The LIME Interface Specification Language and Runtime Monitoring Tool*

Kari Kähkönen, Jani Lampinen, Keijo Heljanko, and Ilkka Niemelä

Helsinki University of Technology TKK
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
ktkahkon@tcs.hut.fi, jani.lampinen@gmail.com,
{Keijo.Heljanko,Ilkka.Niemela}@tkk.fi

Abstract. This paper describes an interface specification language designed in the LIME project (LIME ISL) and the supporting runtime monitoring tool. The interface specification language is tailored for the Java programming language and supports two kinds of specifications: (i) call specifications that specify requirements for the allowed call sequences to a Java object instance and (ii) return specifications that specify the allowed behaviors of the Java object instance. Both the call and return specifications can be expressed with Java annotations in several different ways: as past time LTL formulas, as (safety) future LTL formulas, as regular expressions, and as nondeterministic finite automata. We also describe the supporting LIME interface monitoring tool which is an open source implementation of runtime monitoring for the interface specifications implemented using AspectJ.

1 Introduction

The interface specification language (LIME ISL) developed in the LIME project (`http://lime.abo.fi/`) is a lightweight formal method for defining behavioral interfaces of Java objects. The approach is supported by an open source implementation of a runtime monitoring tool automatically generating AspectJ [1] aspects to monitor that given interface specifications are not violated.

The aim of the LIME ISL is to enable a convenient way for the specification of behavioral aspects of interfaces in a manner that can be efficiently supported by tools. The aim is to extend the *design by contract* [2] approach to software development supported by approaches such as the Java Modeling Language (JML) [3] to behavioral aspects of interfaces. The idea is to divide the component interface to two parts in an assume/guarantee fashion: (i) call specifications (component environment assumptions) that specify requirements

* Work financially supported by Tekes - Finnish Funding Agency for Technology and Innovation, Conformiq Software, Elektrobit, Nokia, Space Systems Finland, Academy of Finland (projects 112016,126860,128050), and Technology Industries of Finland Centennial Foundation.

for the allowed call sequences to a Java object instance and (ii) return specifications (component behavior guarantees) that specify the allowed behaviors of the Java object instance. Both the call and return specifications can be expressed as Java annotations in several different ways: as past time LTL formulas, as (safety) future LTL formulas, as regular expressions, and as nondeterministic finite automata.

Our work also draws motivation from runtime monitoring tools such as MOP [4] and Java PathExplorer [5] as well as the tool of Stolz and Bodden [6] in the tool implementation techniques. However, unlike general event based monitoring approaches LIME interface specifications are more structured and the approach can be seen as a more disciplined approach to specifying runtime monitors for the software system of interest. For example, in our approach each behavioral interface is divided into call specifications and return specifications (assumptions/guarantees). Now a violation of a call specification is always a violation of the caller of the interface, while a violation of a return specification is the fault of the called Java class instance. This approach also allows the closing of open systems in testing by automatically generating test stub code directly from the interface specifications.

We fully agree with the Jackson and Fekete [7] stating: "Formal descriptions must be lightweight; this means that software developers should not have to express everything about the system being developed, but can instead target formal reasoning at those aspects of the system that are especially risky." The LIME ISL tries to achieve this goal by allowing partial specification of behavioral interfaces unlike model based design approaches that usually require modelling a large part of the design in order to be genuinely useful. This is achieved by allowing partial and incremental descriptions of the interfaces that can be made richer as needed.

Another source of inspiration for the design of the LIME ISL has been the rise of standardized specification languages in the hardware design community such as IEEE 1850 - Property Specification Language (PSL) [8]. One of the key features of PSL is the inclusion of both temporal logic LTL as well as regular expressions in the specification language provided for the user. This combination of several specification methods provides a choice of a convenient notation for specifying the different properties at hand. This is one of the reasons why LIME ISL supports past time LTL formulas, (safety) future LTL formulas, regular expressions, as well as nondeterministic finite automata. The inclusion of future time LTL was also motivated by the need to directly reuse specifications from model checking in the runtime monitoring context.

