TKK Reports in Information and Computer Science

Espoo 2009 TKK-ICS-R18

TESTING A JAVA CARD APPLET USING THE LIME INTERFACE TEST BENCH

A Case Study

Roland Kindermann

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D’HELSINKI

TKK Reports in Information and Computer Science

Espoo 2009 TKK-ICS-R18

TESTING A JAVA CARD APPLET USING THE LIME INTERFACE TEST BENCH

A Case Study

Roland Kindermann

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 451 1
Fax +358 9 451 3369
E-mail: series@ics.tkk.fi

ISBN 978-952-248-079-8 (Print)
ISBN 978-952-248-080-4 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://lib.tkk.fi/Reports/2009/isbn9789522480804.pdf

TKK ICS
Espoo 2009

Abstract: The LIME Interface Test Bench is a collection of tools that allow to compile programs in a way such that they monitor interface specifications at runtime in Java and C programs. Specifications can be made using the LIME specification language. Another part of the LIME Interface Test Bench is the LIME Concolic Testing tool (LCT), which uses a combination of concrete and symbolic execution to explore large number of control flow paths in a program or parts of a program.

The Java Card technology allows to use a limited subset of Java to develop applets that run on Smart Cards. These applets communicate with an off-card application using a simple packet-based protocol.

This report describes a case study, in which the LIME Interface Test Bench was used to test a Java Card applet. The case study uses the "logical channels demo" applet, which is part of the Java Card Development Kit [1]. Ten different specifications were added to this applet. In order to use the applet in a realistic environment, an off-card application for the applet was developed. This off-card application was tested using LCT.

Keywords: Testing, runtime monitoring, concolic testing, LIME Interface Test Bench, Java Card

CONTENTS

1 INTRODUCTION

The LIME Interface Test Bench provides two main functionalities: It allows to monitor specifications in C and Java programs at runtime and it allows to systematically test a large number of different control flow paths in a program using concolic testing. For this report, version 0.3.0 of the LIME Interface Test Bench was used.

The Java Card technology [1, 4] can be used to develop Java programs that run on Smart Cards. These programs, called applets, receive commands from and return values to an off-card application using a simple packet-based protocol.

This report describes a case study in which the LIME Interface Test Bench was used to test a Java Card applet. The aim of the case study was to determina how usable and useful the LIME Interface Test Bench tools are for testing Java Card applets and to identify potential improvements of the tools. For the case study, an applet called "logical channels demo", which is part of the Java Card Development Kit [1], was used and extended by LIME specifications. In order to be able to test the applet in a realistic environment, an off-card application for the applet was developed. This off-card application was tested using LCT.

This report is structered as follows: Section 2 describes the used technologies and tools, i.e. the LIME Interface Test Bench and Java Card. In particular, it gives an informal overview over the LIME specification language, a brief introduction to LCT, a description of the logical channels demo and an instruction on how to monitor LIME specifications in Java Card applications. Section 3 describes the actual case study which consists of two main parts, namely the specifications for the logical channels demo and the development and testing of the off-card application. The last section sums up the experiences that have been made during the case study and suggests some useful features that could be added to the LIME Interface Test Bench.

2 USED TOOLS AND TECHNOLOGIES

This section introduces the technologies and tools that were used in the case study. The first two subsections describe two different parts of the LIME Interface Test Bench. Section 2.1 describes how specifications can be made using the LIME specification language and how these specifications can be monitored at runtime. Section 2.2 describes the LIME Concolic Testing tool (LCT), which tries to explore a large number of different control flow paths in a program.

The LIME Interface Test Bench was used to test a Java Card applet and a Java Card off-card application. Section 2.3 describes the Java Card technology in general and Section 2.4 describes how LIME specifications can be monitored in Java Card applets despite the fact that they can only use a limited subset of the Java features. The last subsection, Section 2.5, describes the applet that was used in this case study.

2.1 LIME specifications

This section informally describes parts of the LIME specification language. A more complete description including all features and the exact definition of the semantics of specifications can be found in [8].

LIME specifications for Java programs are made in the form of annotations to a class or an interface. In principle, there are two types of specifications: ones that describe the correct use of the class (or classes implementing the interface if the specifications are made for an interface) and ones that describe correct behavior of the class itself. These two types of specifications roughly correspond to two types of specifications existing in the LIME specification language: call specifications and return specifications. Technically, call specifications are monitored when methods are called while return specifications are monitored when methods return. Thus, call specifications are well-suited for specifying legal behavior of the method callers while return specifications are well-suited for specifying legal behavior of the methods. There are however situations in which specifications about the legal behavior of a method caller can not be made as call specifications but only as return specifications. One example where this is the case is described Section 3.1.

There are three ways to make specifications in the LIME specification language: linear temporal logic with past (PLTL), regular expressions and nondeterministic finite automata (NFAs). NFA specifications are not covered in this report. Regular expression specifications may use the following operators: concatenation (denoted by "`;`"), union ("`|`") and closure ("`*`"). In addition, parentheses can be used and expressions in the form of "`a+`" can be used as abbreviation for "`a ; a*`".

Figure 1 shows a simple example of an interface with a regular expression call specification. The interface describes a file object that has methods for opening and closing a file and reading from and writing to the file. The specification describes that order in which the methods may be called. First, `open` must be called. Then, `read` and `write` may be called arbitrarily often in any order. The method `close` may be called as well but then `open` has to be called again before `read` and `write` may be called again. The specification describes legal behavior of the method caller and is thus made as a call specification.

