Stateful Breakpoints: A Practical Approach to Defining Parameterized Runtime Monitors

Eric Bodden
Software Technology Group, Technische Universität Darmstadt
Center for Advanced Security Research Darmstadt (CASED)
bodden@acm.org

ABSTRACT

A runtime monitor checks a safety property during a program's execution. A parameterized runtime monitor can monitor properties containing free variables, or parameters. For instance, a monitor for the regular expression "`close(s)+ read(s)`" will warn the user when reading from a stream `s` that has previously been closed. Parameterized runtime monitors are very expressive, and research on this topic has lately gained much traction in the Runtime Verification community. Existing monitoring algorithms are very efficient. Nevertheless, existing tools provide little support for actually defining runtime monitors, probably one reason for why few practitioners are using runtime monitoring so far.

In this work we propose the idea of allowing programmers to express parameterized runtime monitors through stateful breakpoints, temporal combinations of normal breakpoints, a concept well known to programmers. We show how we envision programmers to define runtime monitors through stateful breakpoints and parameter bindings through breakpoint expressions. Further, we explain how stateful breakpoints improve the debugging experience: they are more expressive than normal breakpoints, nevertheless can be evaluated more efficiently. Stateful breakpoints can be attached to bug reports for easy reproducibility: they often allow developers to run directly to the bug in one single step. Further, stateful breakpoints can potentially be inferred from a running debugging session or using property inference and fault localization tools.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation; D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Human Factors, Verification

Keywords

Parameterized runtime monitors, Stateful breakpoints

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

1. NEW IDEA: DEFINING RUNTIME MONITORS WITH BREAKPOINTS

A runtime monitor checks safety properties of some program under test during that program's execution. In runtime monitoring, the program under test is instrumented to notify the monitor about certain events of interest during the program's own execution. The monitor then reacts to those events, typically by changing some monitor-internal state. When runtime monitors are used for runtime verification, erroneous sequences of events will typically drive the runtime monitor into an error state, causing the monitor to notify the program of a property violation.

Runtime monitors that react to a single, flat stream of events are, however, not very expressive. For instance, consider a monitor that is supposed to notify users when reading from a stream that has previously been closed. Such a monitor would typically be denoted by a regular expression "`close+ read`". Now assume a program execution in which the program opens streams s_1 and s_2, then closes s_1 but reads from s_2. A naïve monitor would signal an error even in this case, although the close and read operations involved different objects. Researchers have therefore introduced the concept of parameterized runtime monitors. In the parameterized approach, the regular expression is extended to "`close(s)+ read(s)`": a violation only occurs when both events involve the same stream object `s`.

Runtime verification, and in particular runtime verification with parameterized runtime monitors is perceived as a powerful concept within the Runtime Verification community. Opposed to many static program analyses, runtime monitors usually have no false positives, i.e., they notify the user only of actual error as they occur on a particular program run, and integrate well with testing. Efficiency used to be problematic: if implemented naïvely, parameterized runtime monitors can slow down the program under test by several orders of magnitude because at every event of interest the program needs to wait for the monitor to update its internal state. In recent years, however, researchers have shown that even parameterized runtime monitors can be implemented with little to no runtime overhead in the vast majority of cases [3–5]. Hence, runtime monitoring is finally arrived at a point where one would assume the technology to become mainstream. Nevertheless, two key problems still hinder adoption by main-stream programmers: (1) How to define runtime monitors? (2) How to debug the program once an error has been found? In this paper, we propose to address both problems at once by defining monitors through the well-known concept of breakpoints.

1.1 State of the art: textual monitor definition paired with code generation

Many programmers nowadays use Integrated Development Environments (IDE) that support rapid application development through features such as instant feedback, instant compilation, code completion, and so on. Traditional runtime verification tools, on the other hand, require a complex, multi-step development process. Programmers typically need to define the monitor through some textual declarative monitor specification, using a monitoring logic such as Linear Temporal Logic [9], Context-free Grammars, or Regular Expressions [1, 4]. The runtime verification tool then reads this textual specification as an input, along with the code of the program under test, emitting a version of that program instrumented with the runtime monitor that the textual specification defines. The programmer then needs to test-run this version of the program to search for potential property violations. This multi-step process is problematic, as it does not align well with the way in which many of today's programmers write and maintain their program code in modern IDEs.

