temporaljmlc: A JML Runtime Assertion Checker Extension for Specification and Checking of Temporal Properties

Faraz Hussain and Gary T. Leavens

CS-TR-10-08
July 2010

To appear in SEFM 2010.

Keywords: Temporal Logic, LTL, verification, specification languages, specification patterns, Hoare logic.

2010 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, methodologies, tools; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, formal methods, programming by contract; D.3.3 [*Programming Languages*] Language Constructs and Features — abstract data types, data types and structures, patterns; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, invariants, logics of programs, pre- and post-conditions, specification techniques; F.4.1 [*Theory of Computation*] Mathematical Logic and Formal Languages — Temporal logic.

Department of Electrical Engineering and Computer Science
4000 Central Florida Blvd.
University of Central Florida
Orlando, FL 32816, USA

temporaljmlc: A JML Runtime Assertion Checker Extension for Specification and Checking of Temporal Properties

Faraz Hussain
School of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL, USA.
Email: fhussain@eecs.ucf.edu

Gary T. Leavens
School of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL, USA.
Email: leavens@eecs.ucf.edu

***Abstract*—Most mainstream specification languages primarily deal with a program's functional behavior. However, for many common problems, besides the system's functionality, it is necessary to be able to express its temporal properties, such as the necessity of calling methods in a certain order. We have developed temporaljmlc, a tool that performs runtime assertion checking of temporal properties specified in an extension of the Java Modeling Language (JML). The benefit of temporaljmlc is that it allows succinct specification of temporal properties that would otherwise be tedious and difficult to specify.**

***Keywords*-temporal specification; runtime assertion checking; specification patterns; Java Modeling Language; temporaljmlc.**

I. Introduction

Programmers use specification languages to help write correct programs by facilitating program verification and dynamic checking of a software implementation with respect to its specifications. They also aid in providing better documentation, especially to programmers who are extending the work of others and who only have access to the system's API, but not its implementation.

Design-by-contract (DBC) techniques [1], popularized by Bertrand Meyer in the language Eiffel, are widely employed for the specification and checking of computer programs. The DBC approach helps make programs more modular in that they provide a level of "separation of concerns", by allowing the programmer to focus mainly on the implementation, whereas the contract-checking tool handles the responsibility of enforcing programmer-defined specifications.

Most current program specification techniques, such as DBC, are primarily used to describe a system's functional behavior. However, for many programs, there is a natural need to provide a temporal description of the system along with its functional behavior. For example, consider the specification:

> After an account is opened and then activated, either its balance is always positive or it must, at some point, be designated as a swissType account, unless it is marked for suspension.

A program that checks this specification at runtime can be written by setting and unsetting of flags for the expected "events" (i.e. the successful opening and activation of the accounts and the request for account suspension). However, its specification can be expressed in a more intuitive manner if *temporal specification* constructs are also available in a specification language.

In modern programming techniques, a specific task is typically performed by sending a message to an object (i.e. by calling a method). Such method invocations form the basis of our definition of *temporal events*. The calling and termination of methods are, therefore, our temporal control points. In addition, we distinguish between the normal termination (i.e. without throwing an exception) and exceptional termination of a method. By temporal specification, we refer to the way program properties are expected to hold, as delimited by temporal events.

We have implemented temporaljmlc, a tool that allows the specification of a certain class of temporal properties of programs and checks them at runtime. Our primary contribution is that temporaljmlc provides an integrated way of specifying both functional and temporal properties of programs without the need for separating these two kinds of constraints.

Instead of model checking temporal properties, in this paper we describe an approach that checks temporal properties dynamically, using runtime assertion checking. One reason for this is that runtime assertion checking is an important and commonly used technique for checking JML specifications. Runtime assertion checking is also a useful compliment to model checking, since it does not suffer from state space explosion problems.

In Section II we discuss the temporal logic background required for introducing the temporal extension to the JML. The approach used in the implementation of temporaljmlc is described in Section III. This is followed by a case study (§IV) with programs showing both temporal and regular JML specifications of a bank account class. Next, we describe the process of converting temporal specifications written in temporalJML into finite automata (§V). This is the main theoretical contribution of our research. We then discuss certain issues regarding the semantics of temporalJML (§VI). Section VII compares our work with related work in the area. Section VIII discusses limitations of our implementation and scope

Listing 1: JML Invariant specification

```
//@ protected invariant 0 <= hour && hour <= 23;
```

Listing 2: JML method contracts

```
//@ requires true;
//@ ensures 0 <= \result && \result <= 23;
public /*@ pure @*/ int getHour() { return hour; }
```

for future work and is followed by the conclusion (§IX).

II. Temporal Logic and Specifications

Temporal logic is used extensively in the area of specification and verification of computer programs, especially concurrent programs [2], to prove properties such as deadlock-avoidance. An example is the model checker SPIN [3], which uses Linear Temporal Logic (LTL) to specify the properties that a system needs to respect. Another example is the Bandera Specification Language (BSL) [4], [5] which is used by the Bandera project [6], [7] as an input language for temporal specifications. The BSL uses temporal specification patterns [8] to express properties that the programmer wishes to express.