The example in Figure 1 suggests that the LIME specification language uses regular expressions over method calls. This however is not quite true. Instead, regular expressions over propositional formulas are used. The propositional formulas can use common propositional logic operators and three different kinds of propositions: call propositions, value propositions and exception propositions. Call propositions are denoted by a method name followed by one opening and one closing parenthesis. In the example, the call propositions `open()`, `read()`, `write()` and `close()` are used. Call propositions are true if the corresponding method is currently executed. Call propositions however identify a method solely by its name and thus do not differentiate between overloaded methods.

Value propositions are statements about the values of variables like method arguments, return values, member variables or static variables. They are surrounded by "`<{`" and "`}>`". For instance, the value proposition `<{Some-`

```
@CallSpecifications(
    regexp = {
        "FileUsage ::= (open() ; (read() | write())* ;
            close())*"
    }
)
public interface LogFile {
    public void open();
    public void close();
    public String read();
    public void write(String s);
    public int length();
}
```

Figure 1: A simple interface with a LIME specification. Taken from [9].

`class.x != null}>` is true iff the static variable `x` of the class `Someclass` is `null`. Value propositions can refer to arguments of methods by preceding their name with a “`#`” and to fields of the current instance by preceding their name with “`#this.`”. Furthermore, return specifications, which are monitored when methods return, can refer to the value that a given expression had when the method was called using the keyword `#pre` and to the return value of the method using the keyword `#return`. For example, the value proposition `<{#this.y == #pre(#this.y)}>` in a return specification is true iff the value of the field `y` did not change during the execution of the method at which the return specification is monitored.

Like the name suggests, exception propositions refer to an exception type and are true iff an exception of that type is thrown. Exception propositions however are not used in this case study.

In the example in Figure 1, the regular expression only uses propositional formulas that consist of a single call proposition. The specification could be extended as follows:

```
@CallSpecifications(
    regexp = {
        "FileUsage ::= (open() ; (read() | (write() &&
            <{#s != null}>))* ; close())*"
    }
)
```

Now the propositional formula “`write()`” has been replaced with the formula “`(write() && <#s != null>)`. While the old propositional formula was true whenever `write` was executed, the new formula is only true when `write` is executed and in addition the argument `s` does not equal `null`. Off course, much more complex propositional formulas can be used. Table 1 shows some of the propositional logic operators supported by the specification language LIME and their notation.

So far, inline notations were used for all propositions. This however can become rather confusing when more complex value propositions are used. It is also possible to define value propositions separately using the `valuePropositions` field of the specification annotation. For instance, the

Name	Common notation	LIME notation
Negation	$\neg$	`!`
And	$\wedge$	`&&`
Or	$\vee$	`||`
Implication	$\Rightarrow$	`->`
Equivalence	$\Leftrightarrow$	`<->`

Table 1: Some propositional logical operators supported by the LIME specification language.

Name	Notation	Description
Globally	G p	`p` must be true now and every time the specification is monitored in the future.
Yesterday	Y p	The specification must have been monitored before and `p` must have been true the last time the specification has been monitored.
Once	O p	Either `p` is currently true or the specification must have been monitored before and `p` must have been true at some point in the past when the specification was monitored.
Since	p S q	Either `q` is true now or there must have been a time in the past when the specification was monitored and `q` was true and `p` has been true at every time the specification was monitored after that time.

Table 2: Some PLTL operators supported by the LIME specification language. `p` and `q` can be replaced with arbitrary PLTL formulas.

annotation from the example in could be modified to

```
@CallSpecifications(
    valuePropositions = {"argprop ::= <{#s != null}>"},
    regexp = {
        "FileUsage ::= (open() ; (read() | (write() &&
            argprop))* ; close())*"
    }
)
```

A similar mechanism exists for call propositions. As they however tend to be rather short, it is of less use in practice.

Similar to regular expression specifications, PLTL specifications can use call and value propositions and a propositional logic operators. They can in addition use various PLTL operators. Some of these operators are shown in Table 2. Various PLTL specifications of different complexity are described in Section 3.1.

Whether or not a specification is observed by a given program execution may greatly depend on the points in time when it is monitored. For example, if the call specification from Figure 1 is monitored at any time when neither of the methods `open`, `close`, `read` and `write` is executed, it will be violated. A specification is automatically monitored at every method that is referred

```
 1 public class LCTtest{
 2     public static void main(String args[]){
 3         int testvalue = LCT.getInteger();
 4         doSomething(testvalue);
 5     }
 6
 7     public void doSomething(int arg){
 8         if(arg == 42)
 9             error();
10         else
11             ...
12     }
13 }
```

Figure 2: A simple program using LCT.

to by a call proposition used in the specification. Thus, the specification in the example in Figure 1 is monitored at the methods `open`, `close`, `read` and `write` but not at the method `length`. Hence, calling `length` would not result in a violation of the specification. It is however possible to force a specification to be monitored at other methods as well using an `@Observe` annotation. Thus, if however line 11 was replaced by

```
@Observe(specs = {"FileUsage"})
public int length();
```

then the specification would also be observed when `length` was called and consequently calling length would result in a violation of the specification.

2.2 The LIME Concolic Testing tool

Part of the LIME Interface Test Bench [8, 2] is the LIME Concolic Testing tool (LCT) [2, 7]. LCT allows to test a program or parts of a program with different input values. Figure 2 shows a small program that uses LCT to test the method `doSomething`. In line 3, the program retrieves an integer value from LCT. It then passes that value to the method `doSomething`. In the example, calling `testedMethod` with the argument 42 causes an error while any other value is fine. If the example program is compiled like an ordinary Java program, the `LCT.getInteger()` returns a random value. Thus, the `doSomething` method can be tested with various arguments by executing the program multiple times. As the values however are selected randomly, it is quite unlikely that the value 42, which causes an error, is picked. Hence, it is quite unlikely that the error in the `doSomething` method is discovered by executing the program a reasonal number of times.