2 Interface Specifications

The core idea of the LIME interface specification language is to provide a declarative mechanism for defining how different software components can interact through interfaces in a manner that can be monitored at runtime. These interactions can be specified in two ways: by *call specifications* (CS) which define

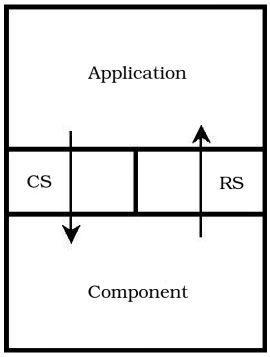

Fig. 1. The interaction model

how components should be used and by *return specifications* (RS) which define how the components should respond. If a call specifications is violated, the calling component can be determined to be incorrect and, respectively, if the called component does not satisfy its return specifications, it is functioning incorrectly. This interaction model between components is illustrated in Fig. 1.

To get an overview of the specification language, let us consider the following example where LIME interface specifications are written for a simple log file interface.

```
1: @CallSpecifications(
2:  regexp = { "FileUsage ::= (open(); (read() | write())*; close())*" },
3:  valuePropositions = { "validString ::= (#entry != null)" },
4:  pltl = { "ProperData ::= G (write() -> validString)" }
5: )
6: @ReturnSpecifations(
7:  valuePropositions = {
8:  "okLength ::= #this.length() == #pre(#this.length()+#entry.length())"
9:  },
10: pltl = { "ProperWrites ::= G (write() -> okLength)" }
11: )
12: public interface LogFile {
13:   public void open();
14:   public void close();
15:   public String read();
16:   public void write(String entry);
17:   public long length();
18: }
```

In this example the call specifications describe the allowed call orders of the interface methods and the valid input values to the write method. The return specifications describe how the implementation of the LogFile should behave when write method is called. Call specifications are similar in spirit to JML preconditions, while return specifications are similar in spirit to JML postconditions.

The main difference is that LIME ISL allows also to specify temporal aspects of an interface (behavior over several method calls) while JML concentrates on the behavior of a single call.

In the LIME interface specification language, the specifications are written as annotations to Java interfaces or classes. The two main annotations that can be written are `@CallSpecifications` and `@ReturnSpecifications`. The annotations for call and return specifications consists of a set of atomic propositions and actual specifications. Atomic propositions are used to make claims about the program execution and the state of the program. These atomic propositions are subdivided into three classes: *value propositions*, *call propositions* and *exception propositions*.

Value propositions are claims about the state of the program and the values of arguments given to the observed methods. A value proposition can be seen as a native language expression that should be free of side effects and that is true if and only if the native language expression evaluates to true. In value propositions there are several reserved words that give special semantics for the propositions. Keyword `#this` allows referencing the instance of the annotated interface (see line 8 of the example), while keyword `#result` allows referencing the return value of a method. Keyword `#pre[`*primitive type*`](`*Java expression*`)` makes it possible to reference an *entry* value in return specifications after the actual method has been executed. This allows specifications that describe how some value must change during execution of the observed method. Primitive type `int` is the default type and therefore it is not necessary to explicitly write it as shown in line 8 of the example specification. By writing `#<argument>`, it is possible to reference the arguments given to an observed method (see line 3 of the example).

Call propositions are claims about method execution. A call proposition is true if and only if the method named in the proposition is currently executing (e.g., the body of `open()` is executing at the top of the call stack). Argument overloading is not yet supported in the current version and therefore the call propositions refer to all methods that have the same name regardless of their argument types. In line 2 of the example, the methods named in the FileUsage specification are call propositions.

Exception propositions are claims about thrown exceptions. Specifically, they are propositions available in return specifications that are true if and only if the observed method threw a specific exception (e.g., RuntimeException has been thrown by a method).

The defined call and return specifications use these atomic propositions to describe the expected properties of the interface components and they can be written in three complementary ways: by using regular expressions, nondeterministic finite automata (NFA) and a large supported subset of Linear Temporal Logic with Past (PLTL).

In LIME ISL the user does not have to explicitly define when the specifications are observed but the observers see an execution trace of the program where the observation points are implicitly defined by the call propositions that are used in the observed specification. We will refer to these observations points

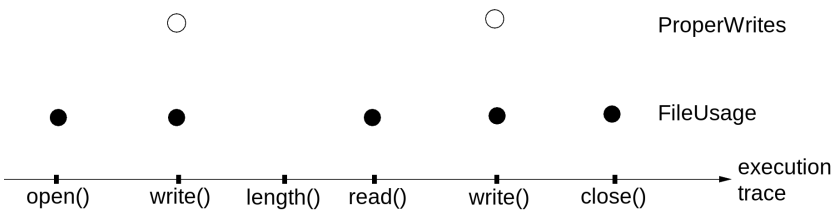

Fig. 2. An execution trace of the LogFile example

as events. There are two types of events that differ slightly from each other: call events and return events.

