MOPBox: A Library Approach to Runtime Verification
(Tool Demonstration)

Eric Bodden

eric.bodden@cased.de
Center for Advanced Security Research Darmstadt
Software Technology Group
Technische Universität Darmstadt, Germany

Abstract. In this work we propose *MOPBox*, a library-based approach to runtime verification. *MOPBox* is a Java library for defining and evaluating parametric runtime monitors. A user can define monitors through a simple set of API calls. Once a monitor is defined, it is ready to accept events. Events can originate from AspectJ aspects or from other sources, and they can be parametric, i.e., can contain variable bindings that bind abstract specification variables to concrete program values. When a monitor reaches an error state for a binding $\vec{v} = \vec{o}$, *MOPBox* notifies clients of a match for $\vec{v} = \vec{o}$ through a call-back interface. To map variable bindings to monitors, *MOPBox* uses re-implementations of efficient indexing algorithms that Chen et al. developed for JavaMOP.
We took care to keep *MOPBox* as generic as possible. States, transitions and variable bindings can be labeled not just with strings but with general Java Objects whose types are checked through Java Generics. This allows for simple integration into existing tools. For instance, we present ongoing work on integrating *MOPBox* with a Java debugger. In this work, transitions are labeled with breakpoints.
MOPBox is also a great tool for teaching: its implementations of monitor indexing algorithms are much easier to understand than the code generated by tools such as JavaMOP. Indexing algorithms use the Strategy Design Pattern, which makes them easily exchangeable. Hence, *MOPBox* is also the perfect tool to explore and test new algorithms for monitor indexing without bothering about the complex intricacies of code generation. In the future, we further plan to integrate *MOPBox* with the Clara framework for statically evaluating runtime monitors ahead of time.

1 Motivation and Description of *MOPBox*

In the past decade, researchers in Runtime Verification have developed a range of specialized tools for generating runtime monitors from formal specifications [1–5]. Typically, those tools support parametric monitor specifications, i.e, specifications that allow the monitoring of individual objects or even combinations of objects. Figure 1, for example shows a finite-state machine representing the

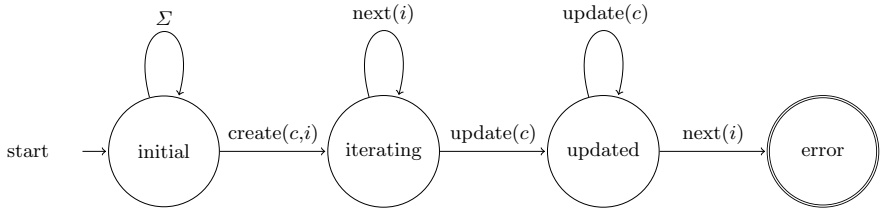

Fig. 1. Runtime monitor for FailSafeIter property [1]: Do not modify a collection while iterating over it.

"FailSafeIter" property [1]: one should not use an iterator i for a collection any longer if c was updated after i had been created. In this case, there exists a single monitor instance (holding the state machine's internal state) for any combination of c and i occurring on the monitored program execution.

Research in Runtime Verification has made big leaps to making runtime monitoring of such parameterized properties efficient [3, 6–8] through the generation of property-specific monitoring code. However, efficiency should not be the only goal to pursue in runtime monitoring. While auto-generated monitoring code may be maximally efficient, it is generally hard to understand and debug. In addition, approaches based on code-generation often involve multiple, loosely integrated tools, hindering integration of those tools into other applications.

Another problem with those loosely integrated tool chains is that they hinder comparison of montoring approaches. In recent work, Purandare et al. perform an in-depth comparison with respect to the relative performance of several monitoring algorithms [9]. As the authors show, this performance can depend on the property to be monitored: different algorithms are ideal for different properties. Current tool chains cannot easily support multiple algorithms as they are not integrated.

2 Defining monitor templates

In this work we hence propose *MOPBox*, a library-based approach to runtime verification. *MOPBox* is a Java library for defining and evaluating parametric runtime monitors such as the one shown in Figure 1. With *MOPBox*, a user can define templates for runtime monitors through a simple set of API calls.[1] Figure 2 shows how a user would define a monitor template for the FailSafeIter property mentioned earlier.

First, in line 2, the user defines that she wishes to implement a template based on finite-state machines, with String labels on transitions and variable bindings that map from `Var` instances to any kinds of Objects. The range of template variables `Var` is defined as an enum in line 4.

