Project Paper: Java Safety Analysis Tool

Team: THEORACTICE
Member: Sangjin Han
Kangwoon Hong
Hyungchoul Kim
Andrew O. Mellinger

1. Introduction

1.1 Motivation

Our team's studio project is based on Java programming language. In addition, we want to check implementation even if the implementation is not completed but partially developed. In particular, we are interested in the sequence of method invocations because we consider our sequence diagrams as one of the most significant design documents.

Thus, the goal of this project was to develop a static Java source code analyzer that could easily find the behavioral inconsistency between design and implementation. This goal led to a few specific requirements:

1) Information should be derived from user specifications. Design intent is not explicit in programming languages such as Java, C, and C++. Therefore, without annotations or rules defined by programmers, an analysis tool could not analyze the consistency between design and implementation with assurance.
2) Specifications should be lightweight and easy to write. If the rules for specification are complex and burdensome to users, the users will be reluctant to use the analysis tool. At the worst case, time for specification may be much larger than time for manual code inspection.
3) Just by modifying specifications, without accessing source code, different kinds of property analyses should be available.

Table 1 shows that only ESC/JAVA and Fluid can analyze java source code statically according to specifications written by users.

Analysis Tool	Analysis Target	Static / Dynamic	Specification by User	Java ∧ Static ∧ User Spec.
ESC/JAVA	Java code	Static	Required	**True**
Fluid	Java code	Static	Required	**True**
PREfix	C/C++ code	Static	Not Required	**False**
Metal	C/C++ code	Static	Required	**False**
Fugue	C#, VB.Net, Managed C++ code	Static	Required	**False**
Daikon	C/C++ code	Dynamic	Not Required	**False**
Blast	C/C++ code	Static	Required	**False**

Table 1: Analysis Tools Comparison

Even though these two tools can satisfy the aforementioned goal, they cannot satisfy the aforementioned three requirements; in particular, requirement 3). ESC/JAVA and Fluid combined user annotation into java source code.

Our tool separated user specification from source code. In addition, for user convenience, we used the terms and concepts used in UML statechart diagram: event, guard, and action.

1.2 JSAT (Java Safety Analysis Tool)

Java Source Code. This is the target source code to be checked.
Specification. This is written by users being separated from the source code.
Syntax Analyzer. This is abstract syntax tree creator included in Crystal2 framework.
Parser. This reads user specifications, which are *.jsat files, and stores those into newly designed data structures.
Safety Checker. This analyzes the target source code to check its consistency with the specifications; this was built on Crystal2 framework and uses dataflow analysis.