Temporal logics such LTL, Computational Tree Logic (CTL) or the μ-calculus are powerful, general logics which are not tied to any specific system or application. However, we consider them overly mathematical for the average programmer. Therefore, in our extention of JML, we follow Trentleman and Huisman [9] in using Bandera-style patterns [10], [8] to describe temporal specifications.

The problem we address in this paper is to specify and dynamically check temporal properties of sequential programs. Our specifications can express temporal properties over a sequence of method-related events. Extending this to concurrent programs is left as future work (§VIII).

Temporal logic extension to the Java Modeling Language

The Java Modeling Language (JML) [11], [12], [13] is a behavioral interface specification language which allows specifications to be written as annotation to Java code.

Invariants (Listing 1) allow the imposition of restrictions on class data members in *visible states* (i.e. post object-construction, except inside method bodies).

Listing 2 shows an example of a JML specification of a method contract. It essentially represents the Hoare triple `{P}S{Q}`, where `P` is the precondition (**`requires`**), `Q` the postcondition (**`ensures`**) and `S` the piece of code (method `getHour`). Any JML compiler must ensure that if `S` is executed when `P` holds, then in the normal post-state of `S`, `Q` must be true. Here, the contract specifies that `getHour` can always be called; however, the value returned by `getHour` must be between 0 and 23 (both inclusive).

Consider again the temporal constraint regarding a bank account class mentioned in the Introduction (§I). Currently, there's no obvious way to specify such properties in JML. We can specify this using a complicated set of JML annotations, like ghost and model fields and model methods. Excerpts from a file with the specification of a bank account class in JML are shown in Listing 3. The setting and unsetting of JML's ghost variables[1] is used to simulate a finite state machine.

Listing 3: BankAC Specification using JML

```
package org.jmlspecs.temporalspec.bankac_casestudy;
import java.util.ArrayList;

public  class RegularSpecBankAC   {
  //@ public ghost static ArrayList
      listOfInstances = new ArrayList();
  //@ public model JMLDataGroup bank_spec;
  //@ public represents bank_spec =
      JMLDataGroup.IT;

  //@ public ghost boolean openAC_normal; in
      bank_spec;
  //@ public ghost boolean openAC_called; in
      bank_spec;
  //@ public ghost boolean openAC_exceptional; in
      bank_spec;

  //@ public ghost boolean activateAC_normal; in
      bank_spec;
  //@ public ghost boolean activateAC_called; in
      bank_spec;
  //@ public ghost boolean
      activateAC_exceptional; in bank_spec;

  //@ public ghost boolean suspendAC_normal; in
      bank_spec;
  //@ public ghost boolean suspendAC_called; in
      bank_spec;
  //@ public ghost boolean suspendAC_exceptional;
      in bank_spec;

  //@ public model boolean is_swissType; in
      bank_spec;
  //@ private represents is_swissType = swissType;

  //@ public model boolean is_balancePositive;
      in bank_spec;
  //@ private represents is_balancePositive =
      (balance>0);


  private  boolean tp_bal;  //@in bank_spec;

  private  boolean tp_swiss; //@in bank_spec;
  private boolean never_called_update_tp_swiss;
      //@in bank_spec;


  //@ public model boolean check_condition; in
      bank_spec;
  //@ private represents check_condition =
      (tp_bal || tp_swiss);

  /*@ private model void update_tp() {
      if (openAC_normal && activateAC_normal &&
          !suspendAC_called) {
        update_tp_bal();
        update_tp_swiss();
      }
    }
    @*/
```

[1]A ghost variable is a specification-only variables which can be used to add state; its value can be set using the JML **`set`** statement.

```
//@ assignable bank_spec;
//@ private model void update_tp_bal() {
//@ if (!tp_bal)
//@  return;
//@
//@  if (!is_balancePositive) {
//@   tp_bal = false;
//@  }
//@
//@}

//@ assignable bank_spec;
//@ private model void update_tp_swiss() {
//@  if (never_called_update_tp_swiss) {
//@    never_called_update_tp_swiss = false;
//@    tp_swiss = false;
//@}
//@
//@  if (is_swissType) {
//@     tp_swiss = true;
//@  }
//@
//@}

//@ ensures (check_condition == true);
/*@ public model pure boolean final_check() {
    return check_condition;
  }
@*/

//@ assignable bank_spec;
//@ private model void init_temporal() {
//@  listOfInstances.add(this);
//@  tp_bal = true;
//@  tp_swiss = true;
//@  never_called_update_tp_swiss = true;
//@}

/*@ public  static model boolean
    all_final_check() {
  for (int i = 0; i < listOfInstances.size();
      i++) {
    RegularSpecBankAC obj = (RegularSpecBankAC)
        listOfInstances.get(i);
    if (!obj.final_check())
      System.out.println("Temporal exception: "
          + obj.toString());
  }
  return true;
}
@*/

//@ assignable bank_spec;
public RegularSpecBankAC() {
    //@ debug init_temporal();
}

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

public void openAC() {
    //@ set openAC_normal = false;
    //@ set openAC_exceptional = false;
```