[1] We use the phrase "monitor template" to denote a property that *MOPBox* should monitor. During the execution of the program under test, each template will generate a set of monitors, one monitor for each variable binding.

```
1  public class FailSafeIterMonitorTemplate
2    extends AbstractFSMMonitorTemplate<String,Var,Object> {
3
4  public enum Var{ C, I }
5
6  protected void fillAlphabet(IAlphabet<String,Var> a) {
7      a.makeNewSymbol("create", C, I);
8      a.makeNewSymbol("update", C);
9      a.makeNewSymbol("iter", I);
10 }
11
12 protected State<String> setupStatesAndTransitions() {
13     State<String> initial = makeState(false);
14     State<String> iterating = makeState(false);
15     State<String> updated = makeState(false);
16     State<String> error = makeState(true);
17
18     initial.addTransition(getSymbolByLabel("create"), iterating);
19     initial.addTransition(getSymbolByLabel("update"), initial);
20     initial.addTransition(getSymbolByLabel("iter"), initial);
21     iterating.addTransition(getSymbolByLabel("iter"), iterating);
22     iterating.addTransition(getSymbolByLabel("update"), updated);
23     updated.addTransition(getSymbolByLabel("update"), updated);
24     updated.addTransition(getSymbolByLabel("iter"), error);
25     return initial;
26 }
27
28 protected IIndexingStrategy<String,Var,Object> createIndexingStrategy() {
29     return new StrategyB();
30 }
31
32
33 protected void matchCompleted(IVariableBinding<Var, Object> binding) {
34     System.err.println("MATCH for binding: "+binding);
35 }
36
37 }
```

Fig. 2. Monitor template for FailSafeIter property in *MOPBox*

In lines 6–9, the user then lists the alphabet to be used, i.e., the different kinds of events that the monitors of this template should prepare to process. At this point the user also binds event names such as `create` to template variables `C` and `I`. In the example, event labels are Strings because the user chose type String as type parameter in line 2. One could have chosen other types of labels. In a current piece of work we are integrating *MOPBox* with the Java debugger of the Eclipse IDE [10]. In this setting, events are labeled with breakpoints that are triggered at debug time [11].

In lines 13–16, the user calls the factory method `makeState` to create the states that will make up the monitor template's state machine. An error state is created using `makeState(true)`. In lines 18–24, finally, the user defines the state machine's transition relation. At the end of the method, by convention, the user returns the machine's initial state.

```
1  after(Collection c) returning(Iterator i):
2      call(* Iterable+.iterator()) && target(c) {
3      IVariableBinding<Var, Object> binding
4        = new VariableBinding<Var, Object>();
5      binding.put(Var.C,c);
6      binding.put(Var.I,i);
7      template.processEvent(
8          "create",
9          binding
10     );
11 }
```

Fig. 3. AspectJ advice dispatching "create" events to the monitor template

In lines 28–30, the user selects an indexing strategy. An indexing strategy implements an indexing algorithm that dispatches parameterized events to monitors for the appropriate parameter instances. In this example, the user opted for our implementation of Chen et al.'s *Algorithm B* [7]. *MOPBox* uses the Strategy Pattern [12] to make indexing strategies easily exchangeable. This also facilitates rapid prototyping and testing of new indexing algorithms. For instance, users can instantiate multiple monitor templates that use different indexing strategies but are otherwise identical. If for the same events one template finds a match and the other one does not, this indicates a bug in one of the indexing strategies. *MOPBox* holds no static state. To reset a monitor, one hence simply needs to re-instantiate a template's indexing strategy.

Last but not least, in lines 33–35, the user defines the call-back method `matchCompleted`. *MOPBox* will call this method automatically whenever one of the monitors of this template completes a match. *MOPBox* passes the matching variable binding into the method as a parameter.

3 Sending events to monitor templates

In Figure 3 we show how users can use AspectJ [13] to send concrete program events to a monitor template. The AspectJ advice intercepts calls to `Collection.iterator()` and notifies the monitor template, passing in a variable binding mapping the template variables `C` and `I` to concrete program values. Users do not necessarily have to use aspects to generate events. In a current piece of work we are integrating *MOPBox* with the Java debugger of the Eclipse IDE [10]. In that case, events are triggered directly by the debugger's application interface.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with

Free Variables to AspectJ. In: OOPSLA. (October 2005) 345–364
2. Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (November 2005)
3. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA. (October 2007) 569–588
4. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: Symposium on the Foundations of Software Engineering (FSE). (November 2006) 219–230
5. Krüger, I.H., Lee, G., Meisinger, M.: Automating software architecture exploration with M2Aspects. In: Workshop on Scenarios and state machines: models, algorithms, and tools (SCESM). (May 2006) 51–58
6. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. In: OOPSLA. (October 2007) 589–608
7. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 5505 of LNCS., Springer (March 2009) 246–261
8. Chen, F., Meredith, P., Jin, D., Roşu, G.: Efficient formalism-independent monitoring of parametric properties. In: ASE. (2009) 383–394
9. Purandare, R., Dwyer, M., Elbaum, S.: Monitoring finite state properties: Algorithmic approaches and their relative strengths. In: RV '11: International Conference on Runtime Verification. (September 2011) To appear.
10. Eclipse IDE: `http://eclipse.org/`.
11. Bodden, E.: Stateful breakpoints: A practical approach to defining parameterized runtime monitors. In: ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. (September 2011) New Ideas Track. To appear.
12. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design patterns: Abstraction and reuse of object-oriented design. In: Proceedings of the 7th European Conference on Object-Oriented Programming. ECOOP '93, London, UK, UK, Springer-Verlag (1993) 406–431
13. AspectJ team: The AspectJ home page, `http://eclipse.org/aspectj/` (2003)