LCT can use concolic testing [10], a combination of symbolic and concrete execution to explore the possible behaviours of the tested code more systematically. After compilation, LCT adds code that performs symbolic execution to a program in a step called instrumentation. When the intrumented program is executed for the first time, `LCT.getInteger` returns a random integer just like in the non-instrumented program. In the further execution of the program, the code added during the instrumentation keeps track of how the program uses the value returned by `LCT.getInteger`. In particular, it keeps track of the value's effects on the control flow. For instance, when the control flow reaches line 11 in the example, the symbolic execution code determines that the current control flow path is taken iff `LCT.getInteger` returns a value other than 42. Analogously, a constraint is added for every `if` condition that depends on a value returned by a call to `LCT.getInteger`. In subsequent executions, LCT then uses the generated constraints to make the control flow follow a yet unexplored path by making `LCT.getInteger` return according values. In order to find values that make the program follow a given control flow path, LCT can use either of the SMT solvers Boolector [3] and Yices [5]. In some cases however, LCT fails to follow the control flow path it intended to follow. If the instrumented program is executed often enough, LCT explores all control flow paths it can explore.

2.3 Java Card applets

Jave Card [1, 4] applets are Java programs that can be run on smart cards. There are two substantial differences between Java Card applets and ordinary Java applications. The first difference is that Java Card Applets can only use a very limited subset of the Java features. For example, the basic data types `long`, `float`, `double`, `char` and `String` are not supported and the support of `int` is optional. Also, multidimensional arrays and most standard Java classes are not supported. Secondly, Java objects on smart cards can be stored in permanent memory and thus may exist for quite long time and even for the entire lifespan of the smart card.

Java Card applications usually consist of two parts: an on-card applet and an off-card application. The applet and the off-card application communicate through the smart card's interface by exchanging data packets called APDUs. First, the off-card application sends a command APDU which consists of a mandatory header and an optional body. The header specifies a class of instruction, an instruction and two parameter bytes. The optional body can contain additional data and allows the off-card application to specify the number of bytes of data that it expects in the reply.

The on-card applet can only become active after receiving a command APDU. The applet then executes the requested operation and sends a response APDU. Like the command APDU, the response APDU can contain data in an optional body. Analogously to the mandatory header of the command APDU, the response APDU contains a mandatory two byte status word which indicates the outcome of the operation. If no errors occur, the status word equals the constant `ISO7816.SW_NO_ERROR`. Otherwise, the status word can either have one of the error codes defined in the ISO 7816 standard or a custom error value defined by the applet developer.

Multiple applets can be installed on the same smart card. Before the off-card application can communicate with one of the applets, that applet must be selected. Usually, only one applet can be selected at a time. Using so-called logical channels, it is however also possible to select multiple applets at once.

Java Card applets are implemented in as subclasses of `javacard.framework.Applet`. Typically, a Java Card applet implements the following methods:

- install – A static method that is called, when the applet is installed on the smart card. The install method then creates an instance of the applet and initializes the applet.
- select – The select method is called when the applet is selected.
- deselect – The deselect method is called when the applet is deselected.
- process – The process method is called when the applet receives a command APDU from the off-card application.

Java Card applets are compiled in two steps. First, the normal Java compiler is used to compile the source code into a class file. Then, the class files are converted into a CAP file using a converter tool which is part of the Java Card Development Kit. This converter also ensures that only the limited supported subset of the Java features is used.

2.4 Monitoring LIME specifications in Java Card applets

Part of the Java Card Development Kit [1, 4] are two simulators that allow to run Java Card applets on an ordinary PC. The first simulator is called the C-language Java Card Reference Implementation (cref), which can load and execute CAP files. However, only applets that use only the restricted subset of Java features supported by Java Card can be converted into CAP files. As programs that monitor LIME specifications use Java features not supported by Java Card, they not be converted into CAP files and thus not be run in the cref.

The second simulator is called Java Card WDE (jcwde) and is implemented in Java. Unlike cref, jcwde use class files instead of CAP files. Thus, it is possible to also use language features that are not supported by Java Card in applets run in the jcwde. This makes it possible to run applets that monitor LIME specifications in jcwde.

Without an off-card application, a Java Card applet does nothing. Thus, it is necessary to also simulate the off-card application when simulating the applet. This can e.g. be done using the apdutool, which is part of the Java Card Development Kit. The apdutool connects to either simulator using TCP/IP. Then, it loads a sequence of command APDUs from a file specified by the user and sends them to the simulator. The simulator hands the command APDUs to the applet and sends the returned response APDUs back to the apdutool. Both command and response APDUs are printed to the command line by the apdutool.

One difficulty when running applets with LIME specifications in jcwde is that jcwde catches all exceptions and simply returns the status word 0x6F00 ("SW_UNKNOWN") without giving any indication which exception was thrown or where it was thrown. Thus, when the status word is SW_UNKNOWN, it is difficult to determine whether this was caused by the violation of a specification or by any other exception like e.g. a NullPointerException caused by a programming error. This problem can be solved by running jcwde in the Java command line debugger jdb and instructing jdb to break whenever an exception of type `fi.hut.ics.lime.aspectmonitor.SpecException` is thrown.