Call events occur right before a call to a method that has been used as a call proposition in the corresponding call specification. This means that the respective call specification is observed at this point of the execution trace. Return events are similarly determined by the call propositions in the corresponding return specification. There is, however, a difference how these events are observed. In order to allow the return specifications to contain value propositions that use `#pre` to describe values at the entry point of the called method, the implementation uses a history variables technique to store the required values at the entry point of a call. It then uses these history variables to monitor the return specification at the return of the called method, where all value propositions are evaluated and the monitored return event happens.

Figure 2 illustrates the concept of events in one possible execution trace of a system that uses the LogFile interface. The filled circles in the picture are call events and the empty circles are return events. The semantics is then that the observers are fed their own linear event sequences and based on that event sequence, the observer can detect failing specifications during system runtime.

3 The Runtime Monitoring Tool

The LIME Interface Monitoring Tool is our first software tool for the introduced specification language. It allows monitoring the specifications at runtime to determine if some component violates the given specifications. Multi-threaded programs are not supported in the current version. An architectural overview of the tool is given in Fig. 3.

The monitoring tool works by reading the specification annotations from the Java source files. The specifications are then translated into deterministic finite state automata that function as observers. These automata are translated into runnable Java code and AspectJ (`http://www.eclipse.org/aspectj/`) is used to weave the code into the original program that is being tested. This results in an instrumented runtime environment where the observers are executed at the timepoints discussed in the previous section.

Spoon [9], the dk.brics.automaton (`http://www.brics.dk/automaton/`) package and SCheck [10] are adopted as third-party software. Spoon is used for an-

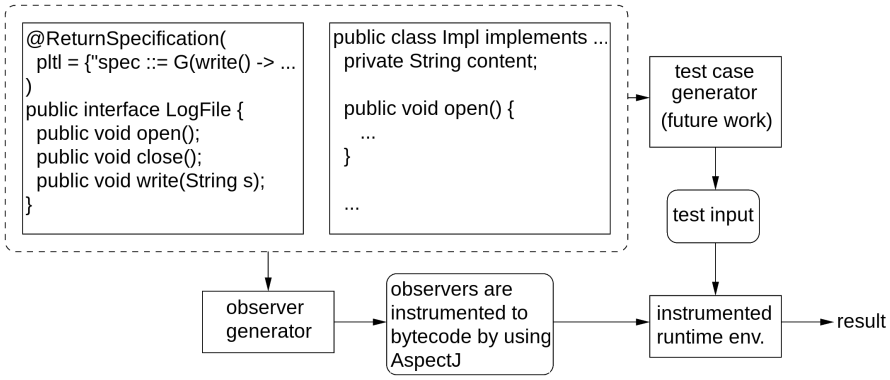

Fig. 3. Architecture of the LIME interface monitoring tools

alyzing the program and the dk.brics.automaton package is used for internal representation and manipulation of regular expression checkers. SCheck is used for converting future time LTL subformulas into finite state automata. The approach of [11] using synthesized code with history variables is used for past time subformulas, while for the future part the tool SCheck is used to encode informative bad prefixes [10] of future LTL formulas to minimal DFA. Our implementation currently allows the use of past-time subformulas LTL in future-time LTL formulas but not vice versa. More implementation details for an early version of the tool can be found from [12].

3.1 Closing Partially Implemented Systems

The call specifications can be used to automatically generate stub code that closes an open system from above. In other words, it is possible to generate a stub code implementation of the application part shown in Fig. 1. so that it uses a component that we want to test. The stub code generates test sequences to the component and the call specifications are used to filter out violating method call sequences.

The LIME runtime monitoring tool supports this idea by providing a generator that creates such stub code implementations. The generated code selects the methods to be called non-deterministically and generates random argument values. The number of method calls is limited to a test depth that can be selected by the user. To avoid reporting call specification violations that are caused by the stub code, such violations are set to be identified as inconclusive test runs.

