2009

Enhancing a behavioral interface specification language with temporal logic features

Faraz Hussain
Iowa State University

Recommended Citation

Hussain, Faraz, "Enhancing a behavioral interface specification language with temporal logic features" (2009). *Graduate Theses and Dissertations.* Paper 10342.

Enhancing a behavioral interface specification language with temporal logic features

by

Faraz Hussain

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Gary T. Leavens, Major Professor
Roger D. Maddux
Samik Basu

Iowa State University

Ames, Iowa

2009

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Listings

ACKNOWLEDGEMENTS

I would like to thank Prof. Gary Leavens for being my guide and mentor throughout my graduate studies at the Iowa State University and now at the University of Central Florida. Thanks also to Prof. Roger Maddux and Prof. Samik Basu for being on my MS Program of Study (POS) committee.

The code snippets included in this thesis report have been inserted using the listings package[1] by Carsten Heinz (maintained by Brooks Moses).

Thanks to Harish B. Narayanappa and Satyadev Nandakumar for help with parsing using ANTLR and during the Abstract Syntax Tree building process. Thanks also to the JML lab members Kristina Boysen, Neeraj Khanolkar, Steve Shaner & Ghaith Haddad for help during different stages of my thesis. Special thanks to Neeraj Khanolkar for helping me with the presentation during my MS oral defense, done remotely from the University of Central Florida.

Finally, I would like to thank my family for their patience, support and constant encouragement.

---

[1] `http://www.ctan.org/tex-archive/macros/latex/contrib/listings/`

ABSTRACT

Specification languages help programmers write correct programs and also aid efforts for dynamically checking a software implementation with respect to its desired specifications. Most mainstream specification languages primarily deal with a program's functional behavior. However, for certain applications it is more natural and intuitive to be able to express a system's temporal properties.

This thesis enhances the capabilities of the Java Modeling Language (JML), a behavioral interface specification language, by incorporating temporal logic constructs. The temporal specification grammar used is a modification of the JML temporal extension proposed by K. Trentelman and M. Huisman in their paper "Extending JML Specifications with Temporal Logic".

I have modified `jmlc`, the runtime assertion checker for the Java Modeling Language, so that it also generates runtime assertion checking code to dynamically check a program's temporal specifications.

CHAPTER 1 Overview

This chapter gives an introduction to current mainstream program specification techniques, motivates the need for introducing constructs for temporal program specification/checking and provides a glimpse into the approach used to add these constructs into a software specification language.

1.1 Introduction

Design-by-contract techniques [24], popularized by Bertrand Meyer by use in the language Eiffel, are widely used in the specification and checking of computer programs. Most current program specification techniques, such as design by contract, are primarily used to describe a system's *functional* behavior. However, for many programming tasks, there is a natural need to provide a temporal description of the system along with its functional behavior. For example, consider the specification:

> "After a file is opened, it is available for reading, until the file is closed."

A program to implement this can be written by the setting and unsetting of flags for the expected "event" (viz the opening of the file). However, its specification can be expressed in a more intuitive manner if *temporal specification* constructs are also available in a specification language. This specification is described later in this chapter and revisited again (§3.3) when discussing the runtime assertion checking of temporal specifications, accompanied with a somewhat realistic example.

In modern programming techniques, a specific task is performed typically by sending a message to an object (i.e. by calling a method). In this thesis, this key practice of method invocation forms the basis of our definition of temporal events. The temporal control points are the calling and termination of method. In addition, we distinguish between the normal termination (i.e. without throwing an exception) of a method and exceptional termination of a method. By temporal specification, we refer to the way program properties are expected to hold or vary delimited by temporal events.

The research presented in this thesis is based on the Java Modeling Language (JML) [23, 8, 5, 6], a behavioral interface specification language developed by Prof. Gary T. Leavens, his colleagues and students, primarily at the Iowa State University. This thesis presents research toward adding temporal specification capabilities to JML.

1.2 Temporal Logic & Specifications

Temporal logic is used extensively in the area of specification and verification of computer programs, especially concurrent programs [10], and is increasingly being used even in non-traditional roles such as sequential program specification, [2]. Temporal logic in computer science has been used traditionally to describe properties of concurrent systems or programs to prove properties related to issues such as deadlock-avoidance. An example is the model checker SPIN[1] [21], which uses Linear Temporal Logic (LTL) to specify the properties that a system needs to respect. Another example is the Bandera Specification Language [13, 25] which is used by the Bandera[2] project [19, 12] as an input language for temporal specifications. The BSL uses temporal specification patterns [15] to express properties that the programmer wishes to express.

Temporal specifications of programs handle descriptions of a sequence of method-related events, rather than the typical functional behavior of a single method, or entire-class invariants, described by traditional program specifications.

1.3 Temporal logic extension to JML

Kerry Trentelman and Marieke Huisman have proposed a temporal logic extension [26] to JML. The research presented in this thesis is an effort to provide an implementation of this temporal logic extension of JML, by adding to the code base of `jmlc`, the JML compiler.

Patterns and Scopes

The work of Trentelman and Huisman is inspired by the SanTos[3] Specification Patterns project [16]. In the Specification Patterns project, a pattern is defined over one of five *scopes*: global, before, after, between, and after-until. Figure 1.1, which is taken from the patterns project website[4], depicts these five temporal specification scopes.

Our implementation of temporal specification constructs is based on a modified semantics of these temporal pattern scopes (Figure 1.2). *Global* scope refers to the entire timeline. The *After R* scope is shown next and refers

---

[1] `http://spinroot.com/`

[2] `http://bandera.projects.cis.ksu.edu/`

[3] `http://santos.cis.ksu.edu/`

[4] `http://patterns.projects.cis.ksu.edu/documentation/patterns/scopes.shtml`

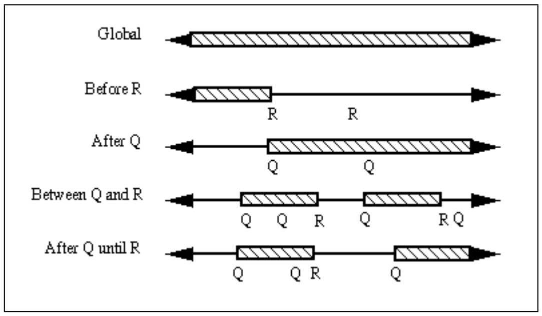

Figure 1.1 Property Specification Scopes

to the the part of the timeline after the first occurrence of event R. Then the *Before R* scope is shown and refers to the part of the timeline before the first occurrence of event R (Figure 1.2).

The scope described by *Between Q and R* is equivalent to the temporal fomula *after Q unless R*; in particular the scope includes the part of the timeline where Q has occurred, but R has not (yet) occurred. The scope described by *After Q until R* describes the part of the timeline between event Q and R, where the event R must occur (Figure 1.2).

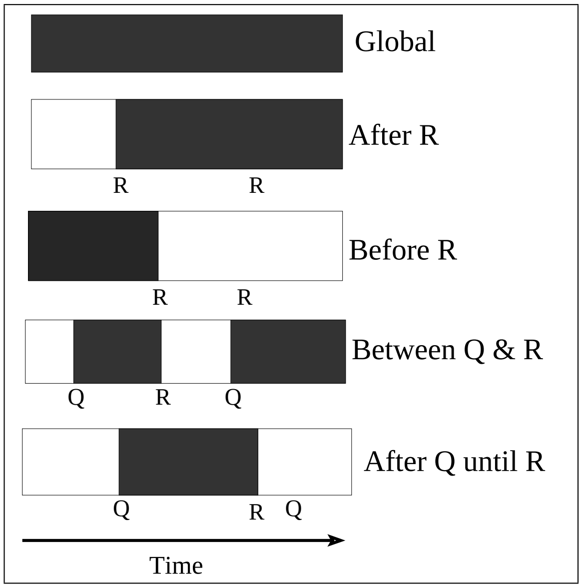

Figure 1.2 Temporal Property Specification Scopes

Trentelman and Huisman's temporal specification constructs also use *occurrence specification patterns*[5] in order to allow the user to describe temporal behavior. These occurrence specification patterns are: Absence (**`never`**), Universality (**`always`**), Existence (**`eventually`**) and Bounded Existence (**`atmost`**).

---

[5] `http://patterns.projects.cis.ksu.edu/documentation/patterns/occurrence.shtml`

Temporal Specification Grammar

```
<TempForm> =   after <Events> <TempForm>
             | before <Events> <TraceProp>
             | <TraceProp> until <Events>
             | <TraceProp> unless <Events>
             | between <Events> <Events> <TraceProp>
             | at most <nat> <Events>
             | <TraceProp>

<TraceProp> =  always <StateProp>
             | eventually <StateProp>
             | never <StateProp>
             | <TraceProp> & <TraceProp>
             | <TraceProp> | <TraceProp>

<Events>    =  <Event>
             | <Event>, <Events>

<Event>     =  <method> called
             | <method> normal
             | <method> exceptional
             | <method> terminates

<StateProp> =  <JMLProp>
             | <method> enabled
             | <method> not enabled
             | <StateProp> & <StateProp>
             | <StateProp> | <StateProp>
             | !<StateProp>
```

Figure 1.3 Proposed Temporal Specification Grammar

Minor changes have been made to the original grammar (Figure 1.3) proposed by Trentelman and Huisman [26]. The actual implementation has been done using the modified temporal specification (Table 2.3).

In the rest of this document, a reference to "the grammar" is to the new grammar (Table 2.3). Our version of the temporal logic extension to JML is henceforth referred to as **temporalJML** and its runtime assertion checker is called **temporaljmlc**.

1.4 The Problem and Approach Used

The problem at hand is to augment the Java Modeling Language with constructs that enable the specification of temporal properties of a program. The basic approach used is to follow the suggestions by Trentelman and Huisman [26] by providing an implementation of their temporal logic extension to JML. This involves all the phases of compiler construction – lexical analysis and parsing of the newly added temporal constructs, building an appropriate abstract syntax tree while parsing a given temporal specification, typechecking to ensure that its a well formed and legal specification according to the defined semantics ([26] §5.1) for temporal specifications, and finally, generating runtime assertion checking code which will perform the actual dynamic checking of these temporal specifications.

The approach used in this research differs from the one used by the JAG tool [18], which translates temporal specifications written in the language extension in [26] into JML annotations (§4.2). The runtime assertion checking of temporal specifications basically builds on the methodology described in Yoonsik Cheon's Ph.D. thesis: "A Runtime Assertion Checker for the Java Modeling Language" [7].

Another goal of the thesis is to clarify the semantics of the newly added temporal specification constructs (§4.1), especially those that are not fully described, or are somewhat ambiguous in [26].

The JML compiler, `jmlc`, is built on top of the Multijava compiler, `mjc` [11]. This thesis work builds upon the `jmlc` compiler, enhancing it with temporal specification capabilities by adding a modified version of the temporal specification constructs suggested in [26]. The implementation is based on the JML2 compiler codebase.[6] To achieve this, the main steps that had to be followed were:

- Add the suggested temporal logic constructs outlined in the grammar proposed by Trentelman and Huisman [26] to the file specifying the current JML grammar in the JML2 system located in `/JML2/org/jmlspec/checker/Jml.g`. The grammar proposed in [26] is shown in in Figure 1.3 and its modification used in this implementation is shown in Table 2.3.
- Create classes that represent the different nodes of the abstract syntax tree for specifications written in this new temporal-specification-augmented grammar.
- Modify the runtime assertion checker so that it also generates code to verify the temporal specifications written by the programmer/user.

1.5 Temporal Specification Examples

Trentelman and Huisman show an example ([26, §4.1]) in which they use their temporal logic extension of JML to describe properties of the JavaCard$^{\text{TM}}$ transaction mechanism. Lets quickly look at how sample temporal specifications can be written in our temporal logic extension of JML, **temporalJML**.

---

[6] The source code can be accessed from `http://jmlspecs.cvs.sourceforge.net/viewvc/jmlspecs/JML2/` under the tag *farazhussain_temporalspecs* or directly from: `http://jmlspecs.cvs.sourceforge.net/viewvc/jmlspecs/JML2/?pathrev=farazhussain_temporalspecs`.

1.5.1 File Operations Example

A simple example class providing basic file operations is shown in Listing B.1. The temporal specifications in the file are reproduced here (Listing 1.1). The intended informal semantics of the three temporal specifications as as follows.

1. The method `writeFile` is *not enabled* (i.e. if it terminates, it terminates by throwing an exception) unless the method `openFile` has been invoked.
2. After method `openFile` has been called and it terminates normally (i.e. without throwing an exception) the method `writeFile` is always *enabled* (i.e. if it terminates, it doesn't throw an exception) *until* the method `closeFile` is invoked; also `closeFile` *must* eventually be invoked.
3. After method `closeFile` has been called and it terminates normally (i.e. without throwing an exception) the method `writeFile` is always *not enabled* (i.e. it it terminates, it does so by throwing an exception) *unless* the method `openFile` is invoked.

Listing 1.1 File Operations temporal specifications

```
//@ public static temporal (\always(\not_enabled(writeFile)) \unless \call(openFile));
//@ public static temporal (\after \normal(openFile); (\always (\enabled (writeFile)) \until
    \call (closeFile)));
//@ public static temporal (\after \normal(closeFile); (\always (\not_enabled(writeFile))
    \unless \call(openFile)));
```

This example is revisited when explaining runtime assertion checking (§3.3) in **temporaljmlc**.