Another difficulty arises from the fact that specification monitoring code is added at every call site of a method for which a LIME specification has been made. This implies that the specifications only work correctly if all parts of the code from which relevant methods are called are compiled with the LIME Interface Test Bench. However, as the source code of jcwde is not publicly available, it is not possible to compile jcwde with the LIME Interface Test Bench. Thus, specifications on the methods called from within the simulator like `install`, `select`, `deselect` and `process` do not work correctly. This difficulty can be circumvented by implementing all such methods as wrapper methods for other methods that do the actual work and on which specifications can be made.

2.5 The logical channels demo

The logical channels demo is one of several demos that are part of the Java Card Development Kit. The idea behind the logical channels demo is that the smart card is part of a device that allows the user to connect to a network for a certain fee. The network is divided into several areas and the user has a home area, in which the fee is lower than in the rest of the network. The smart card on which the logical channels demo is installed keeps track of the user's account's balance.

The logical channels demo consists of two applets: the `AccountAccessor`, which manages the user's account and the `ConnectionManager`, which receives the state of the network connection from the off-card application and debits the account accordingly. The main purpose of the logical channels demo is to illustrate how these two applets can be active at the same time and how the off-card application can use logical channels in order to specify, which of the applets it wants to address.

Writing LIME specifications for the original version of the logical channels demo is difficult mainly due to the fact that Java Card Applets use special Java Card mechanisms to receive arguments from and return values to the offcard application.

Figure 3 shows an example of how arguments are received from the off-card application in the logical channels demo. The figure shows the `credit` method of the `AccountAccessor`. This method is called by the `process` method of the `AccountAccessor` and receives an `APDU` object, which provides several methods related to receiving and sending APDUs. In lines 2 to 4, the `credit` method receives the argument bytes from the command APDU. It then checks that two bytes of data were received and throws an exception if this is not the case. In line 9 the two bytes of data are combined to to one `short` which indicates the amount by which the user's balance will be increased. Then, it is checked whether the resulting amount the is too large and if this is the case, one of two exceptions is thrown. Otherwise, `credit` method increases the user's balance by the specified amount. Due to the fact that the argument of the credit method is received in it's body, it can not be referenced the usual way in value propositions in LIME specifications.

```
private void credit(APDU apdu){
    byte[] buffer = apdu.getBuffer();
    byte numBytes = buffer[ISO7816.OFFSET_LC];
    byte byteRead = (byte) apdu.setIncomingAndReceive();

    if((numBytes != 2) || (byteRead != 2))
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    short creditAmount = (short) ((short)
        (buffer[ISO7816.OFFSET_CDATA] << (short) 8) |
        (short) (buffer[ISO7816.OFFSET_CDATA + 1]));

    if((creditAmount > MAX_BALANCE)
        || (creditAmount < (short) 0)){
        ISOException.throwIt(
            SW_INVALID_TRANSACTION_AMOUNT);
    }

    if((short) (balance + creditAmount) > MAX_BALANCE
       || (short) (balance + creditAmount) < (short) 0){
        ISOException.throwIt(SW_MAX_BALANCE_EXCEEDED);
    }

    JCSystem.beginTransaction();
    balance = (short) (balance + creditAmount);
    JCSystem.commitTransaction();
}
```

Figure 3: The original code of the `credit` method.

```
private short credit(short creditAmount){
    if((creditAmount > MAX_BALANCE) || (creditAmount < (short) 0)){
        return SW_INVALID_TRANSACTION_AMOUNT;
    }

    if ((short) (balance + creditAmount) > MAX_BALANCE || (short) (balance + creditAmount) < (short) 0){
        return SW_MAX_BALANCE_EXCEEDED;
    }

    JCSystem.beginTransaction();
    balance = (short)(balance + creditAmount);
    JCSystem.commitTransaction();

    return ISO7816.SW_NO_ERROR;
}
```

Figure 4: The `credit` method after having been rewritten in a way that makes it easier to make specifications.

Another difficulty is caused by the way exceptions are thrown in Java Card applications. Instead of using the standard `throw` keyword, the Java Card API provides its own exception throwing mechanism. Instead of real exceptions, Java Card applications use two-byte error codes, which are passed to the off-card application as status words. Exceptions are thrown by calling the method `ISOException.throwIt`. Thus, Java Card exceptions can not be referenced by exception propositions in LIME specifications.

As only a few rather simple specifications could be made for the logical channels demo without referencing arguments or exceptions, the code of the logical channels demo was modified in a way such that the Java Card specific mechanisms for receiving arguments and throwing exceptions were only used in the `process` methods of the applets. After the modifications, all other methods receveived their arguments through the standard Java argument passing mechanism and return status words rather than throwing the exceptions themselves.

Figure 4 shows the code from Figure 3 after the modification. The entire argument receiving code has been moved to the `process` method and thus lines 2 to 9 have been removed. Also, the return type has been changed from `void` to `short` and the uses of the Java Card exception mechanism in lines 13 and 18 have been replaced with return statements.

Another property of the channels demo that makes it difficult to add specifications is that all fields of both applets are private and thus can not be accessed in specifications. As it was tried to add specifications while changing the source code as little as possible, only the field that stores the user's balance was given package visibility in order to make it visible to specifications.

3 THE CASE STUDY

This section describes the actual case study, which consists of two main parts. The first part was the development of ten specifications for the logical channels demo. This part is described in Section 3.1. The second part of the case study was the development of an off-card application for the logical channels demo. The off-card application was tested using the LIME Concolic Testing tool. The off-card application and the LCT test are described in Section 3.2.