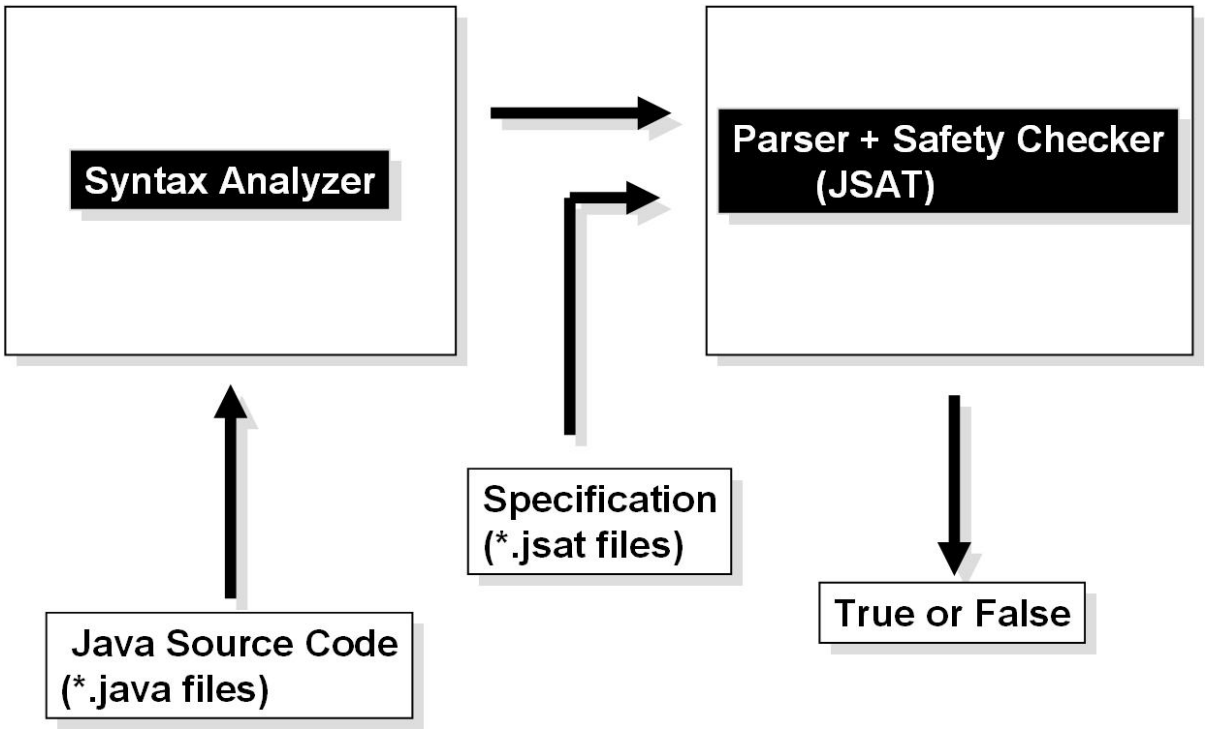

Figure 1: JSAT system architecture

1.3 Theoretical Background

JSAT performs a static and modular analysis to provide a set of analysis messages. The analysis is static because it inspects the program's source code, without any instrumentation to perform checks during execution. In addition, the analysis is modular in that, at a method call site, the analysis inspects the callee's declaration and not its body [1].

JSAT allows users to specify state machine protocols. Using a state machine protocol, the user can constrain the order in which methods can be called to the transitions of a given state machine [1].

Moreover, JSAT uses dataflow analysis implemented in Crystal2. Therefore, we defined new lattice, tuple lattice, and flow function.

2. Safety Property Specification

2.1 Core specification syntax

$$
\begin{aligned}
&\mathit{StateVariable} ::= a \\
&\qquad\qquad | \; a, \mathit{StateVariable} \\
&\\
&\mathit{Event} ::= \mathit{Pattern}, \mathit{Guard} \\
&\qquad | \; \mathit{Pattern}, \mathit{Action} \\
&\qquad | \; \mathit{Pattern}, \mathit{Guard}, \mathit{Action} \\
&\\
&\mathit{Pattern} ::= \mathit{MethodInvocation} \\
&\\
&\mathit{Guard} ::= \mathit{Predicate} \\
&\qquad | \; \mathit{Predicate} \&\& \mathit{Guard} \\
&\\
&\mathit{Predicate} ::= a == e \\
&\qquad\qquad | \; a != e \\
&\qquad\qquad | \; a < e \\
&\qquad\qquad | \; a > e \\
&\qquad\qquad | \; a <= e \\
&\qquad\qquad | \; a >= e \\
&\\
&\mathit{Action} ::= \mathit{Assignment} \\
&\qquad | \; \mathit{Assignment}, \mathit{Action} \\
&\\
&\mathit{Assignment} ::= a = e \\
&\\
&e ::= a \\
&\quad | \; c \\
&\\
&\mathit{variable} \quad a \\
&\mathit{value} \quad c
\end{aligned}
$$

Figure 2: Core specification syntax

2.2 Informal description of syntax

As we mentioned, our specification is based on the UML statechart diagram and similar to the specification of the BLAST [2].

State variable. This is definition of single variables prefaced by the keyword *global*; for example, *global int lockStatus = -1*;.

Events. These are used to change global state and verify properties based on the execution of method invocations. An event consists of the keyword *event* followed by a sequence of sub-directives within braces.

Pattern. This specifies which possible program statements, which are method invocation at this time, activate an event. A pattern consists of the keyword *pattern* followed by a method invocation statement enclosed in braces. An event will be activated for any method invocation statements that match the pattern.

Guard. This can be used as pre- condition for a method invocation. If the guard expression is true, the specified *action* code is run. If the guard is false, it means the inconsistency between specification and implementation. A guard consists of the keyword *guard* followed by predicates inside braces.