1.5.2 Bank Account example

For another example, consider the file in Listing C.1, whose sole temporal specification is reproduced here (Listing 1.2). The *temporal events* in the temporal specification are: the normal termination of the method `openAC`, the normal termination of the method `activateAC` and the call to the the method `suspendAC`.

Listing 1.2 Bank Account Example Temporal Specification

```
//@ public temporal (\after \normal (openAC);
        (\after \normal (activateAC);
        (\always (balance>0) | \eventually (swissType)) \unless \call (suspendAC)) );
```

An informal description of the semantics of this temporal specification is as follows: After the invocation of `openAC` has terminated successfully, followed by a successful (normal) termination of `activateAC`, a property must hold unless the method `suspendAC` is invoked. The property, say P, that must hold is that either $balance$ must always be positive or $swissType$ must at some point get the value $true$. To reiterate, the property P has to hold after `openAC` (has been called and) has terminated normally, and then `activateAC` (has been called and) has terminated normally, until `suspendAC` has been invoked. In case `suspendAC` is invoked the property P no longer has to hold unless `activateAC` (is called again and) terminates normally.

This example is revisited when explaining runtime assertion checking (§3.2) in **temporaljmlc**.

1.6 About this document

Chapter 2 includes a discussion of the grammar, parsing & abstract syntax tree construction, and typing rules and typechecking of temporal specifications.

Chapter 3 describes how runtime assertion checking code for the dynamic checking of temporal specifications is generated by **temporaljmlc**.

Chapter 4 contains notes on the semantics of **temporalJML** and clarifications of certain **temporalJML** constructs. It also discusses scope for future work on this topic and gives pointers to related research in this area.

Chapter 5 contains the conclusion and a discussion on the limitations of the current implementation and the scope of future work in the area.

CHAPTER 2 Parsing, Abstract Syntax Tree Construction & Typechecking

This chapter discusses the grammar of the temporal extension to JML proposed by Trentelman and Huisman, issues related to parsing (§2.2) and construction of the abstract syntax tree (§2.3, §2.4) for temporal specification and finally the typing rules and typechecking (§2.5, §2.6) of temporal formulas.

2.1 Temporal JML grammar and Temporal constructs

The temporal specification grammar extension to JML proposed by Trentelman and Huisman (Figure 1.3) has been modified (Table 2.3) in order to achieve better integration with the existing JML grammar inside the `jmlc` compiler. In particular, we have added a backshlash ("\") before temporal specification keywords to help in the lexical analysis of temporal constructs.

The temporal constructs added to the JML grammar are briefly described below:

Temporal Formula This is the top-level temporal specification grammar rule. A temporal formula can be an: **`after`** formula, **`before`** formula, **`between`** formula, **`atmost`** formula, **`unless`** formula, **`until`** formula or a temporal trace property.

Trace Property A temporal trace property can be an **`always`**, **`eventually`** or **`never`** trace property. We further informally distinguish between a *basic temporal trace property* (one which contains no temporal conjunction (&) or temporal disjunction (|) operator) and a *general temporal trace property* (one which may contain one or more &s and/or |s).

Events The temporal events rule is used to represent a list of temporal events. It evaluates to true if any one of the temporal events comprising this list occurs.

Event A temporal event is caused by a step in program execution which relates to the invocation or completion of a method. It maybe one of the following: a method call (**`call`**) , a normal termination of a method (**`normal`**), an exceptional termination of a method (**`exceptional`**). The temporal event **`terminates`** is said to have occurred if either the event **`normal`** occurs or the event **`exceptional`** occurs.

State Property A temporal state property is either a simple JML property or involves one of the newly added operators **`enabled`** and **`not_enabled`**. We further informally distinguish between a *basic temporal state property* (one which contains no temporal conjunction (&) or temporal disjunction (|) operator connecting two basic state properties both of which contain a **`enabled`** or **`not_enabled`**) and a *general temporal state property* (one which may contain one or more &s and/or |s connecting two basic state properties both of which contain an **`enabled`** or **`not_enabled`**).

2.2 Parsing

The JML compiler, `jmlc`, uses the ANTLR[1] tool for lexical analysis and parsing. The relevant grammar files are `/JML2/org/jmlspecs/checker/Jml.g` and `/MJ/org/multijava/mjc/Mjc.g`. The keyword **`temporal`** is introduced to allow temporal property specification.

Listing 2.1 Start of the jmlDeclaration rule

```
jmlDeclaration [CParseClassContext context,long mods, Token startToken]
{
    TokenReference sourceRef = utility.buildTokenReference( startToken );
    JmlInvariant inv = null;
    JmlPredicate pred = null;
    boolean redundant = false;
    JmlMethodNameList mnList = null;
    JmlStoreRefExpression storeRef = null;
    JmlStoreRef[] storeRefList = null;
    JmlSpecExpression specExpr = null;
    JmlSpecExpression[] specs = null;
          JNameExpression fieldName = null;

          JmlTemporalFormula jtf = null; //--FH
    JExpression jte = null; //--FH
}
    :
```

A temporal specification is similar to a JML **`invariant`** or **`constraint`** specification. A JML invariant node is created in the rule called `jmlDeclaration` (Listing 2.1), using a subrule which is basically of the form "invariant predicate" (Listing 2.2). Invariants and Constraints in `jmlDeclaration` are handled using an OR (|)

---

[1] `http://www.antlr.org`

(Listing 2.2). In a similar way, top level temporal specification parsing functionality has been added (Listing 2.3) in the rule for `jmlDeclaration` using the temporal disjunction (|) operator (not shown in Listing 2.3).

Listing 2.2 Handling invariants and constraints in jmlDeclarationrule

```
:
    (
        ( "invariant" | "invariant_redundantly" { redundant = true; } )
        pred = jmlPredicate[]
        {
            context.addInvariant(
                new JmlInvariant( sourceRef, mods, redundant, pred ));
        }
    |
        ( "constraint" | "constraint_redundantly" { redundant = true; } )
        pred = jmlPredicate[]
        ( "for"
            (
                "\\everything"
            |
                mnList = jmlMethodNameList[]
            )
        )?
        {
            context.addConstraint(
        new JmlConstraint( sourceRef, mods, redundant, pred,
                           mnList ));
        }
```

It can be seen (Listing 2.3) that the top level rule for handling of temporal specifications is really `jmlTemporalExpression` (Listing A.3). This rule depicts the handling of a `TemporalUnlessExpression` and a `TemporalUntilExpression`.

Listing 2.3 Handling temporal specifications in jmlDeclarationrule

```
    ( "temporal" | "temporal_redundantly" { redundant = true;} )
    {
    }
    jte = jmlTemporalExpression[]
    {
        context.addTemporalFormula(
```

```
                new JmlTemporalFormula(sourceRef, mods, redundant, jte));
        }
```

To understand how the other temporal specification constructs are handled, it is necessary to start with the `jmlImpliesExpression` rule in `Jml.g` and keep following the rules in the files `Jml.g` and `Mjc.g`. The order in which the relevant rules containing handlers for temporal specifications are reached is shown in Table 2.1.

Table 2.1 Trace from `JmlImpliesExpression`

jmlImpliesExpression (Jml.g)
jLogicalOrExpression (exists only in Mjc.g)
jLogicalAndExpression
jInclusiveOrExpression
jExclusiveOrExpression
jAndExpression
jEqualityExpression (overridden, so considering the one in Jml.g)
jRelationalExpression (Jml.g)
jShiftExpression (Mjc.g)
jAdditiveExpression (Mjc.g)
jMultiplicativeExpression (Mjc.g)
jUnaryExpression (Mjc.g)
jUnaryExpressionNotPlusMinus (Mjc.g)
jPrimaryExpression (Mjc.g) (overridden, so considering the one in Jml.g)
jPrimaryExpression (Jml.g)

In Jml.g, a `jPrimaryExpression` can be a `mjPrimaryExpression` or a `jmlPrimary`. Now, proceed from a `mjPrimaryExpression` (Table 2.2). It is now clear that from the rule for `jmlImpliesExpression`, either of the rules for `jmlPrimary` and `jmlSpecQuantifiedExprRest` can be reached directly.

Table 2.2 Trace from `mjPrimaryExpression`

mjPrimaryExpression (Mjc.g)
jParenthesizedExpression (overridden, so considering the one in Jml.g)
jParenthesizedExpressionRest (Jml.g)
jmlSpecQuantifiedExprRest (Jml.g)

`jmlSpecQuantifiedExprRest` rule handles the **`after`**, **`atmost`**, **`before`** and **`between`** temporal-expressions. The rule `jmlPrimary` (Listing A.1) handles the temporal trace properties (viz **`always`**, **`eventually`**, **`never`**) and the newly introduced JML temporal state properties (viz **`enabled`** and **`not_enabled`**) as defined in the modified temporal JML grammar (Table 2.3).

Table 2.3 Modified Temporal Specification Grammar

⟨TempForm⟩	::=	(**\after** ⟨Events⟩ ; ⟨TempForm⟩)
		| (**\before** ⟨Events⟩ ; ⟨TraceProp⟩)
		| ⟨TraceProp⟩ **\until** ⟨Events⟩
		| ⟨TraceProp⟩ **\unless** ⟨Events⟩
		| (**\between** ⟨Events⟩ ; ⟨Events⟩ ⟨TraceProp⟩)
		| (**\atmost** ⟨Nat⟩ ⟨Events⟩)
		| ⟨TraceProp⟩
⟨TraceProp⟩	::=	**\always** ⟨StateProp⟩
		| **\eventually** ⟨StateProp⟩
		| **\never** ⟨StateProp⟩
		| ⟨TraceProp⟩ & ⟨TraceProp⟩
		| ⟨TraceProp⟩ | ⟨TraceProp⟩
⟨Events⟩	::=	⟨Event⟩ | ⟨Event⟩, ⟨Events⟩
⟨Event⟩	::=	**\call** (⟨method⟩)
		| **\normal** (⟨method⟩)
		| **\exceptional** (⟨method⟩)
		| **\terminates** (⟨method⟩)
⟨StateProp⟩	::=	⟨JMLProperty⟩
		| **\enabled** (⟨method⟩)
		| **\not_enabled** (⟨method⟩)
		| ⟨StateProp⟩ & ⟨StateProp⟩
		| ⟨StateProp⟩ | ⟨StateProp⟩
		| !⟨StateProp⟩

The rule for `jmlSpecQuantifiedExprRest` (Listing A.2) shows the handling of the temporal specification constructs **\after**, **\before**, **\atmost**, and **\between**. Of these, **\between** involves two temporal-event lists, whereas the others involve one temporal-event list. In each case, the appropriate abstract syntax tree node is created.

In the case of **\after** temporal expressions, the temporal-event list is followed by the grammar element `jmlTemporalExpression`. This means that the expression which is part of a **\after** formula can in turn be another temporal formula such as a **\after** formula, a **\before** formula, etc.

2.3 Description of AST classes

The classes used for representing nodes of the abstract syntax tree created while parsing temporal specifications are in directory `/JML2/org/jmlspecs/checker/`. A list of these files with a brief description of their functions follows.

JmlTemporalAfterExpression As per the grammar in Table 2.3, an **\after** formula contains one temporal event list and an underlying temporal expression. This underlying expression can be another temporal formula, for example, an **\after** temporal formula, a **\between** formula, an **\unless** temporal formula, or simply a temporal trace property (i.e. **`always`**, **`eventually`** or a **`never`** expression or a combination of these basic-trace-properties using the operators & and |).

JmlTemporalBeforeExpression As per the grammar in Table 2.3, a **`before`** formula contains one temporal event list and an underlying temporal trace property (i.e. **`always`**, **`eventually`** or a **`never`** expression or a combination of these basic-trace-properties using the operators & and |), unlike in the case of **`after`** temporal formulas, where it can be an arbitrary temporal formula.

JmlTemporalAtMostExpression As per the grammar in Table 2.3, an **`atmost`** formula contains one temporal-event-list (typically containing only one temporal event), and an integer literal which denotes the maximum number occurrences of each of the events that's allowed. (The typechecking phase will ensure that this integer literal is non-negative).

JmlTemporalBetweenExpression As per the grammar in Table 2.3, a **`between`** formula contains two temporal-event-lists and an underlying temporal-trace-property.

JmlTemporalUntilExpression As per the grammar in Table 2.3, **`until`** formula contains one temporal-event-list and an underlying temporal-trace-property.

JmlTemporalUnlessExpression As per the grammar in Table 2.3,**`unless`** temporal-expressions contain one temporal-event-list and an underlying temporal-trace-property.

JmlTemporalTraceProperty As per the grammar in Table 2.3, a temporal trace property contains one of the basic-trace-property-expressions (viz **`always`**, **`eventually`** or a **`never`** expression).

Note: A conjunction/disjunction of the above mentioned basic trace properties is formed using the class `JmlBitwiseExpression`.

JmlTemporalEvent A `JmlTemporalEvent` holds the type of the event (viz one of **`call`**, **`normal`**, **`exceptional`** or **`terminates`**) and an object of type `JmlMethodName` describing the method which this temporal-event refers to. A `JmlTemporalEvent` also contains a data member called `JmlTemporalEvent next` which points to the next event in a temporal event list (Events in Table 2.3), if any, and is null otherwise.

