the model checker Spin. The trace generated by Hyades is mapped into PROMELA. Each thread is modeled by a PROMELA process. The trace events themselves are translated in a more or less direct way into *d\_step* statements, where each event attribute is modeled by a PROMELA variable assignment. For few simple multithreaded controls, we follow the Java semantics. For example, *join()* method is modeled using the so-called guard condition. Forking (start) of the new thread is modeled by a message send-receive pair. For other, more difficult, thread related Java constructs, we follow a distributed trace approach [9] which assumes that only events of the same thread (process) or transition of information from one thread to another are ordered. Unlike the classical distributed trace analysis, Java threads communicate not by messages, but by means of shared data, object locking, and several designated methods, such as *join()*, *wait()*, *notify()*. JMPaX tool is based on an interpretation of shared variable communication and object locks in terms of message passing [25]. While our approach is close to the one implemented in JMPaX, we disregard interthread communications by shared variables, which are not supposed to be heavily used in the control flow of execution. Instead, various synchronization constructs are typically used. However, as we described above, we model several Java controls, such as *join()*, *wait()*, *notify()*, which absent in JMPaX. Currently, data values and communications via shared variables are not modeled since we believe that they are not needed for antipattern detection. Note that in Java, data based communications are guaranteed to occur only if appropriate synchronization constructs are used, otherwise a change of a variable value by one thread may never become visible to other

threads [27]. The translation procedure for main constituents of the model is as follows.

**Variable/DataType Declaration:** Event attributes, such as Reference to Object Identifier (*ObjectIdRef*), Reference to Class Identifier (*ClassIdRef*), and Reference to Method Identifier (*MethodIdRef*), are declared as integer data type in PROMELA.

**Process Declaration:** Each thread in the trace translates to an active process in PROMELA. For example, a trace that includes three threads, namely, *thread2*, *thread5*, *thread6*, translates to a model with three active processes, namely, *process2*, *process5*, and *process6*, respectively in PROMELA.

**Relevant events:** Currently *start()*, *join()*, *wait()*, *notify()*, *notifyall()*, *lockacquire()*, *lockrelease()* method entry and exit, data access events are kept in the PROMELA model. Other events are not needed for verification purposes and abstracted away, though they may be helpful to locate the problems in the original application once detected.

**Event Body Translation:** Each relevant event in the XML trace translates into a *d\_step* construct in PROMELA and each event attribute translates into a variable assignment statement inside the *d\_step* construct. *d\_step* ensures that a set of assignments that represents a trace event is atomic and instant. A fragment of a trace and its PROMELA presentation are shown below.

```
<methodEntry
    threadIdRef    = "6"
    time          = "1091230513.794350600"
    methodIdRef   = "113"
    objIdRef      = "8606"
    classIdRef    = "116"
    ticket        = "1074"
    stackDepth    = "4" />

d_step{
    name          = wait_methodentry;
    threadIdRef  = 6;
    methodIdRef  = 113;
    objIdRef     = 8606;
    classIdRef   = 116; }
```

**Event Synchronization:** While each thread executes independently, a certain coordination is always required between threads to ensure data exchange and prevent stalls, data races, and other negative effects. Java offers several constructs that enforce synchronization. Among the most commonly used are the following:

- *start()* and *join()* methods allow one to fork a thread and suspend a thread execution until the termination of another thread, respectively;
- *synchronized* keyword prevents simultaneous executions of methods or blocks of code that synchronize over the same locked object (aka monitor or mutex);
- *wait()* and *notify()/notifyAll()* methods are used inside of synchronized blocks or methods to temporarily halt or resume thread execution, giving other threads a chance to execute.

While certain constructs, like *join()*, could be used with a timeout, we have not elaborated modeling of real-time aspects in our model checking approach yet. This is partially justified by the difficulty of model checking of real-time systems.

Consequently, we implemented three main types of multithreaded synchronization in the PROMELA model of the trace, by enforcing the following order:

1. *start()* method entry precedes *run()* method entry when both methods belongs to the same object (but are called from different threads).
2. Thread termination (*run()* method exit) precedes the corresponding *join()* exit.
3. The events of the same thread are modeled by totally ordered events of a PROMELA process. If the immediately preceding lock related events happens on the same object, but in different threads, the order is enforced by message exchange.

