Extending JML Specifications of JavaCard with Temporal Logic

Kerry Trentelman
Automated Reasoning Group
Australian National University
kerryt@discus.anu.edu.au

Marieke Huisman
INRIA
Sophia Antipolis
Marieke.Huisman@sophia.inria.fr

Overview

- Background
- Java Modeling Language
- Specification patterns
- Proposed specification language
- An example: JavaCard's transaction mechanism
- Further work

Background

- In order to verify its implementation and applets, a full formal specification of JavaCard is needed
- Poll and Meijer have been working towards such a specification using the specification language JML
- A case study: the Application Identifier class has been verified in PVS using the LOOP tool
- Thus far, JML inadequate to specify temporal aspects of JavaCard
- Propose extending JML with temporal logic by using *specification patterns*

Java Modeling Language

- A specification language used to specify Java classes and interfaces
- Java code is annotated with pre and post conditions
- Allows users to specify properties of exceptions

A generic *method* specification:

```
/*@ behaviour
    requires: P;
    ensures: Q;
    signals: e R
*/
```

JML clauses

Class specifications include the clauses:

model Declares *model-variables* which refer to 'states' on which the specification relies, *e.g.* `BoundedThing` has 2 model-variables: `size` and `MAX_SIZE`

invariant States those properties that are preserved by a method, *e.g.* `invariant 0<=size && size<=MAX_SIZE`

constraint States a relation between the pre and post state of a method, *e.g.* `constraint MAX_SIZE == \old(MAX_SIZE)`

Specification patterns

- Specification patterns describe the structure of some aspect of a program's behaviour and provide expressions of this behaviour in common specification formalisms

- They capture an experience base of expert specifiers in temporal logic

- Users don't need an in-depth knowledge of the underlying semantics of the specification language they are using

- Pattern repository at `http://www.cis.ksu.edu/santos/spec-patterns/`

Pattern scopes

Pattern *scopes* give the extent of the program execution over which the pattern must hold

global Throughout the entire program execution

after The execution after a given state/event

before The execution up to a given state/event

between Any part of the execution between 2 designated state/events

after-until The execution after a given state/event until another state/event

Specification patterns

Patterns include:

universal A given state/event occurs throughout a scope

existence A given state/event must occur within a scope

absence A given state/event does not occur within a scope

bounded existence A given state/event must occur n times within a scope

Existence pattern example

intent to describe a part of a program's execution that contains an instance of certain states/events

example specifying termination, e.g. on all executions we eventually reach a terminal state

mapping (LTL) P becomes true:

Globally	$\Diamond(P)$
Before R	$!R \; W \; (P \;\&\; !R)$
After Q	$\Box(!Q) \mid \Diamond(Q \;\&\; \Diamond P))$
Between Q and R	$\Box(Q \;\&\; !R \rightarrow (!R \; W \; (P \;\&\; !R)))$
After Q until R	$\Box(Q \;\&\; !R \rightarrow (!R \; U \; (P \;\&\; !R)))$

Bandera project

- Bandera tool automatically extracts an abstract mathematical model based on specified properties from Java source code
- Tool then renders model into the input languages of several different model checking tools
- Specifications are made using the Bandera Specification Language, or BSL
- BSL provides a macro facility that implements the specification patterns

Proposed specification language

```
<TempForm>   =    <Prop>
                | always <Prop>
                | after <Events> <TempForm>
                | before <Events> <TempForm>
                | between <Events> <Events> <TempForm>
                | at most <nat> <Events>
                | <TempForm> until <Events>
<Events>     =    <Event>+
<Event>      =    <method> called | <method> normal
                | <method> exceptional | <method> terminates
<Prop>       =    <JML-expression> | <method> enabled
                | <method> not enabled | <Prop> & <Prop>
                | <Prop>|<Prop> | !<Prop>
```

Comments

- The **always** <Prop> specification is of the form of an universal pattern with global scope

- The **after** <Events> <TempForm> specification is of the form of an existence pattern with the scope of after

- Patterns give a semantics to the specifications

- We write our semantics in terms of JML invariant and constraint clauses, e.g. $[\![\textbf{always}\ <\text{Prop}>]\!] \stackrel{\text{def}}{=} \texttt{invariant}\ <\text{Prop}>$

An example

- JCSystem includes methods which control applet execution and atomic transaction management
- Data integrity needs to be preserved throughout communication with an external application
- `System.beginTransaction` initiates the beginning of a set of updates
- Data is committed to persistent storage once the applet has called `System.commitTransaction`
- `System.abortTransaction` is called if the applet encounters any internal problems or decides to cancel the transaction

`beginTransaction` specification

- `beginTransaction` throws `TransactionException.IN_PROGRESS` if a transaction is already in progress
- **after** `beginTransaction` called (`beginTransaction` not enabled **until** `abortTransaction` called, `commitTransaction` called)
- **after** `abortTransaction` called, `commitTransaction` called (`beginTransaction` enabled **until** `beginTransaction` called)

`commitTransaction` specification

- `commitTransaction` throws `TransactionException.NOT_IN_PROGRESS` if a transaction is not in progress
- **after** `abortTransaction` called, `commitTransaction` called (`commitTransaction` not enabled **until** `beginTransaction` called)
- **after** `beginTransaction` called (`commitTransaction` enabled **until** `abortTransaction` called, `commitTransaction` called)

JCSystem specification

- **after** `beginTransaction` called
  (`beginTransaction` not enabled & `abortTransaction` enabled & `commitTransaction` enabled
  until `abortTransaction` called, `commitTransaction` called)

- **after** `abortTransaction` called, `commitTransaction` called
  (`beginTransaction` enabled & `abortTransaction` not enabled & `commitTransaction` not enabled
  until `beginTransaction` called)

Further work

- Complete semantics
- Check if language is sufficient
- Verification