JmlTemporalStateProperty According to the grammar in Table 2.3, a temporal state property can be a simple JML-property or contain one newly added **`enabled`** or **`not_enabled`** operators which may or may not be associated using a temporal conjunction/disjunction with simple JML properties. For a note on the current restricted implementation of this, see §5.1.

JmlSpecQuantifiedAugmentedExpression This class has been created to serve as a common superclass of the class `JmlSpecQuantifiedExpression` and the newly added abstract class `JmlTemporalSequenceExpression`.

JmlTemporalFormula This is the class used to represent an entire temporal specification. An AST node of this type is created by the rule `jmlDeclaration`.

JmlTemporalExpression This is an abstract class created as a superclass of `JmlTemporalUntilExpression` and `JmlTemporalUnlessExpression` to avoid redundant code.

JmlTemporalSequenceExpression This is an abstract class created as a superclass of `JmlTemporalAfterExpression`, `JmlTemporalBeforeExpression`, `JmlTemporalBetweenExpression` and `JmlTemporalAtMostExpression` to avoid redundant code.

2.4 Testing the abstract syntax tree

The file `/JML2/org/jmlspecs/checker/TestJmlTemporalParser.java` contains `junit` test cases to test the construction of the abstract syntax tree after temporal specifications have been parsed.

To run these AST tests, in the JML2 base directory, which I write as `/JML2/`, run the following command:

```
/JML2$:  junit org.jmlspecs.checker.TestJmlTemporalParser
```

To test that the existing JML tests are still working, run the following command:

```
/JML2$:  junit org.jmlspecs.checker.TestJmlParser
```

2.5 Typechecking

The types added to the existing JML types are shown in Table 2.4. These types are used to typecheck the temporal specifications annotated with any JML code and also during runtime assertion checking. Files representing these type are in the `/JML2/org/jmlspecs/checker/` directory.

The enforcement of typing rules of a temporal specification AST is done by the methods `typecheck` and `getType`. The type system of temporal specifications is also used during runtime assertion checking phase when building the temporal state machine.

Typechecking of any AST node essentially follows the grammar for temporal specifications (Table 2.3). The following is a brief description of typechecking in temporal AST classes and includes the temporal type assigned to each of the temporal AST nodes, which can be retrieved using their respective `getType` methods.

Table 2.4 JML Temporal Types

`JmlStdType.temporalFormula`	Representing the top-level JML temporal formulas
`JmlStdType.temporalTraceProperty`	Representing both basic and general temporal trace properties
`JmlStdType.temporalStateProperty`	Representing temporal state properties, but excluding simple JML properties
`JmlStdType.temporalAtMostExpression`	Representing temporal **`atmost`** formulas
`JmlStdType.temporalAfterExpression`	Representing temporal **`after`** formulas
`JmlStdType.temporalBeforeExpression`	Representing temporal **`before`** formulas
`JmlStdType.temporalBetweenExpression`	Representing temporal **`between`** formulas
`JmlStdType.temporalUnlessExpression`	Representing temporal **`unless`** formulas
`JmlStdType.temporalUntilExpression`	Representing temporal **`until`** formulas

`JmlTemporalFormula` A `JmlTemporalFormula` node represents an entire temporal formula specification. Its `typecheck` method calls `typecheck` on the underlying temporal expression. After typechecking the underlying temporal expression, it uses the `getType` method of the underlying temporal expression to check if this expression has one of the types allowed by the grammar (Table 2.3); if not, an error is reported. The allowed types for the underlying temporal specification are: `JmlStdType.temporalTraceProperty`, `JmlStdType.temporalAfterExpression`, `JmlStdType.temporalAtMostExpression`, `JmlStdType.temporalBeforeExpression`, `JmlStdType.temporalBetweenExpression`, `JmlStdType.temporalUnlessExpression` and `JmlStdType.temporalUntilExpression`.

In turn, the type of a `JmlTemporalFormula` node itself is `JmlStdType.temporalFormula` (Table 2.4), which is the value returned by its `getType` method.

`JmlTemporalAfterExpression` As per the grammar in Table 2.3, an **`after`** formula contains a temporal event and an underlying temporal formula. A `JmlTemporalAfterExpression` is a subclass of `JmlTemporalSequenceExpression`. Its `typecheck` method first calls `super.typecheck()` which typechecks the temporal event. It then calls `typecheck` on the underlying temporal expression, which according to the grammar (Table 2.3) is a temporal formula. After typechecking the underlying temporal expression, it calls `getType` on the underlying temporal expression to check if its of one of the types allowed by the grammar (Table 2.3); if not an error is reported. The allowed types for the underlying temporal expression are: `JmlStdType.temporalTraceProperty`, `JmlStdType.temporalAfterExpression`, `JmlStdType.temporalAtMostExpression`, `JmlStdType.temporalBeforeExpression`, `JmlStdType.temporalBetweenExpression`, `JmlStdType.temporalUnlessExpression` and `JmlStdType.temporalUntilExpression`.

The type of a `JmlTemporalAfterExpression` node itself is `JmlStdType.temporalAfterExpression` (Table 2.4), which is the value returned by its `getType` method.

`JmlTemporalAtMostExpression` As per the grammar in Table 2.3, an **`atmost`** formula contains a temporal event. A `JmlTemporalAtMostExpression` is a subclass of `JmlTemporalSequenceExpression`. Its `typecheck` method calls `super.typecheck` which typechecks the temporal event. It also tests if the integer data field representing the maximum number of events is non-negative.

The type of a `JmlTemporalAtMostExpression` node itself is `JmlStdType.temporalAtMostExpression` (Table 2.4), which is the value returned by its `getType` method.

`JmlTemporalBeforeExpression` As per the grammar in Table 2.3, a **`before`** formula contains a temporal event and an underlying temporal trace property. Its `typecheck` method calls `super.typecheck` which typechecks the temporal event. It then calls `typecheck` on the underlying temporal expression which is a temporal trace property. After typechecking the underlying temporal expression it calls `getType` on the underlying temporal expression to check if is of one of the types allowed by the grammar (Table 2.3); if not an error is reported. The allowed type of the the underlying temporal expression is `JmlStdType.temporalTraceProperty`.

The type of a `JmlTemporalBeforeExpression` node itself is `JmlStdType.temporalBeforeExpression` (Table 2.4), which is the value returned by its `getType` method.

`JmlTemporalBetweenExpression` As per the grammar in Table 2.3, a **`between`** formula contains two temporal events and an underlying temporal expression. A `JmlTemporalBetweenExpression` is a subclass of `JmlTemporalSequenceExpression`. Its `typecheck` method first calls `super.typecheck()` which typechecks one of the temporal events. It then calls `typecheck` on the second temporal event; and then on the underlying temporal expression, which according to the grammar (Table 2.3) is a temporal trace property. After typechecking the underlying temporal expression, it calls `getType` on the underlying temporal expression to check if its of one of the types allowed by the grammar (Table 2.3); if not an error is reported. The allowed type for the underlying temporal expression is `JmlStdType.temporalTraceProperty`.

The type of a `JmlTemporalBetweenExpression` node itself is `JmlStdType.temporalBetweenExpression` (Table 2.4), which is the value returned by its `getType` method.

`JmlTemporalEvent` According to the temporal grammar (Table 2.3) the nonterminal *Events* refers to a (possible singleton) Event list. The type `JmlTemporalEvent` is used to represent this grammar production. A `JmlTemporalEvent` contains an integer that identifies the type of the event (**`call`**, **`normal`**, **`exceptional`** and **`terminates`**), a descriptor for the method which this event is associated with and a pointer to the next temporal event in this list, if any. Its typecheck method confirms if the event-type identifier has a valid value, typechecks the method descriptor and then calls `typecheck` on the temporal event node that this node points to, it the pointer is non-null.

The type of a `JmlTemporalEvent` node itself is `CStdType.Boolean` which is the value returned by its `getType` method.

`JmlTemporalExpression` This is the superclass of `JmlTemporalUntilExpression` and `JmlTemporalUnlessExpression`. Its `typecheck` method typechecks the underlying temporal expression. It then calls `getType` on the underlying temporal expression to check if it is of one of the types allowed by the grammar (Table 2.3); if not, an error is reported.

`JmlTemporalSequenceExpression` This is the superclass of `JmlTemporalAfterExpression`, `JmlTemporalBeforeExpression`, `JmlTemporalBetweenExpression` and `JmlTemporalAtMostExpression`. Its `typecheck` method typechecks the underlying temporal event for each of its subclasses (the first temporal event, in the case of **`between`** formulas).

`JmlTemporalTraceProperty` As per the grammar in Table 2.3, a temporal trace property contains one of the basic temporal trace properties (**`always`**, **`eventually`** or **`never`**) or a combination of basic trace properties made using the temporal conjunction (&) and/or temporal disjunction (|) operators. The class `JmlTemporalTraceProperty` represents a basic temporal trace property. Non-basic, general temporal trace properties which use the temporal conjunction/disjunction operators are formed using `JmlBitwiseExpression` (and, in case of parenthesization, `JParenthesedExpression`). A basic temporal trace property contains an underlying state property and an identifier for the type of trace property (**`always`**, **`eventually`** or **`never`**). Its `typecheck` method typechecks the underlying temporal expression, which is a temporal state property. It then calls `getType` on the underlying expression to check if it is of one of the types allowed by the temporal specification grammar (Table 2.3); if not, an error is reported. The allowed types for the underlying temporal expression are `CStdType.Boolean` (representing simple JML properties) and `JmlStdType.temporalStateProperty` (representing the newly added temporal state property operators **`enabled`** and **`not_enabled`**). The type of a

`JmlTemporalTraceProperty` node itself is `JmlStdType.temporalTraceProperty` (Table 2.4) which is the type returned by its `getType` method.

`JmlTemporalStateProperty` A temporal state property can be either a simple JML property or may contain one of the new temporal state property operators (viz **`\enabled`** and **`\not_enabled`**). Note that currently a restricted version of **`\enabled`** and **`\not_enabled`** operators has been implemented (§5.1).

`JmlTemporalUnlessExpression` As per the grammar in Table 2.3, an **`\unless`** formula contains a temporal event list and an underlying temporal trace property. A `JmlTemporalUnlessExpression` is a subclass of `JmlTemporalExpression`. Its `typecheck` calls `super.typecheck` which typechecks the underlying temporal expression and also typechecks the associated temporal event list.

The type of a `JmlTemporalUnlessExpression` node itself is `JmlStdType.temporalUnlessExpression` (Table 2.4), which is the value returned by its `getType` method.

`JmlTemporalUntilExpression` As per the grammar in Table 2.3, an **`\unless`** formula contains a temporal event list and an underlying temporal trace property. A `JmlTemporalUntilExpression` is a subclass of `JmlTemporalExpression`. Its `typecheck` calls `super.typecheck` which typechecks the underlying temporal expression and also typechecks the associated temporal event list.

The type of a `JmlTemporalUntilExpression` node itself is `JmlStdType.temporalUntilExpression` (Table 2.4), which is the value returned by its `getType` method.

2.6 Typechecking Tests and Errors

The file `/JML2/org/jmlspecs/checker/JmlMessages.msg` contains the list of errors that the typechecker reports when it encounters typing problems in temporal specification.

The directory `/JML2/org/jmlspecs/checker/testcase/typecheck/` contains files to test the typechecking of temporal specifications. Some of the tests cases are such that there is a corresponding .java-expected file which contains the error message that will be generated by JML2. Others have no such corresponding .java-expected file and are those test cases that are compiled successfully by the JML2 compiler without any error message. To run these typechecking tests, run the command following command:

```
/JML2/org/jmlspecs/checker/testcase/typecheck$:  make runtests
```

2.6.1 A typechecking test failure

One typecheck test fails on running the `make runtests` command in directory `/JML2/org/jmlspecs/checker/testcase/typecheck/`. The test is in the file `Primitive_bigint_basic.java`. The test has been isolated and can be observed in the file `/JML2/org/jmlspecs/temporalspec/TemporalTestBigintProblem.java`.

The source of the error is the specification **`/*@spec_bigint_math@*/`** which qualifies the class `Primitive_bigint_basic`. It occurs because I have added the method `JmlExpressionFactory.createBitwiseExpression` (Listing 2.5). This method overrides `JExpressionFactory.createBitwiseExpression` (Listing 2.4). It is this overriding which has caused the problem, not the changes to `JmlBitwiseExpression.java`.

Listing 2.4 JExpressionFactory.createBitwiseExpression()

```
public /*@non_null@*/ JBitwiseExpression createBitwiseExpression(/*@non_null@*/
    TokenReference where,
            int oper,
            /*@non_null@*/ JExpression left,
            /*@non_null@*/ JExpression right){
            return new JBitwiseExpression(where, oper, left, right);
}
```

The only difference between `JmlExpressionFactory.createBitwiseExpression` and `JFactory.createBitwiseExpression` is that the former creates a `JmlBitwiseExpression` whereas the latter creates a `JBitwiseExpression`.

Listing 2.5 JmlExpressionFactory.createBitwiseExpression()

```
public /*@non_null@*/ JBitwiseExpression createBitwiseExpression(/*@ non_null @*/ Token
    op,
                   /*@ non_null @*/ TokenReference where,
                   int oper,
                   /*@ non_null @*/ JExpression left,
                   /*@ non_null @*/ JExpression right) {
    return new JmlBitwiseExpression(where, oper, left, right);
//              return (op instanceof CToken)
//                              ?       new JmlBitwiseExpression(where, oper, left, right)
//                              :       new JBitwiseExpression(where, oper, left, right);
}
```

An attempt can be made to remove this problem if it can be ascertained what in the lexer causes a certain token (for example '+' or '&') to be of type `CToken`.