Action. This can be used as post- condition for a method invocation. An action consists of the keyword *action* followed by sequences of assignment statements inside braces.

Figure 3 shows an example of specification, which reflects full specification syntax.

```
global int lockStatus = -1;

event {
        pattern { smInit(); }
        guard { lockStatus == -1 }
        action { lockStatus = 0; }
}
```

```
event {
        pattern { smLock(); }
        guard { lockStatus == 0 }
        action { lockStatus = 1; }
}

event {
        pattern { smUnlock(); }
        guard { lockStatus == 1 }
        action { lockStatus = 0; }
}
```

Figure 3: Simple Lock & Unlock specification example

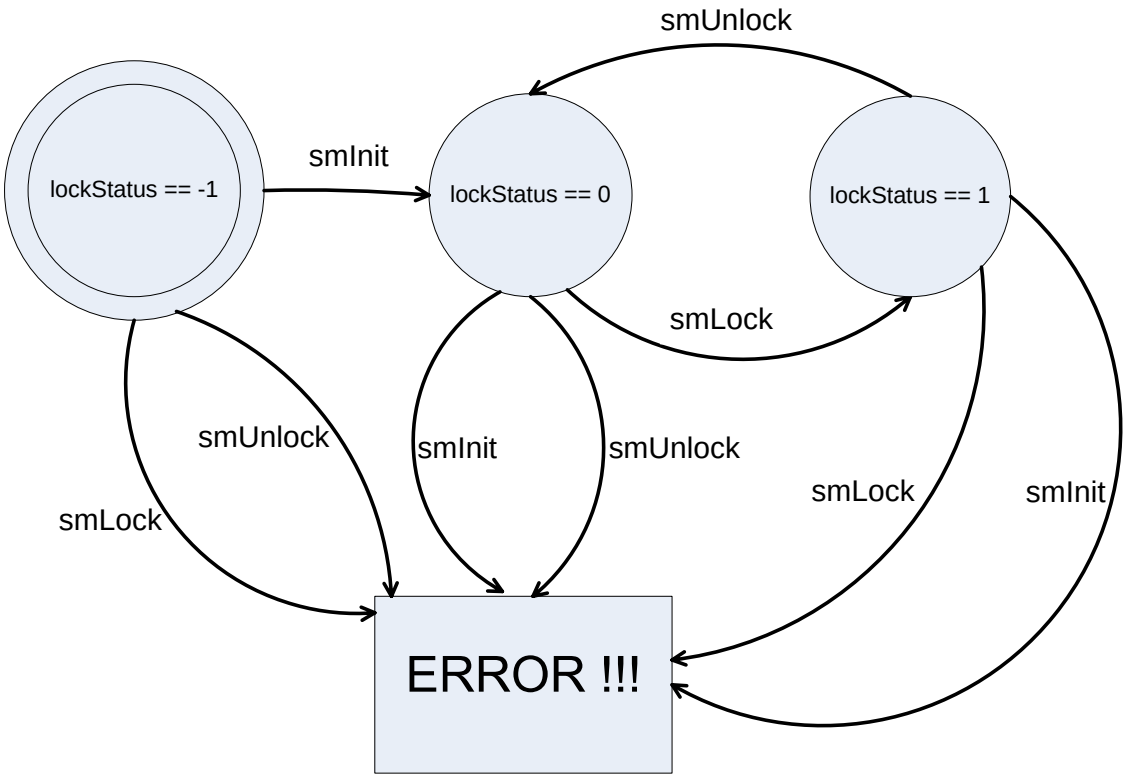

Figure 4: State machine for simple Lock & Unlock example

2.3 Implementation

We implemented a parser that can read specifications and a checker that can check the specifications for consistency with the implementation. Our implementation can handle three examples, which are JSATSample1.java, JSATSample2.java, and JSATSample3.java. We implemented new data structures to save the specifications. In addition, we implemented new lattice, tuple lattice, and flow function to use the concept of data flow analysis.