3.1 LIME specifications for the logical channels demo

After modifying the code logical channels demo, several LIME specifications were added. The aim of this process was to make specifications that would have been difficult to monitor without using the LIME Interface Test Bench. In particular, no specifications that could also easily be monitored using simple `assert` statements were made. Another case study in which LIME specifications are made to the logical channels demo can be found in [6].

The original code of the channels demo neither contains any specifications in natural language nor in any other form. Therefore, specifications had to be “made up”. The main purpose of the specifications was to illustrate the capabilities of the LIME specification language and not to capture the behavior of the logical channels demo as precisely as possible. Thus, the specifications are in some cases stricter than they would have to be and forbid behavior that would not result in any error.

As described in Section 2.1 there are two main types of specifications: ones that specify the correct usage of a method and ones that specify what the method does. In course of the case study five specifications of each type were made.

Specifying the correct usage of methods

Five specifications about the correct usage of methods were made. All of these specifications were made for methods in the `ConnectionManager` applet. Usually, specifications about the correct usage of methods are made as call specifications. This however was only possible for four of the specification.

Some methods of `ConnectionManager` applet may only be called when an `AccountAccessor` instance exists on the same smart card. If there is an `AccountAccessor` instance on the smart card, it can be retrieved using the static method `AccountAccessor.getAccount()`. If there is no `AccountAccessor` instance, that methods returns `null`. Thus, the following specification is violated iff no `AccountAccessor` instance exists:

```
pltl = {
    "AccountExists ::= G(<{AccountAccessor.getAccount()
       != null}>)",
    ...
}
```

As the specification does not contain any call propositions, it is never monitored automatically. Instead, an `@Observe` annotation for the `AccountEx-`

ists specification was added to all methods that require the existence of an AccountAccessor.

The methods setConnection, resetConnection and timeTick may only be called when the ConnectionManager applet is selected. The applet is selected iff select has been called at least once and deselect has not been called since the last time select has been called. Thus, whether or not the applet is selected could be determined using the formula !deselect() S select(). As described in Section 2.3, using the select and deselect methods in specifications does however not work due to the fact that they are called from within the jcwde simulator. The ConnectionManager applet however has methods called initState and clearState, which are exclusively called by select and deselect. Thus, they can be used as replacement for select and deselect in specifications.

Unlike in the first example, the methods at which the specification should be monitored were not specified using an @Observe annotation but in the specification itself:

```
pltl = {
    ...
    "Active ::= G((setConnection()
                  || resetConnection()
                  || timeTick())
                     -> (!clearState() S initState()))",
    ...
}
```

This specification is violated iff one of the methods setConnection, resetConnection and timeTick is executed at a time when the applet is not selected. In addition to these three methods, the specification is also monitored when clearState or initState() is executed. This is necessary in order to keep track of whether or not the "since" subformula is currently true. However, this implies that using @Observe annotations like for the AccountExists specification would not work. If (!clearState() S initState()) and @Observe annotations for setConnection, resetConnection and timeTick was used instead, the specification would be violated whenever clearState() is executed.

The methods timeTick(short newAreaCode) of the ConnectionManager applet is triggered by the off-card application every time unit. It receives one argument which contains the ID of the network area the user is currently in. This area code can be any short except for the constant ConnectionManager.INACTIVE_AREA which is reserved for other uses. This can be specified as follows:

```
pltl = {
    ...
    "TimeTickValidArgument ::= G(timeTick() ->
        <{#newAreaCode !=
        ConnectionManager.INACTIVE_AREA}>)",
    ...
}
```

The setConnection method is called, when the network connection is activated. When the setConnection method is called, the user already has to pay for the current time unit. In order to determine the price, the ConnectionManager however needs to know in which area of the network the user currently is. Thus, setConnection may only be called if the ConnectionManager knows about the area the user is currently in.

The off-card application sends the current area's code every time it triggers timeTick. Thus, setConnection may only be called if timeTick has been called before. If the ConnectionManager applet is deselected, the code of the current area is discarded. Hence, setConnection may be called if the applet was not deselected since the last time timeTick was called. This can be specified as follows:

```
pltl = {
    ...
    "AreaCodeSet ::= G(setConnection() ->
        (!clearState() S timeTick()))"
}
```

Dually to the setConnection method, there is a resetConnection method in the ConnectionManager applet, which is called when the network connection is deactivated. Although the requirements in the original ConnectionManager applet are less strict, it might make sense to require that the setConnection method may only be called when the network is inactive and the resetConnection method may only be called, when the network connection is active. Furthermore, both methods may only be called when the ConnectionManager applet is selected and the applet may not be deselected while the network connection is active. These requirements could be specified using a regular expression as follows:

```
regexp = {
    "ConnectionCallOrder ::= (initState() ;
        (setConnection() ; resetConnection())* ;
        clearState())*"
}
```

This specification however does not take into account that the setConnection method is not guaranteed to succeed. For instance, if the user's balance is too low to pay for the connection, a special status word indicating the failure is returned nothing else is done. In particular, the network connection remains inactive. In order to take this into account in the specification, it is necessary to differentiate between successful and unsuccessful calls to setConnection. This can be done using the following value proposition:

```
valuePropositions = {
    "success ::= (#result ==
        javacard.framework.ISO7816.SW_NO_ERROR)",
    ...
}
```

The value proposition success is true iff the current method returned javacard.framework.ISO7816.SW_NO_ERROR, which indicates that the method was executed successfully. Using the value proposition, the ConnectionCallOrder specification can be corrected as follows:

```
regexp = {
        "ConnectionCallOrder ::= (initState() ;
            ((setConnection() && !success)* ;
            (setConnection() && success) ;
            resetConnection())* ; clearState())*"
}
```

Now, an arbitrary number of unsuccessful executions of setConnection is possible when the applet is selected and the connection is in inactive state.