CHAPTER 3 Code Generation & Runtime Assertion Checking

This chapter uses the examples mentioned briefly in the introductory chapter (§1.5) to explain general ideas about the code generated by **temporaljmlc** and runtime assertion checking ideas for temporal specification in general.

The runtime assertion checking code in the JML2 project is located inside `/JML2/org/jmlspecs/jmlrac/`. This is the directory in which files have been added and existing files modified to runtime assertion checking capabilities for temporal specifications.

The basic approach I use to generate runtime assertion checking code for temporal specifications is to create one finite state machine, with accepting and non-accepting states, per temporal specification, when temporal specifications are parsed. After parsing, code is produced to generate a finite state machine at runtime when the JML-augmented Java code is compiled using **temporaljmlc**. The transitions of these finite state machines are the temporal events (which are specified using temporal event constructs, viz **`call`**, **`normal`**, **`exceptional`** and **`terminates`**.) For every temporal specification, there is a variable representing each of its basic trace properties (if it doesn't contain one of the temporal state properties viz **`enabled`** and **`not_enabled`**). The temporal state machine causes these variables to be updated as appropriate (only if the machine is in a trace property checking state). When the program terminates, the values of the variables representing the trace property of each temporal specification are checked to decide if a trace-property violation error is to be reported. Also, each machine's final state is checked to see if its an accepting state; if not, an error is reported. This is used as the checking mechanism for constructs like an **`until`** temporal formula and **`enabled`** and **`not_enabled`** state properties.

3.1 Major Code Generation Ideas

In `/JML2/org/jmlspecs/jmlrac/`, the method `TransType.translate` has been modified to insert code for the translation of temporal specifications (Listing 3.1).

The various methods called by `translate` perform (temporal runtime assertion checking) code generation work like adding the appropriate data members and methods to the target type.

For example, the `addTemporalSpecificationObserverUpdateMethod` method uses class

`TemporalUpdateMethodProducer`, a visitor subclass of `RacAbstractVisitor` to add the method `update$temporalspec` (the equivalent of a visitor pattern `update` [17]) to the target class. Similarly, the visitor `TemporalTracePropertyIdentifier` is used to add trace property checking/error reporting methods to the target type.

Listing 3.1 TransType.translate()

```
public void translate()
{

    // translate type decl if it is not a model
    if (hasFlag(typeDecl.modifiers(), ACC_MODEL))
        return;

    String wrapperClass = null;
    if (genSpecTestFile) wrapperClass = translateForSpecTestFile();

    markTemporalSpecificationInstanceStaticFormulaExistence(); //FH
    addTemporalSpecificationListOfInstancesToType(); //FH
    addTemporalSpecificationEventLists();//FH
addTemporalSpecificationRuntimeTemporalMachineVariables(); //FH


    //FH: Adds temporal trace property end-scope check methods
    TemporalTracePropertyIdentifier ttpi = new TemporalTracePropertyIdentifier(
                    typeDecl.temporalFormulas(), this);
    ttpi.perform();


    ArrayList methods = new ArrayList(typeDecl.methods());
    ArrayList inners = typeDecl.inners();
    JPhylum[] fieldsAndInits = typeDecl.fieldsAndInits();

    // translate invariants and (history) constraints.
    translateInvariant();
    translateConstraint();

    //Translate temporal formulas
addTemporalSpecificationObserverUpdateMethod(); //FH
createTemporalSpecificationRuntimeMachineInitMethodForStaticFormulas(); //FH
```

```
    createTemporalSpecificationRuntimeMachineInitMethodForInstanceFormulas(); //FH
        translateTemporalFormulaUsingStateMachine(); //FH
        addTemporalStaticMachineFinalStateCheckMethod(); //FH (24DEC08)
        addTemporalInstanceMachineFinalStateCheckMethod(); //FH (30JAN09)
        addTemporalInstanceMachineFinalStateCheckMethodCaller(); //FH (30JAN09)



        // translate represents clauses.
        // WARNING! The translation of represents clauses must precede
        // that of body which also performs the translation of
        // model fields if any.
        // The reason is that if this type declaration contains both
        // a model field declaration and its represents clause,
        // the model field access method should be generated from the
        // represents clause, not from the model field declaration
        // (see translateRepresents and translateField).
        translateRepresents(typeDecl.representsDecls());

        // translate body (i.e., inner classes, field, and methods)
        translateBody(inners, methods, fieldsAndInits);


        if (genSpecTestFile) {
            postTranslationChangesForSpecTestFile(wrapperClass);
        }

        // do subclass (class or interface) specific finalization,
        // e.g., generating specification inheritance mechanism,
        // surrogate class, etc.
        finalizeTranslation();

    }
```

The methods `createTemporalSpecificationRuntimeMachineInitMethodForInstanceFormulas` and `createTemporalSpecificationRuntimeMachineInitMethodForStaticFormulas` are used to generate runtime temporal machine initialization code. They both use the visitor `TemporalStateMachineBuilder` to build one temporal state machine per specification. Class `TemporalStateMachineBuilder`, along with `TemporalStateMachineGenerator` do the actual machine code generation using the classes `TemporalState` and `JMLRuntimeTemporalStateMachine`.

A detailed understanding of the classes used in code generation requires a lengthy perusal of the code, which can be obtained from:
`http://jmlspecs.cvs.sourceforge.net/viewvc/jmlspecs/JML2/?pathrev=farazhussain_temporalspecs`. In the rest of this chapter using the examples from the introductory chapter (§1.5) will be used to explain code generation.

3.2 Generated Code for the Bank Account Example

Consider again the bank account temporal specification example (§1.5.2) from the introductory chapter. The specifications are reproduced in Listing 3.2.

Listing 3.2 Bank Account Temporal Specification Reproduced

```
//@ public temporal (\after \normal (openAC);
        (\after \normal (activateAC);
        (\always (balance>0) | \eventually (swissType)) \unless \call (suspendAC)) );
```

The non-static temporal events which occur are stored in the ArrayList variable `temporalEventList$instance`. The wrapper for each method is used to add elements to this ArrayList. Consider the method `getBalance`; in the generated file, it is renamed `internal$getBalance` and a wrapper method is generated with the name `getBalance` following the wrapper method approach used by Cheon [8]. A **`try`**-block contains contains the call to the original method, now renamed as `internal$getBalance`.

Before the **`try`**-block which contains the call to the `internal$getBalance` method, the call to `checkTemporalFormulas$instances` performs temporal specification checking in the *pre-state* of the execution of method `internal$getBalance`.

Before `internal$getBalance` is called, the event `getBalanceLParenRParenI$temporalspec$called` is deemed to have occurred and is added to `temporalEventList$instance`. After `internal$getBalance` returns, the event `getBalanceLParenRParenI$temporalspec$normal` is deemed to have occurred and is added to `temporalEventList$instance`, if `internal$getBalance` *does not throw an exception.* If `internal$getBalance` throws a non-runtime assertion checking exception then the event `getBalanceLParenRParenI$temporalspec$exceptional` is deemed to have occurred and is added to `temporalEventList$instance`. (A non-runtime assertion checking exception is one which is not a subtype of `JmlAssertionError`.)

The **`finally`** block contains code showing temporal specification checking by invoking the method `checkTemporalFormulas$instances` in the *post-state* of the execution of method `internal$getBalance`.

3.2.1 The Temporal State Machine

A temporal state machine is created for each temporal specification. In the code generated by the runtime assertion checker, these temporal machines are represented by the type `JMLRuntimeTemporalStateMachine` (Listing A.5). For `TemporalSpecBankAC` class temporal specification, the runtime temporal state machine is initialized in method `init$instance$temporalspecs$RuntimeTemporalMachines`. The method starts with the following line:

Listing 3.3 Runtime Temporal State Machine initialization

```
tsm$temporalspec$TF0 = new JMLRuntimeTemporalStateMachine(this, 0 );
```

The current object is passed to the constructor of the runtime temporal state machine. This is done so as to provide a rudimentary implementation of the *Observer Pattern* [17], which is required for appropriate calls to trace-property-update methods. Essentially, the object `this`, which is of type `TemporalSpecBankAC`, becomes an *observer* of the runtime temporal state machine represented by `tsm$temporalspec$TF0`.

The file `TemporalSpecBankAC.java` has only one temporal specification (Listing 3.2) and the corresponding machine is represented by the variable `tsm$temporalspec$TF0`. The temporal state machine is defined in the method `init$instance$temporalspecs$RuntimeTemporalMachines` The method creates four temporal States (numbered 0, 1, 2, 3), defines a start state (State 0), and adds relevant transitions. Figure 3.1 gives a graphical representation of the automaton generated.

The start state (`State0`) is shown by an incoming arrow. The accepting states[1] (`State0, State1, State2, State3`) are marked by a double frame box. The (only) temporal trace property checking state (`State2`) is colored blue and also marked by an asterisk (*). The long arrows with the arrowheads touching some state represent transitions from the state touching the arrow tail to the state touching the arrow head. (Note that in Figure 3.1, the names of the temporal events causing the transitions have been shortened for convenience.)

3.2.2 Temporal States

The class `TemporalState` represents a temporal state. A `TemporalState` contains the following data members:

- A state number, which uniquely identified this state: `state`
- A flag indicating if its an accepting state: `acceptingState`
- A flag indicating if its a trace-property-checking-state: `tracePropertyCheckingState`

---

[1] If, at program termination, the temporal state machine is not in one the accepting states, an exception is generated.

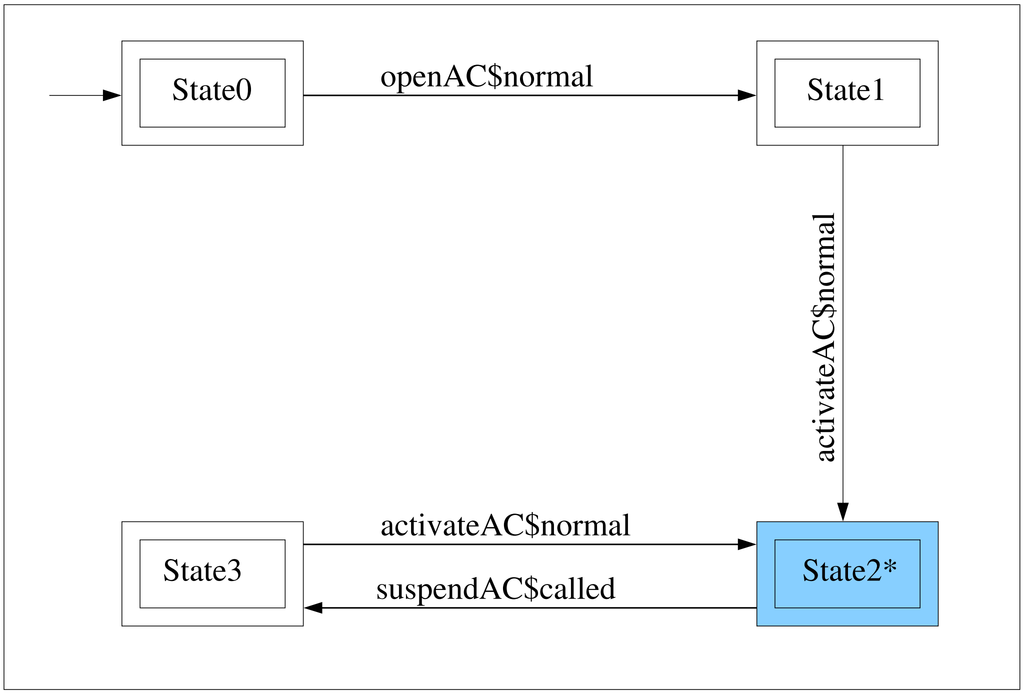

Figure 3.1 BankAC Temporal State Machine Automaton

- A string which, for a non-accepting state, holds a value describing the reason why its a non-accepting state.
- Four other data members required for the implementation of \\`enabled` and \\`not_enabled` formulas (not used in this example).

The method `init$instance$temporalspecs$RuntimeTemporalMachines` in the generated file `testTemporalSpecsExample.java.gen` contains code where temporal states are constructed.

`State0` is an accepting, non-temporal-traceproperty checking state.

`State1` is an accepting, non-temporal-traceproperty-checking

`State2` is an accepting, temporal-traceproperty-checking state.[2]

`State3` is an accepting, non-temporal-traceproperty-checking state.

3.2.3 Checking Temporal Specifications

The method `checkTemporalFormulas$instance` checks the temporal formula in Listing 3.2. It uses the `temporalEventList$instance` to see if there any new events have occurred. If there are, then it considers

---

[2] If the specification were changed by replacing \\`unless` by \\`until` then this state would become a non-accepting state and the reason for non-acceptance would be represented by the following string value: “Temporal Formula TF0 contains a TemporalUntilExpression: Expecting one of the following temporal events: [terminateACLParenRParenV$temporalspec$called]"

them in their order of occurrence and feeds each to the runtime machine's `consume` method, and then removes that event from `temporalEventList$instance`. This is the mechanism with which the machines keeps itself up-to-date. After making all the necessary transitions depending on the temporal events that have occurred, the machine's `performTemporalChecks` method is called.