To implement order/synchronization in a PROMELA model, we use two approaches, namely, “variable/flag” based and “message passing” based approaches.

The variable/flag based approach is used to model the behavior of the *join()* method. The global Boolean variable *ActiveThread<sub>i</sub>*, where *i* is a thread identifier, is initially declared false. When the thread is started (i.e., *run()* entry event) this variable is set to true (“*ActiveThread<sub>i</sub> = true*”). Similarly, when the thread terminates (i.e., *run()* exit event), this variable is set to false (*ActiveThread<sub>i</sub> = false*). To enforce the correct event ordering, *join()* exit event is guarded with the condition (*ActiveThread<sub>i</sub> == false*). In the case when *join()* exit happens due to a timeout, our model may be inadequate.

The message passing based approach is used to enforce the order for *wait()*, *notify()*, *notifyAll()*, *lockacquire-lockrelease* and *start()* events. Message based synchronization is used in our early research prototypes, since it is easy to visualize message exchange with message sequence diagrams in Spin. However, we find that this reduces the scalability of our approach, due to Spin limitations on the number of messages and queues. Since messages could be modeled with shared variables, we also developed a more scalable version of the program, where a bit array is used to order events. The variable based approach has certain disadvantages over the message passing based approach, for example, we cannot use Spin to generate MSC (Message Sequence Chart) in the former case. Thus, in the future, we could compensate the lack of Spin visualization with a designated problem traceability/visualization module that completely hides the model checking machinery from the user.

## 7. EXPERIMENTAL EVALUATION

The custom detectors are able to analyze large programs, by using fine-tuned filters to reduce the trace size. Without filters, traces grow up to hundreds of megabytes, causing memory overflow. To counter such a problem, probably more scalable XML tools could be used for XML parsing.

We performed experiments on three applications and two antipatterns using both custom and model checking approaches.

The first application is a fragment of Java multithreaded platform Guest [17]. The second is a toy demo program (borrowed from JProbe), both with injected faults and third one is a so-called vending machine server. The experiments are performed on AMD Athlon 900 MHz system with 500M of RAM and Windows 2000 operating system. Table 1 presents some of the results obtained. Our experiments show that custom detectors are faster. However, in the case of model checking, most of the time is consumed by auxiliary steps (model checking itself takes less than a second), such as building a PROMELA model, compiling it into executable, etc.

**Table 1. Experimental Results**

Application	Anti pattern	Trace Size	PROMELA Model Size	Model Building and Verification	Custom analysis
1	Double start	64 KB	3.2 KB	13 s	4 s
2	Premature join	29 KB	2.3 KB	10 s	4 s
3	Double start	667 KB	22.0 KB	20 s	6 s

## 8. CONCLUSION AND FUTURE WORK

We proposed a method to increase the number of observable events in a method level based tracer. The method is implemented in a tool that makes Hyades adequate for detection of deficiencies in a multithreaded application. We presented our versions of custom and formal analysis of Java executions adjusted to antipattern detection. We implemented custom antipattern detectors, a prototype Spin based antipattern detection tool, and performed experimental comparison with custom antipattern detectors. While more extensive experiments, with numerous applications, antipatterns, and detection tools, should be performed to confirm such conclusions, our preliminary results imply the following. Spin model checking itself is faster than custom analyzers. However, when considered with auxiliary operations, such as model generation and compilations, Spin model checking approach appears to be somewhat slower. However, the model checking approach is indispensable for detection of many antipatterns, which are sensitive to the event order. A model checker is capable of generating event sequences that correspond to possible program executions, thus revealing not only apparent, but also possible problems. Model checking accommodates for certain distortion of the event order during tracing. On the other hand, for the detection of certain simple antipatterns, such as double *start()*, model checking does not provide any significant advantages.

Our future work includes implementing additional antipattern detectors, a more scalable back-end technique, e.g., an event-based callback XML API instead of the tree-based JAXB, and means to better visualize analysis results.