The ConnectionCallOrder specification describes, in which order the methods of the ConnectionManager applet may be called, i.e. it describes how it should be used. As described in Section 2.1, the idea behind call and return specifications is that call specifications describe how a class or object should be used while return specifications describe the behavior of class or object. Hence, the ConnectionCallOrder specification should be implemented as a call specification. This however is not possible due to the fact that it uses the return value of the setConnection which can only be used in return specifications. Thus, the ConnectionCallOrder specification is an example of a specification that describes how a class or object may be used but which can only be implemented as return specification.

All specifications described so far can be violated by a faulty off-card application. If e.g. the off-card application uses ConnectionManager.INACTIVE_AREA as argument when sending a time tick command, the TimeTickValidArgument specification is violated. Thus, the specifications described so far do not only specify legal behaviour of the applet but also legal behaviour of the off-card application.

Specifying the effects of methods

Five different specifications that describe what methods should do were made. Unlike the specifications that describe correct usage of methods, these specifications only specify legal behaviour of the logical channels demo and can not be violated by a faulty off-card application. Two of the method effects specifications were made for the AccountAccessor applet and three for the ConnectionManager applet. All of these specifications were made as return specifications.

The first AccountAccessor specification describes the expected behavior of the debit method. As the name suggests, this method is used to debit the user's account. It receives two arguments which indicate the area the user is currently in and whether or not he uses the smart card's contactless interface. Based on these arguments, the debit method computes the price the user has to pay for one time unit. If the user's balance is high enough to pay the price, the account is debited and the method returns true. Otherwise, nothing is done and the method returns false. Part of this behavior was specified as follows:

```
pltl = {
    "Debit ::=  G(debit() ->
      ((<{#result == true}>
        && <{#pre(#this.balance) >  #this.balance}>)
      || (<{#result == false}>
        && <{#pre(#this.balance) == #this.balance}>)))",
    ...
```

This specification is satisfied iff the debit method either returns true and reduces the user's balance or returns false and does nothing.

It would also be possible to specify exactly under which circumstances the debit method should return false and do nothing. This would however require the LIME specification to be able to access the prices, which is not possible due to the fact that they are stored in private fields.

Immediately after the AccountAccessor applet is initialized, the user's account is empty. The only way to increase the user's balance is to execute the credit method. Thus, it should not be possible to debit the user's account before credit is called the first time. One might try to specify this as follows:

```
    "FirstCreditThenDebit ::= G ((debit() && ! (O
        credit())) -> <{#result == false}>)"
```

This specification would be monitored whenever the debit method or the credit method return. Whenever a specification is monitored, all of its propositions are evaluated. For this specification, this in particular means that the value proposition <{#result == false}> is evaluated every time debit or credit return. This however does not work due to the fact that the credit method returns a short and not a boolean. Thus, the specification shown above results in a compile error.

Currently, the LIME specification language does not provide any mechanisms to circumvent this difficulty. It is however possible to work around this difficulty using overloaded methods like the following ones:

```
public static boolean equalsFalse(boolean b){
    return b == false;
}
public static boolean equalsFalse(short i){
    return false;
}
```

Using these methods, the value proposition <{equalsFalse(#result)}> returns true if the most recent return value is of type boolean and equals false and returns false if the return value is true or of type short. Thus, the specification can be made as follows:

```
    ...
    "FirstCreditThenDebit ::= G ((debit() && ! (O
        credit())) -> <{equalsFalse(#result)}>)"
}
```

If the timeTick method is used correctly, it can have three different outcomes. One possibility is that the network connection currently is not active. In this case, timeTick should do nothing and return the status word ISO7816.SW_NO_ERROR to indicate that no problems occurred. If the network connection is active, the timeTick method should try to debit the user's account. If this fails due to the fact that the balance is too low, timeTick should call resetConnection and return the status word SW_NEGATIVE_BALANCE. Otherwise, ISO7816.SW_NO_ERROR should be returned. This behavior was specified in three return specifications. These three specifications use the success value proposition and three other value propositions:

```
valuePropositions = {
    "success ::= (#result ==
       javacard.framework.ISO7816.SW_NO_ERROR)",
    "negativeBalance ::= (#result ==
       ConnectionManager.SW_NEGATIVE_BALANCE)",
    "balanceDecreased ::=
       (#pre(AccountAccessor.getAccount().balance) >
       AccountAccessor.getAccount().balance)",
    "balanceUnchanged ::=
       (#pre(AccountAccessor.getAccount().balance) ==
       AccountAccessor.getAccount().balance)"
}
```

The `success` proposition is the one that has also been used in the `ConnectionCallOrder` specification. It is true iff the current method's return value indicates that nothing unusual happened during its execution. Similarly, the proposition `negativeBalance` holds, if the return value indicates that debiting the user's account failed. The value propositions `balanceDecreased` holds if the user's account's balance did not change during the execution of the current method and `balanceUnchanged` holds if the user's account's balance decreased.