But let's assume that the programmer does test-run the instrumented program, and the runtime monitor indeed notifies the user of a property violation. This is great, we have just found a programming error! But then what? Current runtime verification tools typically issue an error message on the command line, potentially along with a stack trace at the time at which the violation was detected. While this is useful, the user often still needs to establish a debug session to actually find out the root cause of the bug.

1.2 Proposed solution: Runtime monitors as stateful breakpoints

In this work we propose to integrate runtime monitoring tightly with the debugging support of a modern IDE. The idea is to provide developers with a means to define parameterized runtime monitors using a concept typically well-known to them: breakpoints. In our approach, a runtime monitor resembles a "stateful breakpoint". Such a breakpoint comprises three important elements:

1. a set of primitive (ordinary) breakpoints, along with textual labels for these breakpoints,
2. a pattern defining a temporal ordering over these labeled breakpoints,[1] and
3. a set of variables (parameters). Every primitive breakpoint can define a program expression for each variable. The expressions will be evaluated when the breakpoint "hits", binding the expression's return value to the variable.

We are currently integrating support for such stateful breakpoints into the Eclipse IDE [7]. Figure 1a shows how we envision programmers to define runtime monitors with our Eclipse plugin. Figure 1b shows an example program which the programmer may want to debug using the stateful breakpoint from Figure 1a. The intention of the stateful breakpoint is to halt the program during debugging when reading from a closed stream. In the example program, this is the case when line 19 executes in the context of the method call in line 10.

[1]We propose to use a regular expression.

Name	No read after close	
Breakpoint	Label	Variable 1
• Example - line 14	closeStream	`stream`
• Example - line 19	readStream	`is`
Pattern:	closeStream+ readStream	

(a) Schematic view of monitor definition

```
1  class Example {
2
3  public static void main(String[] args)  {
4     InputStream s = new FileInputStream(..);
5     InputStream s2 = new FileInputStream(..);
6
7     closeit(s2);
8     readit(s);
9     closeit(s);
10    readit(s);
11 }
12
13 static void closeit(InputStream stream) {
14 •   stream.close(); //<-- prim. breakp. "close"
15 }
16
17 static void readit(InputStream is) {
18    int i;
19 •  while((i=is.read())>-1) { //<-- prim. breakp. "read"
20        System.out.println(i);
21    }
22 }
23
24 }
```

(b) Example program using streams

Figure 1: Example monitor and program

As Figure 1a shows, our envisioned user-interface lets users define monitors through a table-like view. On the left, in the column "Breakpoint", the user can select primitive breakpoints from a list of breakpoints defined through Eclipse's usual Breakpoints View. In the column "Label", the user can assign every breakpoint a name. Our plugin will assign a sensible default to the label but in principle the user can choose any label she wishes. The "Pattern" at the bottom defines a temporal ordering over these labeled breakpoints. In the example, the monitor detects a property violation, and halts the program during debugging, when the breakpoint "read" at line 19 is visited after the program's execution has passed through the breakpoint "close" at line 14 at least once. The column "Variable 1" allows programmers to restrict the matching of the regular expression to such fragments of the execution trace where the expressions defined in this column all evaluate to the same value/object. In the example, the programmer stated that the program should halt only if the expression `stream` at line 14 returns the same object as the expression `is` at line 19. Users can add further Variable columns as needed.

We wish to note that primitive breakpoints cannot only comprise line-based breakpoints as shown in Figure 1b. Integrated Development Environments like Eclipse further offer the opportunity to define primitive breakpoints that trigger on any invocation of a specific method no matter where the

invocation is taking place, on read or write accesses to a specific field, and on throwing exceptions of a certain type. A programmer can use all these kinds of breakpoints as primitive breakpoints within a stateful breakpoint.