Listing 3.4 Runtime machine's temporal checks

```
public void performTemporalChecks() {
        if (this.currentState.isTracePropertyCheckingState()) {
                //this.setChanged();
                //this.notifyObservers();
                this.myObserver.update$temporalspec(this, null);
        }
}
```

The method `performTemporalChecks` (Listing 3.4) tests if the machine is currently in a temporal-trace-property-checking state. If it is, then the machine's *observer's* `update$temporalspec` method is called.

3.2.4 Checking trace properties

A given temporal specification formula can have only one trace property. However, this trace property can be a conjunction/disjunction of multiple *basic trace properties* (i.e. **`always`**, **`eventually`**, **`never`**). Each basic temporal trace property needs to be checked when the temporal state machine is in the trace-property-checking state.

For a given temporal specification formula, there is a variable associated with each *basic trace property*. Here, the variables are `tpID$TF0$TP1$Ty145` and `tpID$TF0$TP2$Ty146`.

The observer's update method, `update$temporalspec`, in turn calls the appropriate trace property update methods for each basic trace property for the temporal specification whose representative machine invoked this oberver update method.

The trace property in the temporal specification in Listing 3.2 is a disjunction of two the two basic trace properties **`always`**`(`*balance* > `0)` and **`eventually`**`(swissType)`. The observer's `temporalspec$update` method calls update method for updating the status of both basic trace properties `updateAlwaysTP$instance$tpID$TF0$TP1$Ty145` and `updateEventuallyTP$instance$tpID$TF0$TP2$Ty146` are called respectively).

The method `updateAlwaysTP$instance$tpID$TF0$TP1$Ty145` checks the value of `balance>0` and appropriately updates the corresponding basic trace property variable `tpID$TF0$TP1$Ty145`. The method `updateEventuallyTP$instance$tpID$TF0$TP2$Ty146` checks the value of `swissType` and appropriately updates the corresponding trace property variable `tpID$TF0$TP2$Ty146`.

Consider again the always-trace-property update method, `updateAlwaysTP$instance$tpID$TF0$TP1$Ty145`. It works by checking the value of and appropriately setting the value of the variable `tpID$TF0$TP1$Ty145`. While checking an **`always`** temporal trace property, if the underlying temporal-state-property does not hold, then that trace property is false.

Accordingly, in this case the always-temporal-trace property variable `tpID$TF0$TP1$Ty145` is *permanently* set to false. On the other hand, if the underlying temporal-state-property does hold, then the temporal trace property variable `tpID$TF0$TP1$Ty145` is set to true.

The method `updateEventuallyTP$instance$tpID$TF0$TP2$Ty146` checks the value of `swissType` and appropriately updates the corresponding basic trace property variable `tpID$TF0$TP2$Ty146`. Initially, this basic eventually trace property's scope has not started and hence the inital value of the variable `tpID$TF0$TP2$Ty146` is true. On the first call to the eventually-trace-property update method, `updateEventuallyTP$instance$tpID$TF0$TP2$Ty146`, the value of that variable is set to false because the underlying state property is not known ever have been true up to this point since it is yet to be checked. Indeed, then the value of the underlying state property, `swissType` is checked. If it is true, then the trace-property-variable `tpID$TF0$TP2$Ty146` is *permanently* set to true. If not, the value of `tpID$TF0$TP2$Ty146` remains false.

3.2.5 Temporal State Machine post-final state checking

In the generated file `TemporalSpecBankAC.java.gen`, the original `main` method has been renamed `internal$main` and a new wrapper method has been created with the name `main`. This method is similar to the generated wrapper method `getX` in the addition of temporal-events (`mainLParenArrLjavaSlashlangSlashStringRParenV$temporalspec$called`, `mainLParenArrLjavaSlashlangSlashStringRParenV$temporalspec$normal`, `mainLParenArrLjavaSlashlangSlashStringRParenV$temporalspec$exceptional`) to the static temporal event list `temporalEventList$static` and the checking of temporal specifications in the pre-state and post-state by calling `checkTemporalFormula$TF0$instances`). Note that since this is a static method, the static and not instance temporal event list is modified. Also, the temporal formula checking method which is called is `checkTemporalFormulas$instances` (Listing 3.5), which itself is static.

Listing 3.5 Check all (instance) temporal formulas

```
public static void checkTemporalFormulas$instances(java.lang.String rac$msg) {
for (int i = 0; i < listOfInstances$temporalspec.size(); i++){
TemporalSpecBankAC anObject = (TemporalSpecBankAC) listOfInstances$temporalspec.get(i);
anObject.checkTemporalFormulas$instance(rac$msg);

}
}
```

`checkTemporalFormulas$instances` in turn calls `checkTemporalFormulas$instance` (Listing 3.6) on each of the `TemporalSpecBankAC` objects in existence (which are inserted by the constructor wrapper into variable `listOfInstances$temporalspec`).

Listing 3.6 Check (instance) temporal formulas

```
public void checkTemporalFormulas$instance(java.lang.String rac$msg) {
String anEvent = "";
while (temporalEventList$instance.size() > 0) {
 anEvent = (String) temporalEventList$instance.get(0);
tsm$temporalspec$TF0.consume(anEvent);
tsm$temporalspec$TF0.performTemporalChecks();
temporalEventList$instance.remove(0);
}
}
```

However, the wrapper method `main` differs from the wrappers for all other methods in what happens after the post-state call to `checkTemporalFormulas$instances`.

At this point it is clear that the original `main` has now completed execution, either with or without throwing an exception. Depending on this, the latest addition to `temporalEventList$static` is either `mainLParenArrLjavaSlashlangSlashStringRParenV$temporalspec$normal` or `mainLParenArrLjavaSlashlangSlashStringRParenV$temporalspec$exceptional`.

Method `main` now calls `checkTraceProperties$instances` to check the final status of the temporal trace properties for this temporal specification.

Recall that each call to `checkTemporalFormulas$instance` was used to appropriately update the variables representing the basic trace properties, by the mechanism of the runtime temporal state machine (which calls the observer's `update$temporalspec` method if its in a trace-property-updating state; `update$temporalspec` in turn calls the trace property update methods, `updateAlwaysTP$instance$tpID$TF0$TP1$Ty145()` and `updateEventuallyTP$instance$tpID$TF0$TP2$Ty146()`. These trace property update method keep the value of the basic trace property variables, `tpID$TF0$TP1$Ty145` and `tpID$TF0$TP2$Ty146`, up-to-date).

The method `checkTraceProperties$instance` is reproduced in Listing 3.7. Note from the original temporal specification (Listing 3.2) that the trace property is actually a disjunction of the basic traceproperties `\always` ($balance > 0$) and and `\eventually` (swissType). For this reason, the boolean values of the variables representing these two basic temporal trace properties are combined with a disjunction (|). If either of the variables is true (i.e. either of the basic trace properties is true), then the temporal specification is respected and no exception is thrown. Otherwise, the temporal specification did not hold, and a `TemporalSpecificationException` is thrown.

Listing 3.7 Final trace property check

```
public void checkTraceProperties$instance(java.lang.String rac$msg) {
if (!((tpID$TF0$TP1$Ty145) || (tpID$TF0$TP2$Ty146))) {
      throw new JMLTemporalSpecificationError( "Temporal Trace Property at location <File
          \"TemporalSpecBankAC.java\", line 7, character 25> violated",
          "TemporalSpecBankAC", " checkTraceProperties$instance" , new
          java.util.HashSet());
}
}
```

Furthermore, `main` calls the method `checkTemporalMachineFinalState$instances`, which in turn calls `checkTemporalMachineFinalState$instance` (Listing 3.8) for each object that is created (and hence is in `listOfInstances$temporalspec`).

Listing 3.8 Temporal Machine Final State check

```
public void checkTemporalMachineFinalState$instance(java.lang.String rac$msg) {

//--------Code for machine for instance temporal formula 0-----------
if (!tsm$temporalspec$TF0.inFinalState()) {
String reasonForNonAcceptingState =
    tsm$temporalspec$TF0.getReasonForCurrentNonAcceptingState();
throw new JMLTemporalSpecificationError( "Temporal Specification at location <File
    \"TemporalSpecBankAC.java\", line 7, character 25> violated: " +
    reasonForNonAcceptingState, "TemporalSpecBankAC", "
    checkTemporalMachineFinalState$instance" , new java.util.HashSet());
}
//--------End of code for machine for instance temporal formula 0-----------
}
```

The method `checkTemporalMachineFinalState$instance` checks to ensure if the runtime temporal state machine's final state is an accepting state. If not, then it prints why final state is not an accepting state. In this example, all states are accepting states, so the temporal state machine final state check will always succeed and this (indirect) call to `checkTemporalMachineFinalState$instance` will not report any errors regarding a non-accepting state.

Recall that when a `TemporalState` object is created, a flag inside it is set which indicates if its an accepting state or not; and if its a non-accepting state, a String data member inside the object hold the "reason" for it being a non-accepting state.

Minor modification and an (even more) contrived example

To explain the utility of the temporal machine, final state checking, consider the following case. If we modify part of the specification, replacing the `\unless` by an `\until`, to arrive at a more contrived example, the behavior differs as explained next. In this case, the (indirect) call to `checkTemporalMachineFinalState$instance` will ensure that the semantics of the `\until` formula is respected. The semantics of the expression "`someTraceProperty \until (\call aTemporalEvent)`" is that the trace property should hold until the temporal event `\call aTemporalEvent` occurs *and* that the temporal event `\call aTemporalEvent` *must* occur.

3.2.6 Sample runs of the BankAccount class

Consider the main driver in Listing 3.9 for the bank account class (Listing C.1).

Listing 3.9 Bank account main driver –1

```
 public static void main(String[] args)  {
            TemporalSpecBankAC ac1 = new TemporalSpecBankAC();
            ac1.openAC();
            ac1.setBalance(-100);
            ac1.setBalance(200);
            ac1.activateAC();
            ac1.setBalance(-300); //positive or negative
            //ac1.setSwissType(true);
            ac1.suspendAC();
 }
```

The output produced by **temporaljmlc** is shown in Figure 3.2. It shows that the trace property for the temporal specification was violated. As can be seen from the main driver (Listing 3.9), this occurred since the balance was negative even after the account was activated and the bank was never declared to be a *swissType* account.

```
$ jmlrac2 TemporalSpecBankAC
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLTemporalSpecificationError:
Temporal Trace Property at location <File "TemporalSpecBankAC.java", line 7, character 24>
violated:
        at TemporalSpecBankAC.checkTraceProperties$instance(TemporalSpecBankAC.java:116)
        at TemporalSpecBankAC.checkTraceProperties$instances(TemporalSpecBankAC.java:126)
        at TemporalSpecBankAC.main(TemporalSpecBankAC.java:1443)
```

Figure 3.2 Bank Account Driver-1 Output

Now consider the main driver in Listing 3.10 for the bank account class (Listing C.1).

Listing 3.10 Bank account main driver –2

```
 public static void main(String[] args)  {
            TemporalSpecBankAC ac1 = new TemporalSpecBankAC();
            ac1.openAC();
            ac1.setBalance(-100);
            ac1.setBalance(200);
            ac1.activateAC();
            ac1.setBalance(-300); //positive or negative
            ac1.setSwissType(true);
            ac1.suspendAC();
 }
```

No output (in particular, no temporal specification violation exception) is produced by **temporaljmlc** as shown in Figure 3.3. This behavior is expected because the trace property is not violated anymore since the flag `swissType` is set to true by the driver (Listing 3.10).

```
faraz@hussain-machine:~/Software/JML2/org/jmlspecs/temporalspec/temporalfiles$jmlrac2 TemporalSpecBankAC
faraz@hussain-machine:~/Software/JML2/org/jmlspecs/temporalspec/temporalfiles$
```

Figure 3.3 Bank Account Driver-2 Output

3.3 Revisiting the File Operations Example

The file operations example's code is listed in Listing B.1, Listing B.2, Listing B.3 and Listing B.4. Now lets run some sample tests using these to see how the temporal specification runtime assertion checker behaves.

Which version of the class `TemporalSpecFileOps` from the above listings is being used will be specified along with an explanation of the **temporaljmlc** output.

Consider the `main` driver as in Listing 3.11.

Listing 3.11 File Operations main driver –1

```
public static void main(String[] args) {
    String filename = "file1.txt";

    try {
    File f = new File(filename);
    openFile(f);
    writeFile(f);
    closeFile(f);
    writeFile(f);
    } catch (Exception e) {
        System.out.println(e);
    }

}
```

On running **temporaljmlc** on the `main` in Listing 3.11, using the class `TemporalSpecFileOps` version in Listing B.1, the output produced is shown in Figure 3.4. This output is to be expected because `writeFile` throws an exception if its not preceded by `openFile`. In fact, the third temporal specification(TS2) requires that this property hold.

```
$ jmlrac2 TemporalSpecFileOps
java.lang.Exception: Cannot write to file.
```

Figure 3.4 File Operations Driver-1 Output

On running **temporaljmlc** on the `main` in Listing 3.12, using the class `TemporalSpecFileOps` version in Listing B.2, the output produced is shown in Figure 3.5. Note that in the `main` for *Driver2*, `writeFile` is invoked without first calling `openFile`. This should lead to an exception. However, note that (Listing B.2, line 33) the program now doesn't respect the intended specification because the exception throwing has been excluded.