The first specification for the `timeTick` method describes that the method either decreases or does not change the user's balance and that the returned status word is either `SW_NEGATIVE_BALANCE` or `ISO7816. SW_NO_ERROR`.

```
pltl = {
    ...
    "TimeTick ::= G(timeTick() -> ((balanceDecreased ||
       balanceUnchanged) && (success ||
       negativeBalance)))",
```

Alternatively, "`timeTick() ->`" could have been omitted if the specification was added to the `timeTick` method using an `@Observe` annotation.

The second `timeTick` annotation defines the behavior of the `timeTick` method if it succeeds. In this case, the account should be debited if and only if the network connection is active. Whether or not the network is active can be determined by keeping track of the executions of `setConnection` and `resetConnection`. The network is active iff `setConnection` was successfully called at least once and `resetConnection` was not called since the last time `setConnection` was called successfully, i.e. if `!resetConnection() S (setConnection() && success)` holds. This leads to the following specification:

```
    ...
    "PayIffConnected ::= G((timeTick() && success) ->
       (balanceDecreased <-> (!resetConnection() S
       (setConnection() && success))))",
    ...
```

The last `timeTick` specification describes situations in which the `timeTick` method tries to debit the user's account but the balance is to low, i.e. the situations in which the value proposition `negativeBalance` holds. In these situations, the user's balance should remain unchanged. Also, the `timeTick` method should call `resetConnection`. Last but not least, the

① `timeTick` called

...

② `timeTick` tries to debit account but balance is to low

...

③ `timeTick` calls `resetConnection`

...

④ `resetConnection` returns

...

⑤ `timeTick` returns `ConnectionManager.SW_NEGATIVE_BALANCE`

Figure 5: Execution of the `timeTick` method in case the network connection is active but the user's balance is too low to pay the fee.

`timeTick` method should only try to debit the user's account if the network connection is active. Thus, the network connection should have been active before the `timeTick` method called `resetConnection`. This can be specified as follows:

```
    ...
    "NegativeBalance ::= G((timeTick() &&
        negativeBalance) -> (balanceUnchanged && Y
        (resetConnection() && timeTick()) && Y Y
        (!resetConnection() S (setConnection() &&
        success))))"
}
```

The formula `Y (resetConnection() && timeTick())` specifies that the `timeTick` method should have called `resetConnection` if it returns `ConnectionManager.SW_NEGATIVE_BALANCE`. This can be illustrated using Figure 5, which shows what happens during the execution of the `timeTick` method if the connection is active and the user does not have enough money left to pay the current time unit. After it has been found out that the user's balance is to low in step ②, `timeTick` calls `resetConnection` in step ③. As the specification is a return specification, it is monitored in steps ④ and ⑤. In step ⑤, `(timeTick() && negativeBalance)` holds. At step ④ (i.e. "yesterday" in step 5), both the call proposition `timeTick()` and the call proposition `resetConnection()` are true.

After `timeTick` has called `resetConnection`, the network connection is not active any more. If `timeTick` tried to debit the user's account, the connection should have been active before `resetConnection` was called. This is specified by the subformula `Y Y (!resetConnection() S (setConnection() && success))`.

After adding the specifications, it was verified that violations of the specifications result in exceptions. While some specifications can be violated by simulating a faulty off-card application, others can only be violated by modifying the applets' source code.

3.2 An off-card application for the logical channels demo

The logical channels demo does not contain any off-card application. For testing the demo, a program called apdutool, which is part of the Java Card

Development Kit, can be used. The apdutool allows to send a sequence of APDUs to a Java Card applet. In order to examine the logical channels demo in a more realistic environment however, a small off-card application for the logical channels demo was developed for the case study. The off-card application simulates a part of a larger application that receives commands from other parts of the program. Every command is translated into one or multiple command APDUs for the applet. The off-card application ensures that the smart card is initialized properly and that afterwards commands can be executed in arbitrary order without causing errors in the applet. In particular, APDUs send by the off-card application should not cause any violations of specifications in the applet.

The off-card application supports the following four commands:

Name	arguments	description
`credit`	amount	Increases the user's balance by the specified amount.
`getbalance`	-	Returns the user's current balance.
`tick`	current area and `up` or `down`	Indicates that one time unit has passed. Receives as arguments the current network area and whether the network connection is currently active.
`contactless`	`start` or `stop`	The off-card application starts or stops using the contactless interface of the smart card.

The off-card application was tested using LCT. First, a LCT test program was developed. The test program first initializes the logical channels applet with values received from LCT. Then, it asks LCT for an integer in the range from zero to three. This integer decides which command is sent to the off-card application, e.g. if LCT returns zero, the credit command is sent to the off-card application. Furthermore, the test program asks LCT for the arguments for the selected command. Asking LIME for an command and arguments for that command is repeated N times for a given N.

First, the test program was only compiled but not instrumented and executed several times. In these executions, the test program simply executed N commands with random arguments. Then, the program was instrumented using LCT.

If the test program is instrumented by LCT, it can be executed much more systematically. In the first execution, LCT still returns random values. The instrumented program however keeps track off the path the control flow takes and in particular of how the values returned by LCT influence the control flow. Then, in the second execution, LCT tries to return values that make the control flow follow a yet unexplored path. This is repeated until LCT explored all paths. It may however happen that LCT fails to make the program follow some paths. In this case, these paths are ignored.

In case of logical channels off-card application, LCT tries to execute every possible sequence of N commands. Also, LCT tries the same sequence of commands with different arguments if the arguments make the control flow follow different paths in the off-card application. LCT however can not observe the control flow in the applet. Thus, the test program does not necessarily try all possible control flow paths in the applet.