The parser consists of the following files:

Event.java
GlobalType.java
PatternType.java
GuardType.java
ActionType.java
SpecParser.java
LinkedListNode.java

The checker consists of the following files:

JSATAnalysis.java
JSATAnalysisDefinition.java
JSATDataModel.java
JSATLatticeElement.java
JSATTupleLatticeElement.java

As we mentioned, we wrote three examples:

JSATSample1.java
JSATSample2.java
JSATSample3.java

Sample1.jsat is specification for JSATSample1.java, Sample2.jsat is for JSATSample2.java, and Sample3.jsat is for JSATSample3.java.

3. Lessons Learned

The initial implementation of JSAT tried to emulate the pattern matching idea in Metal/Blast which leads to some interesting properties. Pattern matching against method names means that state is coupled to a global state not a per object basis. (Unless the pattern is design to match an exact variable name, this still has problems with aliasing of course.) While this may not seem desirable, it does hold true to the Metal paradigm. In the future, we think that the specification syntax should be extended to have support to indicate that patterns are method signatures and should be track on a per object basis. The global usage behavior can be seen clearly in the Sample3 code.

We had also intended to support regular expressions with the engine, but ran into specification syntax issues. It is nice to be able to specify a basic method pattern without having to escape parenthesis. Parenthesis show up frequently in code, but are special symbols in regular expressions.

The first implementation was intended to match against any code and not just method names. This was not implemented because of the difficulty of limited the pattern match to just the code exclusive to the node. Being a tree, the code associated with a node, may also be associated with a child node. There is no simple way to determine at what level a particular piece of source should be matched, and how to transfer values. This is not a trivial problem.

We implemented three sample cases. The first was the trivial state locker used in the presentation, the second was the test code used in the BLAST assignment (ported to Java), and the third was the SimpleProtocol test code used in the protocol assignment. While using these we found an interesting property of the test. The Crystal2 flow analysis did not provide branch-merger code as expected on the BadDriver case. In this example, two if statements exists (not an if-else) and both can be entered during execution of the method. It appears that the flow analysis did not enter both in the same code path an attempt to merge the results. Due to time constraints, an analysis of the flow analysis tool was not done, and we do not know why this test case did not behave as expected.

4. Implementation Details: Analysis Engine

The implementation has three major components: a parser, a data model, and the analysis engine. The parser and resulting data model are straightforward modularized engineering tasks. The analysis engine, which relies heavily on the Crystal2 framework, was implemented in the following ways:

JSATLatticeElement.java

This was modified to track the value of a specific global variable. Unlike previous analysis (i.e. mapping ASTNodes to particular values), Strings were mapped to integer values. As with the protocol analysis assignment, explicit values were not known at compile time, so a map was used to hold all the possible values with corresponding element objects. The object references were kept in the lattice so the join and alap (atLeastAsPrecise) logic, which are references based, would not have to be changed.

JSATTupleLatticeElement.java

This class was updated to work with the node types (String not ASTNode) used in the JSATLatticeElement. Additionally a routine was added to be able to identify if a node existed in the lattice. If the value of a global is not in the lattice, we pull it from the initialized values in the specification. Normally we would trigger off the lattice's default value, but since the values were implemented as an open range of integers, there isn't a simple way to signify a unique default value.

JSATAnalysisDefinition.java

The code holds the transfer function for the MethodInvocation which executes the actions. All events are matched against each MethodInvocation. If a match is made, then the actions are executed for each.

JSATAnalysis.java

This holds the code that kicks off the specification parser and the visit function that is called for every MethodInvocation. All events are matched against each MethodInvocation. If a match occurs, all guard conditions are checked and errors generated as needed.

5. Future Works

These areas are ripe for future development:

1) Understand how to make the flow analysis engine produce the correct behavior in Sample2.
2) Implement regular expression matching.
3) Apply pattern matching against all code not just against MethodInvocation.
4) Extend the specification language to work with 'captures' or values allowing for globals to reflect actual program values, not only states defined in the specification.
5) Extend the flow analysis engine for interprocedural calls and reachability analysis.

6. Usage Notes

A full system path to the specification must be passed into the JSATAnalysis object when constructed. However, it is parsed when the tool is run in the Eclipse Debug session. Thus, the specification can be modified without having to recompile/relaunch the eclipse debug session.

References

[1] R. DeLine and M. F¨ahndrich. The fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004.
[2] BLAST tutorial. October 2005. http://embedded.eecs.berkeley.edu/blast/doc/blast.pdf.