Listing 3.12 File Operations main driver –2

```
public static void main(String[] args) {
    String filename = "file1.txt";
```

```

        try {
        File f = new File(filename);
        //openFile(f);
        writeFile(f);
        closeFile(f);
        //writeFile(f);
        } catch (Exception e) {
            System.out.println(e);
        }

    }
```

The output produced (Figure 3.5) shows that **temporaljmlc** gave a temporal specification violation exception. It also points to the the temporal specification which was violated (TS0). In addition, its diagnostic message also indicates which *bad event* occurred – `writeFileLParenLjavaSlashioSlashFileRParenV$temporalspec$normal` (i.e. the *normal* termination of method `writeFile`). According to the temporal specification, if `writeFile` is called without `openFile` having been called before, an exception should be thrown by the program (\\`not_enabled` `writeFile`). However, since this exception was not thrown (a *bad event*) due to the comment on line 33, a temporal specification exception was thrown (Figure 3.5).

```
$ jmlrac2 TemporalSpecFileOps
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLTemporalSpecificationError:
        Temporal Specification at location <File "TemporalSpecFileOps.java", line 9, character 31>
        violated:
        Bad Events: [TemporalSpecFileOps.writeFile( java.io.File )
        terminates without throwing an exception]:
        at TemporalSpecFileOps.checkTemporalFormulas$static(TemporalSpecFileOps.java:239)
        at TemporalSpecFileOps.writeFile(TemporalSpecFileOps.java:746)
        at TemporalSpecFileOps.internal$main(TemporalSpecFileOps.java:48)
        at TemporalSpecFileOps.main(TemporalSpecFileOps.java:1038)
```

Figure 3.5 File Operations Driver-2 Output

Now consider the third driver in this file operations example (Listing 3.13), which uses the class `TemporalSpecFileOps` version in Listing B.3. Its clear that the `main` driver here respects the intended temporal specifications. However, note that (Listing B.3) `openFile` now does not set `openFlag` as it should. This causes `writeFile` to throw an exception even though the file has been opened.

Listing 3.13 File Operations main driver –3

```
    public static void main(String[] args) {
        String filename = "file1.txt";
```

```java

        try {
        File f = new File(filename);
        openFile(f);
        writeFile(f);
        closeFile(f);
        //writeFile(f);
        } catch (Exception e) {
            System.out.println(e);
        }

    }
```

The **temporaljmlc** output on running this is shown in Figure 3.6. **temporaljmlc** complains that a temporal specification (TS1) was violated because `writeFile` terminated by throwing an exception even though TS1 says that after a successful opening of the file (`\normal (openFile)`), `writeFile` must not terminate with an exception (`\enabled (writeFile)`) as long as `closeFile` is not invoked.

```
$ jmlrac2 TemporalSpecFileOps
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLTemporalSpecificationError:
        Temporal Specification at location <File "TemporalSpecFileOps.java", line 12, character 32>
        violated:
        Bad Events: [TemporalSpecFileOps.writeFile( java.io.File ) terminates by throwing an exception]:
        at TemporalSpecFileOps.checkTemporalFormulas$static(TemporalSpecFileOps.java:245)
        at TemporalSpecFileOps.writeFile(TemporalSpecFileOps.java:747)
        at TemporalSpecFileOps.internal$main(TemporalSpecFileOps.java:49)
        at TemporalSpecFileOps.main(TemporalSpecFileOps.java:1039)
```

Figure 3.6 File Operations Driver-3 Output

Now consider the fourth driver in this file operations example (Listing 3.14), which uses the class `TemporalSpecFileOps` version in Listing B.4. Its clear that the `main` driver here does not respect the intended specifications because `writeFile` is invoked even after `closeFile` has terminated and there is no intervening invocation of `openFile`. According to the program specifications, this should throw an exception.

Listing 3.14 File Operations main driver –4

```java
    public static void main(String[] args) {
        String filename = "file1.txt";

        try {
        File f = new File(filename);
        openFile(f);
        writeFile(f);
```

```
        closeFile(f);
        writeFile(f);
        } catch (Exception e) {
            System.out.println(e);
        }

    }
```

The **temporaljmlc** output on running this is shown in Figure 3.7. According to the message **temporaljmlc** complains that the third temporal specification (TS2) was violated, because a *bad event*, viz `writeFileLParenLjavaSlashioSlashFileRParenV$temporalspec$normal` (i.e. the normal termination of `writeFile`) occured. Clearly, an exception was expected, but because of the erroneous implementation of `closeFile` (where the unsetting of `openFlag` was excluded), the exception was not thrown. Thus, a temporal specification (TS2) was violated causing **temporaljmlc** to throw an exception.

```
$ jmlrac2 TemporalSpecFileOps
Exception in thread "main" org.jmlspecs.jmlrac.runtime.JMLTemporalSpecificationError:
        Temporal Specification at location <File "TemporalSpecFileOps.java", line 15, character 31>
        violated:
        Bad Events: [TemporalSpecFileOps.writeFile( java.io.File )
        terminates without throwing an exception]:
        at TemporalSpecFileOps.checkTemporalFormulas$static(TemporalSpecFileOps.java:251)
        at TemporalSpecFileOps.writeFile(TemporalSpecFileOps.java:748)
        at TemporalSpecFileOps.internal$main(TemporalSpecFileOps.java:51)
        at TemporalSpecFileOps.main(TemporalSpecFileOps.java:1040)
```

Figure 3.7 File Operations Driver-4 Output

The above examples have shown, using modifications to the `main` driver and by introducing errors in the implementation of the methods of the class (like `openFile` and `closeFile`), how the temporal runtime assertion checker **temporaljmlc** reports implementation problems dynamically if they do not respect the temporal specification.

CHAPTER 4 Discussion

This chapter contains a general discussion of certain issues regarding the semantics of the temporal logic extension [26] and also related notes on the semantics of the temporal runtime assertion checker, **temporaljmlc**, that I have implemented (§4.1). Later in the chapter, I also discuss related work in the area and how my work differs from existing efforts on temporal specification (§4.2).

4.1 Notes on Semantics

Trentelman and Huisman give a state-based semantics for their temporal logic extension of JML ([26] §5.1). Here are some general notes about the subrules of a temporal formula.

- A `\before` formula specification is equivalent to an `\always`-`\until` specification. Note that a `\before` formula is fundamentally different from a `\after` formula in that it cannot contain another top-level temporal formula, but only a temporal trace property.
- An `\until` formula is a realization of the temporal logic *strong until* operator and is used to specify that one of the the following temporal events *must* occur.
- An `\unless` formula is a realization of the temporal logic *weak until* operator and is used to specify that the all of the following temporal events may never occur, in which case the `\unless` formula holds if the underlying trace property holds.

Below are a couple of clarifications regarding the semantics and the behavior of **temporaljmlc** , my implementation of the temporal logic extension proposed in [26].

Attempted specification on an internal state

Consider the following specification:

(`\after` `\call`(method1); (`\before` `\normal`(method1); `\always`(P)));

This seemingly innocuous temporal specification hides a subtle semantics issue. Between the two temporal events described in this specification, there is no state in which temporal formula specifications can be checked, since they are checked using wrapper methods, just before a `\call` event and just after a `\normal` or `\exceptional` event. Therefore, this specification essentially is an attempt to describe the program state in an internal state, which cannot be done because the runtime assertion checking is done only at the method control points (i.e. the invocation and termination of methods). In this case, the only temporal formula check happens in the wrapper method right when `method1` is called, so the success of failure of this temporal formula depends on the whether property P holds right at the point of the invocation of `method1`. **temporaljmlc** has the correct semantics in this case by performing the temporal formula check only at that point.

The semantics of `\atmost` formulas

According to the grammar in [26] (Figure 1.3), the `\atmost` formulas describe the number of time an event can happen using a natural number. It is to be noted that the natural numbers include zero, therefore an `\atmost` formula can be used to prohibit the occurrence of a temporal event (or a list of such events).

4.2 Related Work

The research work here is primarily an effort to provide an implementation for the temporal logic extension to JML proposed by Kerry Trentelman and Marieke Huisman ([26]). They also propose translating a subset (viz the formulas which express *safety properties*) of the new constructs of this temporal extension back into standard JML expressions[26, §5.2]. On the other hand, Groslambert et al [4] propose a method for the verification of *liveness properties* in this temporal extension of JML. I have implemented their JML temporal extension, with some modifications, on top of the `jmlc` runtime assertion checker using the JML2 compiler codebase.

JAG [18] is a **J**ML **A**nnotation **G**enerator that translates formulas expressed in the extension described in [26] into JML annotations. This differs from my approach (which is based on the temporal logic extension proposed in [26]) because I translate the Java code annotated with temporal (and normal JML) specifications into Java, whereas JAG translates temporal formula specifications back into JML.

Cheon and Perumandla propose an extension to the Java Modeling Language that allows the specification of sequences of method calls (*protocols*) [9]. Their basic approach is to use regular-expression like syntax (a *call sequence clause*) to define the permitted sequences of method calls. Ying Jin has suggested the use of context free grammars (CFG) to represent the possible method call sequences of a Java program, thus allowing static verification of properties by by inserting protocol checking the CFG implementation [22]. The approach suggested by the above authors helps primarily in specifying *protocol properties* of Java types. This differs in essence from my work because their approach provides (and demands) separation of temporal properties (*protocols*) from functional behavior whereas my approach (which is based on the temporal logic extension to

JML proposed by Trentelman and Huisman [26]) allows integration of the two using Bandera-style patterns to describe temporal behavior and trace properties to specify functional behavior. In Aspect Oriented Programming, specifying history constraints with tracematches ([1]) also uses regular expression type techniques to build state automata, like Cheon and Perumendla's method call sequences ([9]).

Temporal Rover [14] is a verification tool that allows specifications written in an extension of Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) to be annotated to code written in C, C++, Java, Verilog and VHDL. In this sense, their approach seems similar to the one used by Trentelman and Huisman [26] (on which my research is based), but Temporal Rover is proprietary software and not available for free. This tool, developed by Time-Rover Software[1] generates code from the written specifications which is linked to the application that its part of. Morover, Temporal Rover does not integrate with JML.

Jass (**J**ava with **ass**ertions) [3] is an extension to Java which allows specifications to be annotated with Java code. Jass translates this annotated Java code into pure Java and checks compliance with the specifications dynamically. Jass supports specification of *trace assertions*, the ordering of method calls using design ideas from CSP [20], unlike Trentelman and Huisman [26] whose approach is based on Bandera type specification patterns [15, 16] and which I also have adopted for the research presented here.

The Bandera Specification Language is a "source-level, model-checker independent language for expressing temporal properties of Java program actions and data [13]. It attempts to aid temporal specification by avoiding the overly formal traditional ways of expressing such specifications like Computational Tree Logic (CTL) and Linear Temporal Logic (LTL). It is different from our approach (and from any of the others described in this section), in that its based on model checking, whereas we follow primarily a design by contract approach [24].

---

[1] `http://www.time-rover.com/`

CHAPTER 5 Conclusion

This chapter summarizes my contribution to the field of specification and verification of programs, outlines the limitations of my implementation and discusses scope for future work in the area (§5.1).

Our contribution is the addition of temporal specification capability using Bandera-style patterns to the Java Modeling Language, **temporalJML**, and an implementation of **temporalJML** by integrating it with the JML toolset. This augmented JML tool (built on top of the JML runtime assertion checker, `jmlc`) is called **temporaljmlc**.

Unlike traditional program specification constructs **temporalJML** allows specifications using multiple program control points in a single specification. Also, our implementation differs from certain other attempts at the temporal specification of programs, like method call sequences (§4.2), because **temporalJML** allows the integration of temporal and functional specifications.

temporalJML is based on the temporal logic extension of JML sugggested by Trentelman and Huisman ([26]). Another contribution of thesis is the clarification of certain issues related to the semantics (§4.1) of the temporal logic extension in ([26]).

5.1 Limitations & Future Work

One key obstacle faced during this implementation process was the lack of good documentation for the `jmlc`, the JML runtime assertion checker. In order to aid the implementation of further extensions to JML, it would be helpful to put effort into creating a `javadocs`-style API for `jmlc`.

A typechecking test failure caused due to the the temporal logic specification's use of JML's bitwise operators (& and |) is explained earlier (§2.6.1).

The software currently does not handle temporal specifications written in Interfaces. Also, there is right now no support for inheritance of temporal specifications. Future work may involve introducing constructs to allow protocol specification (§4.2).

For now the newly added temporal state properties, (viz `\enabled` and `\not_enabled`), by default assume that the state property is part of an `\always` trace property and the mixing of the temporal state operators

`enabled` and **`not_enabled`** is currently disallowed. Also, the negation operator (!) for temporal state properties has not been implemented yet.

I also plan to make the technique for checking simple Temporal State Properties (i.e. those involving normal JML properties), similar to the technique currently used for implementing the newly added temporal state properties **`enabled`** and **`not_enabled`**. Therefore, the simple temporal state properties will also be implemented using variables *encoded inside the JMLRuntimeTemporalMachine* instead of having them as data members in the translated target class.

Finally, there should also be an attempt for any future coding effort to minimize changes to the existing files in the `jmlc` runtime assertion checker.