## 9. REFERENCES

- [1] Allen, E. Bug Patterns in Java. Springer, 2002.
- [2] Andrews, J.H. Deriving State-Based Test Oracles for Conformance Testing. In Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004), (Edinburgh, Scotland, May 2004), 9-16.
- [3] Bacon, D., Block, J., Bogda, J., Click, C., Haahr, P., Lea, D., May, T., Maessen, J.-W., Mitchell, J., Nilsen, K., Pugh, B., and Siroer, E.G. The “Double-Checked Locking is Broken” Declaration. <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>.
- [4] Clarke, E.M., Grumberg, O., and Long, D. Model Checking. MIT Press, 2000.
- [5] Cohen, S. JTKek.
- [6] Eclipse Platform. <http://www.eclipse.org/>.
- [7] Engler, D., and Musuvathi, M. Static Analysis versus Software Model Checking for Bug Finding. In Proceeding of the Verification, Model Checking and Abstract Interpretation (VMCAI'04), (Venice, Italy, January 2004), 191-210.
- [8] Hallal, H., Alikacem, E., Tunney, P., Boroday, S., and Petrenko, A. Antipattern-based Detection of Deficiencies in Java Multithreaded Software. In Proceedings of the Fourth International Conference on Quality Software (QSIC2004), (Braunschweig, Germany, September 8-10, 2004), 258-267.
- [9] Hallal, H., Boroday, S., Ulrich, A., and Petrenko, A. An Automata-based Approach to Property Testing in Event Traces. In Proceedings of the International Conference on Testing of Communicating Systems (TestCom 2003), (Sophia Antipolis, France, May 26-29, 2003), 180-196.
- [10] Havelund, K., and Rosu, G. Monitoring Java Programs with Java PathExplorer. In Proceeding of First Workshop on Runtime Verification (RV'01), (Paris, France, July 23, 2001).
- [11] Havelund, K., Johnson, S., and Rosu, G. Specification and Error Pattern Based Program Monitoring. In Proceeding of European Space Agency Workshop on On-Board Autonomy, (Noordwijk, Holland, October 2001).
- [12] Havelund, K., Stoller, S., and Ur, S. Benchmark and Framework for Encouraging Research on Multi-Threaded Testing Tools. In Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD), (Nice, France, April 22-26, 2003).
- [13] Hewlett-Packard Company. Visual Threads Home. [http://h21007.www2.hp.com/dspp/tech/tech\\_TechSoftwareDetailPage\\_IDX/1,1703,5062,00.html](http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,5062,00.html).
- [14] Holzmann, G. J. The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, 2003.
- [15] Hyades Project. <http://www.eclipse.org/hyades/>.
- [16] Java MultiPathExplorer. JMPaX 2.0. <http://yangtze.cs.uiuc.edu/~ksen/jmpax/web>, 2005.
- [17] Magnin, L., Pham, V.T., Dury, A., Besson, N., and Thieffaine, A. Our guest agents are welcome to your agent platforms. In Proceedings of the ACM Symposium on Applied Computing (SAC 2002), (Madrid, Spain, March 10-14, 2002), 107-114.
- [18] Naumovich, G., Avrunin, G.S., and Clarke, L. A. Data Flow Analysis for Checking Properties of Concurrent Java Programs. In Proceedings of the 21st International Conference on Software Engineering, (May 1999), 399-410.
- [19] Petrenko, A., Boroday, S., and Groz, R. Confirming Configurations in EFSM Testing. IEEE Transactions on Software Engineering, 30, 1, (January 2004), 29-42.
- [20] Quest Software. JProbe. <http://www.quest.com/jprobe>.
- [21] Runtime Monitoring and Checking. <http://www.cis.upenn.edu/~rtg/mac/>, 2005.
- [22] Rutar, N., Almazan, C.B., and Foster, J. S. A Comparison of Bug Finding Tools for Java. In Proceeding of 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04), (Saint-Malo, France, November 2004), 245-256.
- [23] Sammapun, S., Sharykin R., DeLap, M., Kim, M., and Zdancewic, S. Formalizing Java-MaC. Electr. Notes Theor. Comput. Sci. 89, 2, (2003).
- [24] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Trans. on Comp. Syst., 15, 4, (November 1997), 391-411.
- [25] Sen, K., Rosu, G., and Agha, G. Online Efficient Predictive Safety Analysis of Multithreaded Programs. In Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), (Barcelona, Spain, March 2004), 123-138.
- [26] Spec Patterns. <http://patterns.projects.cis.ksu.edu/>, 2005.
- [27] Sun Microsystems. Java. <http://sun.java.com>, 2005.