The "Pattern" to be entered is in our current design a regular expression over an alphabet Σ of literals defined by the breakpoints' labels. In our example, for instance, we have $\Sigma = \{\texttt{closeStream}, \texttt{readStream}\}$. In our current prototype, we plan to allow for the operators +, *, ?, $\cdot$ and $(\cdot)$ respectively, with their usual semantics. In the expression, literals end at operators or at white space. Hence, "`closeStream readStream`" is a valid expression while the expression "`closeStreamreadStream`" is not: the latter expression refers to a single literal `closeStreamreadStream`, which is not defined as a label. Our current prototype eases the writing of regular expressions with using a code-completion approach that suggests declared labels while typing.

1.3 Benefits over traditional breakpoints

Stateful breakpoints are much more expressive than primitive breakpoints. The stateful breakpoint in the example allows programmers to halt at line 19 only if the "read" was preceded by a "close". This avoids unnecessary stepping through the code. The support for parameters such as "Variable 1" allows for further filtering and ease of debugging: without parameter support, the debugger would need to halt the program also when the "read" at line 19 executes in the context of the method call in line 8, although at this point the program has closed only stream `s2` and not `s`.

Stateful breakpoints provide several interesting software engineering benefits.

- They extend the well-known concept of breakpoints in a natural way.
- If defined correctly, they allow the debugger to execute the program directly until the point of error in one single step.
- We plan to allow users to persist stateful breakpoints, and to attach them to bug reports. This allows users to easily communicate bugs to developers.
- Conversely, API developers can ship their libraries with stateful breakpoints that get loaded into the IDE automatically when the library is used. Should the developer accidentally misuse the library, e.g. by calling methods in the wrong order, the program will halt automatically, providing the programmer full context for debugging.

While we believe that stateful breakpoints are reasonably easy to define, in future work we will consider inferring stateful breakpoints automatically from a debugging session that uses primitive breakpoints only. By observing the order in which the programmer sets, disables and re-enables primitive breakpoints we hope to be able to infer higher-level stateful breakpoints automatically. Fault localization tools [11] and approaches to API property mining could be another source of useful stateful breakpoints.

Another interesting piece of future work would involve the evaluation of stateful breakpoints not within the virtual machine that hosts the debugger (the Debugger-VM) but rather directly within the virtual machine that hosts the program under test. This way of breakpoint evaluation has the potential to reduce the communication between both virtual machines by orders of magnitudes, hence speeding up the debugging process significantly: instead of notifying the Debugger-VM of every single event of interest, the Debugger-VM would only be notified once the monitor has actually reached an error state.

1.4 Benefits over traditional runtime verification tools

Stateful breakpoints support properties just as expressive as the ones supported by traditional runtime verification tools. Nevertheless, stateful breakpoints provide several software engineering benefits over those tools.

- Stateful breakpoints can be defined directly in the IDE.
- Stateful breakpoints are defined using the concept of breakpoints, which should be familiar to most programmers.
- Stateful breakpoints allow programmers to immediately halt the program when a property violation is detected. At this point, the programmer can use all tooling provided by the debugger to identify the cause of this violation.
- Stateful breakpoints are a "push-button technology". There is no need for separate monitor definition, code generation and execution of the instrumented program. Instead, the debugger simply executes the original, uninstrumented program, but is notified of the necessary events through the debugging protocol.

2. NOVELTY OF THE IDEA

The idea is new because it is the first attempt to bring declarative monitor specifications into an IDE. Further, it is the first attempt to express runtime monitors with the well-known concepts of breakpoints. In traditional approaches, monitors are instead defined through some form of declarative event patterns, typically AspectJ [2] pointcuts.

While there has been one approach [6] (see below) to adding the notion of control flow to breakpoints (see Section 3), our approach goes much beyond the simple notion of control flow: we propose to define fully-fledged runtime monitors, parameterized through variable bindings that bind free variables to the return values of program expressions. Parameters are essential for obtaining breakpoints that are expressive enough so that they can allow programmers to run "to the bug" in one single step.