APPENDIX A   Code References

Listing A.1   jmlPrimary Rule

```
jmlPrimary []
returns [JExpression self = null]
{
    TokenReference sourceRef = utility.buildTokenReference( LT(1) );
    JmlSpecExpression specExpression;
    JExpression expression;
    JmlStoreRef[] storeRefList;
    JmlMethodNameList names = null;
    JmlSpecExpression[] specExpressionList;
    CType type = null;
    JmlStoreRefExpression storeRefExpression = null;



    int aen = -1; //for always eventually never //--FH
    JmlSpecExpression jse = null; //--FH
    JExpression jmlp = null; //--FH

    JmlMethodName mn = null; //--FH
    boolean enabled = false; //--FH

}
        :
                "\\result"  { self = new JmlResultExpression( sourceRef ); }
        |
                "\\old" LPAREN
                    specExpression = jmlSpecExpression[]
                    ( COMMA label:IDENT)?
                RPAREN
                { self = new JmlOldExpression( sourceRef,
                                        specExpression,
                                        (label!=null? label.getText(): null) );
        }
        |
                "\\pre" LPAREN specExpression = jmlSpecExpression[] RPAREN
                { self = new JmlPreExpression( sourceRef, specExpression ); }
        |
                "\\not_modified" LPAREN storeRefList = jmlStoreRefList[] RPAREN
                { self = new JmlNotModifiedExpression( sourceRef, storeRefList ); }
        |
                "\\only_accessed" LPAREN storeRefList = jmlStoreRefList[] RPAREN
                { self = new JmlOnlyAccessedExpression( sourceRef, storeRefList ); }
        |
                "\\not_assigned" LPAREN storeRefList = jmlStoreRefList[] RPAREN
                { self = new JmlNotAssignedExpression( sourceRef, storeRefList ); }
        |
```

```
                "\\only_called" LPAREN names = jmlMethodNameList [] RPAREN
                { self = new JmlOnlyCalledExpression( sourceRef, names ); }
        |
                "\\only_captured" LPAREN storeRefList = jmlStoreRefList[] RPAREN
                { self = new JmlOnlyCapturedExpression( sourceRef, storeRefList ); }
        |
                "\\only_assigned" LPAREN storeRefList = jmlStoreRefList[] RPAREN
                { self = new JmlOnlyAssignedExpression( sourceRef, storeRefList ); }
        |
                "\\fresh" LPAREN specExpressionList = jmlSpecExpressionList[] RPAREN
                { self = new JmlFreshExpression( sourceRef, specExpressionList ); }
        |
                "\\working_space" LPAREN expression = jExpression[] RPAREN
                { self = new JmlWorkingSpaceExpression( sourceRef, expression ); }
        |
                "\\space" LPAREN specExpression = jmlSpecExpression[] RPAREN
                { self = new JmlSpaceExpression( sourceRef, specExpression ); }
        |
                "\\duration" LPAREN expression = jExpression[] RPAREN
                { self = new JmlDurationExpression( sourceRef, expression ); }
        |
                "\\reach" LPAREN
                specExpression = jmlSpecExpression[]
                ( COMMA type = jClassTypeSpec[null, null] // WMD TODO
                        ( COMMA storeRefExpression = jmlStoreRefExpression[] )? )?
                RPAREN
                { self = new JmlReachExpression( sourceRef, specExpression, type,
                                storeRefExpression );
          /*
           * The following isn't used, but is kept as an example
           * of how to deprecate something, if you want to make
           * something deprecated.  It can be deleted when you have
           * something you really want to deprecate.
           *
           * utility.reportTrouble(
           *     new CWarning( sourceRef,
           *                   JmlMessages.DEPRECATED_REACH ));
           */
        }
        |
                infDesc:INFORMAL_DESC
                { self = new JmlInformalExpression( sourceRef, infDesc.getText() ); }
        |
                "\\nonnullelements" LPAREN specExpression = jmlSpecExpression[] RPAREN
                { self = new JmlNonNullElementsExpression( sourceRef,
                                specExpression ); }
        |
                "\\typeof" LPAREN specExpression = jmlSpecExpression[] RPAREN
                { self = new JmlTypeOfExpression( sourceRef, specExpression ); }
        |
                "\\elemtype" LPAREN specExpression = jmlSpecExpression[] RPAREN
                { self = new JmlElemTypeExpression( sourceRef, specExpression ); }
        |
                "\\type" LPAREN type = jTypeSpec[] RPAREN
                { self = new JmlTypeExpression( sourceRef, type ); }
        |
                "\\lockset"  { self = new JmlLockSetExpression( sourceRef ); }
        |
                "\\max" LPAREN specExpression=jmlSpecExpression[] RPAREN
                    {  self = new JmlMaxExpression(sourceRef,specExpression); }
        |
                "\\is_initialized" LPAREN type = jClassTypeSpec[null, null] RPAREN
// WMD TODO
                { self = new JmlIsInitializedExpression( sourceRef, type ); }
```

```
        |
                "\\invariant_for" LPAREN specExpression = jmlSpecExpression[] RPAREN
                { self = new JmlInvariantForExpression( sourceRef, specExpression ); }
        |
                self = jmlWarnExpression []
        |
                self = jmlMathModeExpression []
    |
        // FH--The following line called jmlStatePropery[] which is also now  in jmlPrimary[]
        //FH--from jmlTraceProperty[]
                (
                                "\\always"
                                {
                                        aen = Constants.OPE_TEMPORAL_ALWAYS; //--Const. val = 145
                                        //System.out.println("FH: Parsing always...");
                                }
                        |
                                "\\eventually"
                                {
                                        aen = Constants.OPE_TEMPORAL_EVENTUALLY; //--Const. val = 146
                                        //System.out.println("FH: Parsing eventually...");
                                }
                        |
                                "\\never"
                                {
                                        aen = Constants.OPE_TEMPORAL_NEVER; //--Const. val = 147
                                        //System.out.println("FH: Parsing never...");
                                }
                )
                LPAREN jse = jmlSpecExpression[] RPAREN
                //But what if two trace properties are being combined using '|' or '&'? Is that
                //covered by JmlSpecExpression?
                        {
                                //System.out.println("FH: Creating JmlTemporalTraceProperty node.");
                                        self = new JmlTemporalTraceProperty(sourceRef, aen, jse);
                        }

    |
        //FH --from jmlStateProperty[]
        (
                (
                        "\\enabled" { enabled = true; }
                |
                        "\\not_enabled" { enabled = false; }
                ) LPAREN mn =  jmlMethodName[] RPAREN
                {
                        self = new JmlTemporalStateProperty(sourceRef, enabled, mn);
                }
        )
        ;
```

Listing A.2   jmlSpecQuantifiedExprRest Rule

```
jmlSpecQuantifiedExprRest [TokenReference sourceRef]
returns [JmlSpecQuantifiedAugmentedExpression self = null]
{
    int oper = -1;
    JmlVariableDefinition[] quantifiedVarDecls;
    JmlSpecExpression predicate = null;
    JmlSpecExpression specExpression = null;

    JmlTemporalEvent ev = null; //--FH
```

```
    JExpression jt = null; //--FH

    JmlTemporalEvent ev1 = null; //--FH
    JmlTemporalEvent ev2 = null; //--FH
    JmlSpecExpression jse = null; //--FH
}
    :
    (
        (       "\\forall"      { oper = Constants.OPE_FORALL; }
        |       "\\exists"      { oper = Constants.OPE_EXISTS; }
        |       "\\max"         { oper = Constants.OPE_MAX; }
        |       "\\min"         { oper = Constants.OPE_MIN; }
        |       "\\num_of"      { oper = Constants.OPE_NUM_OF; }
        |       "\\product"     { oper = Constants.OPE_PRODUCT; }
        |       "\\sum"         { oper = Constants.OPE_SUM; }
        )
        quantifiedVarDecls = jmlQuantifiedVarDecls[] SEMI
        (
            predicate = jmlSpecExpression[]
            ( SEMI specExpression = jmlSpecExpression[] )?
            {
                if (specExpression == null) {
                    // really the predicate is optional
                    specExpression = predicate;
                    predicate = null;
                }
            }
        |
            SEMI specExpression = jmlSpecExpression[]
        )
        {
            self = new JmlSpecQuantifiedExpression( sourceRef, oper,
                quantifiedVarDecls,
                predicate == null ? null : new JmlPredicate( predicate ),
                specExpression );
        })

    |

        //try to do this the way its done for \forall--TODOFH

        // LPAREN is in jParenthesizedExpr[] -- IMPORTANT
        //RPAREN is in jParenthesizedExprRest[]
         "\\after" ev = jmlEvents[] SEMI jt =  jmlTemporalExpression[]
          {
                //System.out.println("FH: After parsed");
                self = new JmlTemporalAfterExpression(sourceRef, ev, jt,
                                                Constants.OPE_TEMPORAL_AFTER);
          }
    |
         "\\before" ev = jmlEvents[] SEMI  jt = jmlTemporalExpression[]
         {
                //System.out.println("FH: Before parsed");
                self = new JmlTemporalBeforeExpression(sourceRef, ev, jt,
                                                Constants.OPE_TEMPORAL_BEFORE);
         }
    |
        //FH--replaced jmlTraceProperty[] with jmlSpecExpression[] in 'between' subrule
        //Originally, jmlTraceProperty[] was called from the \between subrule

        //!TODO! -- Should I call jmlPrimary[] here instead of jmlSpecExpression[]
        //Also: Should there be another semicolon after the second set of events?
         "\\between" ev1 = jmlEvents[] SEMI ev2 =  jmlEvents[] jse = jmlSpecExpression[]
         {
```

```
                //System.out.println("FH: Between parsed");
                self = new JmlTemporalBetweenExpression(sourceRef, ev1, ev2, jse,
                                        Constants.OPE_TEMPORAL_BETWEEN);
        }
    |
        "\\atmost" maxNum: INTEGER_LITERAL SEMI ev = jmlEvents[]
        {
                //System.out.println("FH: At most parsed");
                self = new JmlTemporalAtMostExpression(sourceRef, ev,
                        Integer.parseInt(maxNum.getText()), Constants.OPE_TEMPORAL_ATMOST);
        }
    ;
```

Listing A.3 jmlTemporalExpression Rule

```
jmlTemporalExpression []
returns [JExpression self = null]
{
        JExpression subExpression = null;
        JmlTemporalEvent ev = null;
        JExpression temp = null;
        JmlTemporalTraceProperty tmp = null; //this can be commented.

        //boolean isUnlessUntil = false;

        //What really do we need to do here for TokenReference?
        //TokenReference sourceRef = utility.buildTokenReference( LT(1) );
}
        :
                //This goes to jParenthesizedExpression and then jmlPrimary
                self = jmlImpliesExpression[]
                {
                        //Should I wrap this in a JmlTemporalExpression here?
                        //(To ensure that what is returned above is a
                        //JmlTemporalExpression|JmlTemporalUntilExpression|JmlTemporalUnlessExpression)

                        //why does trying to create this node give errors?
                        //selfcopy = self;
                        //self = new JmlTemporalExpression(selfcopy.getTokenReference(), selfcopy);

                }
                (
                        "\\unless"  ev = jmlEvents[]
                        {
                                //isUnlessUntil = true;
                                //System.out.println("FH: Unless parsed.");
                                self = new JmlTemporalUnlessExpression(self.getTokenReference(), self, ev);
                        }
                |
                        "\\until"   ev = jmlEvents[]
                        {
                                //System.out.println("FH: I'm in until and I have: " + self.getClass());
                                //isUnlessUntil = true;
                                //System.out.println("FH: Until parsed.");
                                self = new JmlTemporalUntilExpression(self.getTokenReference(), self, ev);
                        }
                )?
                {
                        //System.out.println("\t\tWe are in : " + self.getClass());
                        //if (self instanceof JmlRelationalExpression) {
                                //JmlRelationalExpression temp1 = (JmlRelationalExpression) (self);
                                //System.out.println("FH: RelExpr of type: " + temp1.oper());
```

```
                        //}
                }
;
```

Listing A.4 Type JmlTemporalAfterExpression

```
package org.jmlspecs.checker;

import org.multijava.mjc.CExpressionContextType;
import org.multijava.mjc.CStdType;
import org.multijava.mjc.CType;
import org.multijava.mjc.JExpression;
import org.multijava.util.compiler.PositionedError;
import org.multijava.util.compiler.TokenReference;

public class JmlTemporalAfterExpression extends JmlTemporalSequenceExpression
                implements Cloneable
{

        public JmlTemporalAfterExpression(TokenReference where,
                        JmlTemporalEvent jmltemporalevent, JExpression jtExpression, int operator) {
                super(where, operator, jmltemporalevent);
                //jEvent = je;
                jte = jtExpression;

        }

        public CType getType() {
                //return CStdType.Boolean;
                //if (jte.getType() instanceof JmlTemporalType)
                        return JmlStdType.temporalType;
                //else
                        //throw new UnsupportedOperationException();
        }



        public JExpression getJte() {
                return jte;
        }


        // TODO check the return type of typecheck in JmlTemporalAfterExpression
        public JExpression typecheck(CExpressionContextType context)
                        throws PositionedError {

                try {
                        super.typecheck(context);

                        //System.out.println("FH: I'm typechecking JmlTemporalAfterExpression.");
                        //super.typecheck(context);

                        jte.typecheck(context);

                        //Added on 16 AUG 2008
                        //jte = jte.typecheck(context);

                        //System.out.println("FH: The type of After's expression is " + jte.getType());

                        if ((jte.getType() != JmlStdType.temporalType))
                                //throw new UnsupportedOperationException();
```

```
                context.reportTrouble(new PositionedError(getTokenReference(), JmlMessages.TEMPORAL_TEMPORALTYPE_EXPECTED,
                    jte.getType()));

        }


        catch (PositionedError e) {
            context.reportTrouble(e);
        }

        return this;
    }


    private JExpression jte;
    //private JmlTemporalEvent jEvent; -- now in superclass






}
```