Purely random environment is likely to generate a large number test runs that are inconclusive. For this reason the described approach is intended to be used with a testing tool based on dynamic symbolic execution similar to jCUTE [13] and Pex [14]. This prevents the generation of multiple instances of the same test case and also allows us to generate test cases that are difficult to obtain by using

only random testing. The implementation of the test case generator is work in progress.

As an example of the stub code generation, let us consider the LogFile interface again. The class TestDriver shown below has been generated by the monitoring tool and it consists of a simple loop (line 8) where one of the methods in the LogFile interface is called. The ExceptionOverride class (line 7) is used to set the call specification violations to be identified as inconclusive test runs. The random values generated by the stub code can be replaced by input values received from the test case generator when the test generator tool is used.

```
1:   public class TestDriver {
2:    public static void main( String[] args ) {
3:      Random r = new Random();
4:      int testDepth = 0;
5:      FileImpl obj = new FileImpl();
6:      java.lang.String javalangString1;

7:      ExceptionOverride.setCallException(obj,
                                           InconclusiveException.class);
8:      while (testDepth < 5) {
9:        testDepth++;
10:       int i = r.nextInt(5);
11:       switch (i) {
12:         case 0: obj.length(); break;
13:         case 1: javalangString1 = RandomString.getString(r);
14:                 obj.write(javalangString1); break;
15:         case 2: obj.read(); break;
16:         case 3: obj.close(); break;
17:         case 4: obj.open(); break;
18:       }
19:     }
20:   }
21: }
```

4 Conclusion

We have described the LIME interface specification language and interface monitoring tool, available from: `http://www.tcs.hut.fi/~ktkahkon/LIMT/`. There are interesting topics for further work. The SCheck tool could be extended to allow free mixing of future and past LTL subformulas. The implementation of a test case generator that can be used with the automatically generated stub code is currently work in progress. We are also investigating how the test case generation process can be guided to achieve good interface specification coverage with a small number of test cases. Adding support for multi-threaded programs is one important topic for future work. We are also working on porting the specification language to the C programming language. Another more far reaching research direction would be to investigate interface compatibility of different interfaces along the lines of [15].

Acknowledgements We thank our LIME research partners at Åbo Akademi University and colleagues at TKK for feedback on earlier versions of the LIME interface specification language, and the anonymous referees of RV 2009 for valuable suggestions for improving the paper.

References

1. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings. Volume 2072 of Lecture Notes in Computer Science., Springer (2001) 327–353
2. Meyer, B.: Applying "design by contract". IEEE Computer **25**(10) (1992) 40–51
3. Burdy, L., Cheon, Y., Cok, D., Ernst, M.D., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Software Tools for Technology Transfer **7**(3) (June 2005) 212–232
4. Chen, F., Rosu, G.: MOP: An efficient and generic runtime verification framework. In Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S., eds.: OOPSLA, ACM (2007) 569–588
5. Havelund, K., Rosu, G.: An overview of the runtime verification tool Java PathExplorer. Formal Methods in System Design **24**(2) (2004) 189–215
6. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. Electr. Notes Theor. Comput. Sci. **144**(4) (2006) 109–124
7. Jackson, D., Fekete, A.: Lightweight analysis of object interactions. In Kobayashi, N., Pierce, B.C., eds.: TACS. Volume 2215 of Lecture Notes in Computer Science., Springer (2001) 492–513
8. IEEE: IEEE Standard 1850 - Property Specification Language (PSL) (2005)
9. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program Analysis and Transformation in Java. Research Report RR-5901, INRIA (2006)
10. Latvala, T.: Efficient model checking of safety properties. In Ball, T., Rajamani, S.K., eds.: Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings. Volume 2648 of Lecture Notes in Computer Science., Springer (2003) 74–88
11. Havelund, K., Roşu, G.: Efficient monitoring of safety properties. Software Tools for Technology Transfer (STTT) **6**(2) (2004) 158–173
12. Lampinen, J.: Interface specification methods for software components. Research Report TKK-ICS-R4, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland (June 2008)
13. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In Ball, T., Jones, R.B., eds.: CAV. Volume 4144 of Lecture Notes in Computer Science., Springer (2006) 419–423
14. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In Beckert, B., Hähnle, R., eds.: TAP. Volume 4966 of Lecture Notes in Computer Science., Springer (2008) 134–153
15. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In Henzinger, T.A., Kirsch, C.M., eds.: EMSOFT. Volume 2211 of Lecture Notes in Computer Science., Springer (2001) 148–165