3. RELATED WORK

The most related work is the work by Chern and De Volder on Control-Flow Breakpoints [6]. The authors observed that traditional breakpoints can be made much more useful if programmers can constrain them to halt the program only after some other breakpoints were visited already (or not visited yet). Using an empirical evaluation, the authors show that the ability to restrict breakpoints using such control-flow restrictions can indeed ease the reproduction of a significant portion of known bugs.

Our idea of using stateful breakpoints to define runtime monitors goes, however, much beyond the idea of simple control-flow restrictions. In particular, we propose to enable programmers to define stateful breakpoints using declarative

patterns such as regular expressions. Further, we argue that stateful breakpoints must support parametric runtime monitors. Otherwise, the monitor will yield too many false positives, halting the program although no property violation has been observed.

The work of Chern and De Volder is also different in the sense that it fulfils a different goal. Chern's and De Volder's goal was to ease debugging. Our primary goal is rather to make runtime verification a mainstream technology. Breakpoints are merely our means to achieve this goal and the improved ease of debugging is a welcome side effect.

4. EXPECTED FEEDBACK

We are currently finalizing the implementation of our research prototype. The author's research group has a good understanding of the techniques involved to make the technology work. To be able to evaluate parameterized runtime monitoring without having to resort to code generation (as in traditional runtime verification tools), we implemented a novel runtime verification library called MOPBox [8]. This library encapsulates the entire matching logic required to determine whether or not to hold at a particular breakpoint.

At the conference, we hence expect feedback mainly with respect the two following important topics:

- Does the user interface appear intuitive to other developers? Can it be improved, and if so, how?
- How can we effectively evaluate the usefulness of our approach? In the runtime verification community [10], runtime monitoring approaches are typically evaluated with respect to their expressiveness or with respect to the ability to implement efficient monitors for the approach. Usability evaluations are rarely performed.

5. CONCLUSION

We have presented the idea of defining runtime monitors through a concept well known to programmers: breakpoints. A runtime monitor is defined as a "stateful breakpoint" that halts the program only after its primitive breakpoints have been visited in a given order, and only if the context at the times during which these breakpoints were visited fulfils certain constraints. Stateful breakpoints are very expressive and often allow developers to directly run to a bug in one single step. This makes it useful to persist stateful breakpoints, and to attach them to bug reports as a means to communicate bugs to developers. Conversely, developers can ship libraries along with stateful breakpoints which will halt the client's program when misusing the library.

We are currently implementing the proposed approach as an extension to the popular Eclipse IDE [7]. To be able to evaluate parameterized runtime monitoring without having to resort to code generation (as in traditional runtime verification tools), we implemented a novel runtime verification library called MOPBox [8].

Acknowledgements.

We wish to thank the anonymous reviewers for their helpful and thorough feedback. This work was supported by the Center for Advanced Security Research Darmstadt, CASED (www.cased.de).

6. REFERENCES

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364. ACM Press, October 2005.

[2] The AspectJ home page, 2003.

[3] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 589–608. ACM Press, October 2007.

[4] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588. ACM Press, October 2007.

[5] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science (LNCS)*, pages 246–261. Springer, March 2009.

[6] Rick Chern and Kris De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 96–106, New York, NY, USA, 2007. ACM.

[7] Eclipse IDE. `http://www.eclipse.org/`.

[8] MOPBox, runtime monitoring in a box. `http://mopbox.googlecode.com/`.

[9] Amir Pnueli. The temporal logic of programs. In *IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society, October 1977.

[10] Workshop and conference series on Runtime Verification. `http://www.runtime-verification.org/`.

[11] Cheng Zhang, Dacong Yan, Jianjun Zhao, Yuting Chen, and Shengqian Yang. Bpgen: an automated breakpoint generator for debugging. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 271–274, New York, NY, USA, 2010. ACM.