The LCT test for the off-card application was run for $N = 3$. Running the LCT test for the first time revealed a flaw in the `PayIffConnected` and the `NegativeBalance` specifications. Originally, the subformula for determining whether the connection is active did not differentiate between successful and unsuccessful calls to `setConnection`, i.e. the subformula was `(!resetConnection() S setConnection())` instead of `(!resetConnection() S (setConnection() && success))`. As a result, the `PayIffConnected` specification erroneously assumed that the user also has to pay for every time unit after an unsuccessful attempt to activate the network connection. This error was not found using manual testing but was discovered after the LCT test tried a combination of events that violated the erroneous `PayIffConnected` specification. After the mistake was discovered, the specifications were corrected and LCT was run again. This time, no further errors were found. LCT explored 494 control flow paths and failed to explore 284 control flow paths. In total, running the tests took almost 30 minutes.

4 CONCLUSIONS AND SUGGESTIONS FOR FUTURE FEATURES

Using a suiting simulator and the jdb debugger, it was possible to use the LIME Interface Test Bench to monitor specifications in Java Card applications. The LIME Interface Test Bench proved to be very useful for monitoring complex specifications that would have been very difficult to monitor by hand. The following small additions could even improve the usefulness of the LIME Interface Test Bench:

- As was illustrated using the `FirstCreditThenDebit` specification described in Section 3.1, specifications about return values currently only work if all methods at which the specification is monitored have return types for which the used value specifications are legal expression. This makes it difficult to use return values in specifications about methods of different return types. It is even impossible to use return values in specifications that refer to a method that does not return anything. It would be very helpful to have a way to deal with the issue. For example, it would be much easier to make such specifications if there was a way to ensure that some value specifications are only evaluated for some methods and treated as false for all other methods.

- If specifications become to lengthy, it can become quite difficult to understand their meaning. Therefore, a feature that allows to split up large specifications into smaller chunks would be helpful. The specifications `PayIffConnected` and `NegativeBalance` described in Section 3.1 for instance both use the subformula `!resetConnection() S (setConnection() && success)` in order to determine whether the network is currently active. These specifications would be much easier to read if it was possible to define a shortcut for the subformula and to use that shortcut in the actual specifications.

- There is no way to define at which points the program may be terminate. One might e.g. want to ensure that a file is closed before the program terminates. Currently, this can not be specified using the LIME specification language. A solution might be to introduce an "exit" proposition which is true when the program terminates and to monitor all specifications that use that proposition when the program exists. Then, it could be specified that every file must be closed using e.g. using the regular expression specification `(open(); close())*; exit`.

During the case study, an off-card application for the logical channels demo was developed. The off-card application was tested using LCT. LCT revealed a flaw in one of the specifications and thus proved to be useful. As LCT still is at a very earlier development stage, no suggestions for improving LCT are given in this case study.

Achnowledgements The support from the Academy of Finland through grant 128050 is gratefully acknowledged.

REFERENCES

[1] Java Card Technology website. http://java.sun.com/javacard/, June 2009.

[2] LIME project website. https://poseidon.cs.abo.fi/trac/gaudi/lime, July 2009.

[3] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *TACAS*, pages 174–177, 2009.

[4] Zhiqun Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, 2000.

[5] B. Dutertre and L. De Moura. The YICES SMT solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2006.

[6] Petter Holmström and Sören Höglund. Evaluation of Three Specification-based Testing Approaches. LIME project deliverable, March 2009.

[7] Kari Kähkönen. Automated dynamic test generation for sequential Java programs. Master's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2008.

[8] Kari Kähkönen, Jani Lampinen, Keijo Heljanko, and Ilkka Niemelä. The LIME Interface Specification Language and Runtime Monitoring Tool. In *Proceedings of the 9th International Workshop on Runtime Verification*, Grenoble, 2009. To appear.

[9] Jani Lampinen, Sami Liedes, Kari Kähkönen, Janne Kauttio, and Keijo Heljanko. Incremental Specification of Software Components and Interfaces. LIME project deliverable, 2009.

[10] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV. Tool Paper*, 2006.

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R8	Abhishek Tripathi, Arto Klami, Samuel Kaski Using Dependencies to Pair Samples for Multi-View Learning. October 2008.
TKK-ICS-R9	Elia Liitiäinen, Francesco Corona, Amaury Lendasse A Boundary Corrected Expansion of the Moments of Nearest Neighbor Distributions. October 2008.
TKK-ICS-R10	He Zhang, Markus Koskela, Jorma Laaksonen Report on forms of enriched relevance feedback. November 2008.
TKK-ICS-R11	Ville Viitaniemi, Jorma Laaksonen Evaluation of pointer click relevance feedback in PicSOM. November 2008.
TKK-ICS-R12	Markus Koskela, Jorma Laaksonen Specification of information interfaces in PinView. November 2008.
TKK-ICS-R13	Jorma Laaksonen Definition of enriched relevance feedback in PicSOM. November 2008.
TKK-ICS-R14	Jori Dubrovin Checking Bounded Reachability in Asynchronous Systems by Symbolic Event Tracing. April 2009.
TKK-ICS-R15	Eerika Savia, Kai Puolamäki, Samuel Kaski On Two-Way Grouping by One-Way Topic Models. May 2009.
TKK-ICS-R16	Antti E. J. Hyvärinen Approaches to Grid-Based SAT Solving. June 2009.
TKK-ICS-R17	Tuomas Launiainen Model checking PSL safety properties. August 2009.

ISBN 978-952-248-079-8 (Print)
ISBN 978-952-248-080-4 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)