Listing A.5 Runtime Temporal State Machine

```
package org.jmlspecs.jmlrac.runtime;


import java.util.ArrayList;
import java.util.HashMap;
import org.jmlspecs.checker.JmlTemporalFormula;
import org.jmlspecs.jmlrac.runtime.TemporalObserver;


public class JMLRuntimeTemporalStateMachine {
    public JMLRuntimeTemporalStateMachine(TemporalObserver o,  int indexOfTemporalFormulaInType) {
        this.myObserver = o;

        this.transitionTable = new HashMap();
        this.temporalFormulaNumber = indexOfTemporalFormulaInType;

    }

    public void setStartState(int stateNumber) {
        this.currentState = this.getState(stateNumber);
    }

    public TemporalState getState(int stateNumber) {
        TemporalState found = null;

        for (int i = 0; i < listOfStates.size(); i++) {
            TemporalState temp = (TemporalState) listOfStates.get(i);
            if ( temp.getStateNumber() == stateNumber)
                found = temp;
        }

        return found;
    }

    public void addTransition(int fromStateNumber, String event,  int toStateNumber) {
        TemporalState fromState = this.getState(fromStateNumber);
        TemporalState toState = this.getState(toStateNumber);
```

```

                Object otbl = transitionTable.get(fromState);
                transitionTable.remove(fromState); //removing now and we’ll add modified entry later

                HashMap newTable;
                if (otbl == null) {
                        newTable = new HashMap();
                        newTable.put(event, toState);
                } else {
                        newTable = (HashMap) otbl;
                        if (newTable.get(event) != null) {
                                System.err.println("FH: The key " + event + " already exists in the table with value " + newTable.get(event)
                                                + " but I’ll insert the new value for this key anyway.");
                        }
                        newTable.put(event, toState);
                }
                transitionTable.put(fromState, newTable);
        }

        public void performTemporalChecks() {
                if (this.currentState.isEnabledNotEnabledStateProperty()) {
                        //inform
                }
                else if  (this.currentState.isTracePropertyCheckingState()) {
                        //this.setChanged();
                        //this.notifyObservers();
                        this.myObserver.update$temporalspec(this, null);
                }
        }

        public void consume(String newTemporalEvent) {
                if (this.currentState.isEnabledNotEnabledStateProperty()) {
                        this.currentState.informNewEvent(newTemporalEvent);
                }


                HashMap transitionsFromCurrentState = (HashMap) this.transitionTable.get(currentState);
                if (transitionsFromCurrentState != null) {
                        if (transitionsFromCurrentState.get(newTemporalEvent) != null) {
                                currentState =  (TemporalState) transitionsFromCurrentState.get(newTemporalEvent);
                        }
                }
        }

        public int getTemporalFormulaNumber() {
                return this.temporalFormulaNumber;
        }

        public void setStateList(ArrayList listOfStates) {
                this.listOfStates = listOfStates;
        }

        public String getReasonForCurrentNonAcceptingState() {
                return currentState.getReasonForBeingNonAccepting();
        }

        public boolean inFinalState() {
                return currentState.isAcceptingState();
        }


        private ArrayList listOfStates;
        private HashMap transitionTable;
        private TemporalState currentState;
```

```
        private int temporalFormulaNumber; //the number in the type that this machine represents
        private TemporalObserver myObserver;

}
```

APPENDIX B File Operations Example

Listing B.1 TemporalSpecFileOps.java: Driver-1

```
//File TemporalSpecFileOps.java

import java.io.*;
import org.jmlspecs.jmlrac.runtime.*;

public class TemporalSpecFileOps implements TemporalObserver {

    //TS0
    //@ public static temporal (\always(\not_enabled(writeFile)) \unless \call(openFile));

    //TS1
    //@ public static  temporal (\after \normal(openFile); (\always (\enabled (writeFile)) \until \call (closeFile)));

    //TS2
    //@ public static temporal (\after \normal(closeFile); (\always (\not_enabled(writeFile)) \unless \call(openFile)));

    public static boolean openFlag = false;

    /** Opens the file (Sets field 'openFlag' to true).
     *  A file must be close after its opened.
    */
    public static void openFile(File f) {
        try {
            f.createNewFile();
            openFlag = f.canWrite();
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public static void writeFile(File f) throws Exception {
        if (!openFlag) {
            throw new Exception("Cannot write to file.");
        }
        try {
        FileOutputStream fo = new FileOutputStream(f);
        fo.write(97);
        fo.close();
        } catch(IOException ef) {
            System.out.println(ef);
        }
    }

    public static void closeFile(File f) {
        openFlag = false;
    }

```

```java

    public static void main(String[] args) {
        String filename = "file1.txt";

        try {
            File f = new File(filename);
            openFile(f);
            writeFile(f);
            closeFile(f);
            writeFile(f);
        } catch (Exception e) {
            System.out.println(e);
        }

    }

}
```

Listing B.2 TemporalSpecFileOps.java: Driver-2

```java
//File TemporalSpecFileOps.java

import java.io.*;
import org.jmlspecs.jmlrac.runtime.*;

public class TemporalSpecFileOps implements TemporalObserver {

    //TS0
    //@ public static temporal (\always(\not_enabled(writeFile)) \unless \call(openFile));

    //TS1
    //@ public static  temporal (\after \normal(openFile); (\always (\enabled (writeFile)) \until \call (closeFile)));

    //TS2
    //@ public static temporal (\after \normal(closeFile); (\always (\not_enabled(writeFile)) \unless \call(openFile)));

    public static boolean openFlag = false;

    /** Opens the file (Sets field 'openFlag' to true).
     *  A file must be close after its opened.
    */
    public static void openFile(File f) {
        try {
            f.createNewFile();
            openFlag = f.canWrite();
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public static void writeFile(File f) throws Exception {
        if (!openFlag) {
            //throw new Exception("Cannot write to file.");
        }
        try {
        FileOutputStream fo = new FileOutputStream(f);
        fo.write(97);
        fo.close();
        } catch(IOException ef) {
            System.out.println(ef);
        }
    }
```

```java

    public static void closeFile(File f) {
        openFlag = false;
    }


    public static void main(String[] args) {
        String filename = "file1.txt";

        try {
            File f = new File(filename);
            //openFile(f);
            writeFile(f);
            closeFile(f);
            //writeFile(f);
        } catch (Exception e) {
            System.out.println(e);
        }

    }

}
```

Listing B.3   TemporalSpecFileOps.java: Driver-3

```java
//File TemporalSpecFileOps.java

import java.io.*;
import org.jmlspecs.jmlrac.runtime.*;

public class TemporalSpecFileOps implements TemporalObserver {

    //TS0
    //@ public static temporal (\always(\not_enabled(writeFile)) \unless \call(openFile));

    //TS1
    //@ public static  temporal (\after \normal(openFile); (\always (\enabled (writeFile)) \until \call (closeFile)));

    //TS2
    //@ public static temporal (\after \normal(closeFile); (\always (\not_enabled(writeFile)) \unless \call(openFile)));

    public static boolean openFlag = false;

    /** Opens the file (Sets field ’openFlag’ to true).
     *  A file must be close after its opened.
    */
    public static void openFile(File f) {
        try {
            f.createNewFile();
            //openFlag = f.canWrite();
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public static void writeFile(File f) throws Exception {
        if (!openFlag) {
            throw new Exception("Cannot write to file.");
        }
        try {
        FileOutputStream fo = new FileOutputStream(f);
        fo.write(97);
```

```
        fo.close();
        } catch(IOException ef) {
            System.out.println(ef);
        }
    }

    public static void closeFile(File f) {
        openFlag = false;
    }


    public static void main(String[] args) {
        String filename = "file1.txt";

        try {
            File f = new File(filename);
            openFile(f);
            writeFile(f);
            closeFile(f);
            //writeFile(f);
        } catch (Exception e) {
            System.out.println(e);
        }

    }

}
```

Listing B.4 TemporalSpecFileOps.java: Driver-4

```
//File TemporalSpecFileOps.java

import java.io.*;
import org.jmlspecs.jmlrac.runtime.*;

public class TemporalSpecFileOps implements TemporalObserver {

    //TS0
    //@ public static temporal (\always(\not_enabled(writeFile)) \unless \call(openFile));

    //TS1
    //@ public static  temporal (\after \normal(openFile); (\always (\enabled (writeFile)) \until \call (closeFile)));

    //TS2
    //@ public static temporal (\after \normal(closeFile); (\always (\not_enabled(writeFile)) \unless \call(openFile)));

    public static boolean openFlag = false;

    /** Opens the file (Sets field 'openFlag' to true).
     *  A file must be close after its opened.
    */
    public static void openFile(File f) {
        try {
            f.createNewFile();
            openFlag = f.canWrite();
        } catch (Exception e) {
            System.out.println(e);
        }
    }

    public static void writeFile(File f) throws Exception {
        if (!openFlag) {
```

```
            throw new Exception("Cannot write to file.");
        }
        try {
        FileOutputStream fo = new FileOutputStream(f);
        fo.write(97);
        fo.close();
        } catch(IOException ef) {
            System.out.println(ef);
        }
    }

    public static void closeFile(File f) {
        //openFlag = false;
    }


    public static void main(String[] args) {
        String filename = "file1.txt";

        try {
            File f = new File(filename);
            openFile(f);
            writeFile(f);
            closeFile(f);
            writeFile(f);
        } catch (Exception e) {
            System.out.println(e);
        }

    }

}
```

APPENDIX C Bank Account Example

Listing C.1 TemporalSpecBankAC.java

```
//package org.jmlspecs.temporalspec.temporalfiles;

import org.jmlspecs.jmlrac.runtime.*;

public  class TemporalSpecBankAC implements TemporalObserver  {

    //@ public temporal (\after \normal (openAC); (\after \normal (activateAC); (\always (balance>0) | \eventually (swissType)) \unless \call
           (suspendAC)) );

    private int balance = 0;
    private  boolean swissType = false;

    public void openAC() { /* Opens A/C -- just to show temporal events */ }
    public void activateAC() { /* Activates A/C -- just to show temporal events */ }
    public void suspendAC()
    {
        /* Temporarily deactivates A/C; this can be reversed using activateAC --
        just to show temporal events */
    }

    public int getBalance() { return balance; }

    public void setBalance(int n) {
        balance = n;
    }

    public void setSwissType(boolean onOrOff) {
        swissType = onOrOff;
    }


    public static void main(String[] args)  {
        TemporalSpecBankAC ac1 = new TemporalSpecBankAC();
        ac1.openAC();
        ac1.setBalance(-100);
        ac1.setBalance(200);
        ac1.activateAC();
        ac1.setBalance(-300); //positive or negative
        //ac1.setSwissType(true);
        ac1.suspendAC();
    }
}
```

BIBLIOGRAPHY

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–364, New York, NY, USA, 2005. ACM.

[2] Eric Allen. Diagnosing Java code: Assertions and temporal logic in Java programming: `http://www.ibm.com/developerworks/java/library/j-diag0723.html`, 2002.

[3] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In K. Havelund and G. Roşu, editors, *Electronic Notes in Computer Science*, volume 55(2). Elsevier Science BV, 2001.

[4] F. Bellegarde, J. Groslambert, M. Huisman, O. Kouchnarenko, and J. Julliand. Verification of liveness properties with JML. Technical Report RR-5331, INRIA, 2004.

[5] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[6] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *FMCO*, pages 342–363, 2005.

[7] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, April 2003. Technical Report 03-09, Department of Computer Science, Iowa State University.

[8] Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). Technical Report 02-05, Department of Computer Science, Iowa State University, March 2002. In SERP 2002, pp. 322-328.

[9] Yoonsik Cheon and Ashaveena Perumandla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15(1):7–25, March 2007.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[11] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, May 2006.

[12] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[13] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *STTT*, 4(1):34–56, 2002.

[14] Doron Drusinsky. The Temporal Rover and the ATG rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[15] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, Kansas State University, University of Massachusetts, University of Hawai'i, , 1998.

[16] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, New York, NY, USA, 1998. ACM.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[18] A. Giorgetti and J. Groslambert. JAG: JML Annotation Generation for verifying temporal properties. In *FASE'2006, Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*, pages 373–376, Vienna, Austria, March 2006. Springer.

[19] John Hatcliff and Matthew B. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 39–58, London, UK, 2001. Springer-Verlag.

[20] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[21] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[22] Ying Jin. Formal verification of protocol properties of sequential Java programs. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*, pages 475–482, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, Department of Computer Science, January 2006. Also *ACM SIGSOFT Software Engineering Notes*, 31(3):1-38, March 2006.

[24] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[25] Robby. Bandera Specification Language: A specification language for software model checking, 2000. Master's thesis, Kansas State University.

[26] Kerry Trentelman and Marieke Huisman. Extending JML specifications with temporal logic. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 334–348, London, UK, 2002. Springer-Verlag.