In order to express temporal specifications, temporaljmlc uses an extension of the JML, which we call temporalJML. One of our main contributions is the ability of temporaljmlc to automatically generate this finite state machine for any given temporal specification written in temporalJML; and thus avoid its manual specification using ghost variables. The grammar for temporalJML (Table I) is based on the extension to JML proposed by Trentelman and Huisman [9] whose work is inspired by the SanTos Specification Patterns project [10].

Patterns and Scopes: In the Specification Patterns project, a pattern is defined over one of five *temporal scopes*: global, before, after, between, and after-until [14]. temporalJML also uses *occurrence specification patterns* [15] 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**).

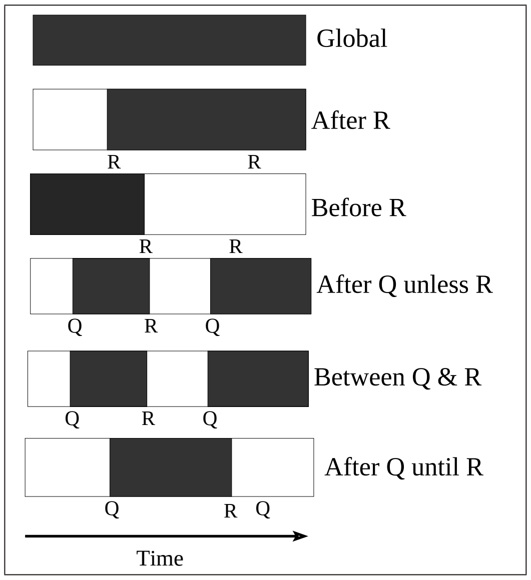


Fig. 1: Temporal Property Specification Scopes use a modified semantics of the Bandera project scopes, [14]

Our implementation of temporal specification constructs is based on a modified semantics (Figure 1) of these temporal pattern scopes. *Global* scope refers to the entire timeline. The *After R* scope refers to the the part of the timeline after the first occurrence of event R. The *Before R* scope refers to the part of the timeline before the first occurrence of event R. 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.

Note from the grammar (Table I) that *trace properties* are used to describe the functional properties of a system over a given occurrence pattern. A *temporal formula* can contain a trace property and is used to describe the temporal behavior of a system constrained by the occurrence of temporal events.

III. The Problem and Approach Used

The problem is to augment JML with constructs that enable the specification of temporal properties of a program. This involves all the phases of compiler construction following the defined semantics ([9, §5.1],[16, §4.1]) for temporal specifications, including generating runtime assertion checking code

TABLE I: The temporalJML grammar (which is based on the temporal logic extension to JML suggested in [9])

⟨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⟩

which performs the actual dynamic checking of these temporal specifications on executing the program.

The JML runtime assertion checker (RAC), `jmlc`, is built on top of the Multijava compiler, `mjc` [17]. The temporaljmlc tool has been implemented by extending `jmlc`, enhancing it with temporal specification capabilities by adding temporal specification constructs (Table I). The current implementation of temporaljmlc is based on the JML2 compiler codebase.[2] For a detailed description of the semantics of the grammar on which temporalJML is based, see [9, §5].

The basic approach used to generate RAC code for temporal specifications is to create one finite state machine, with accepting and non-accepting states, per temporal specification. When the JML-augmented Java code is compiled using temporaljmlc, code is produced to construct instances of the finite state machine at runtime. The transitions of these finite state machines are the temporal events or method control points (viz. **\call**, **\normal**, **\exceptional** and **\terminates** for any method).

For every temporal specification, there is a variable representing each of its basic trace properties (i.e. those that do not contain one of the temporal state properties viz. **\enabled** and **\not_enabled**, or the & and | operators). The temporal state machine causes these variables to be updated as appropriate. The values of the variables representing the trace property of each temporal specification are checked, on program termination, to decide if a trace-property violation error is to be reported, because certain specification violations (e.g., **\eventually** properties) cannot be ascertained before the program completes execution. 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.

We follow the approach used in Yoonsik Cheon's PhD thesis by inserting code in *wrapper methods* [18, §4.3] in order to check temporal specifications. For any method `m`, the original method is renamed `orig$m` and a wrapper for it is created with the name `m`. The main template of such wrapper methods is shown in Listing 4.

The checks for temporal specification are on lines 2, 4, 9, 20 and 26. The call to the original method `m` is on line 8. The temporal event `m$called` is deemed to have occurred as soon as we enter the wrapper method. Temporal specifications are checked (line 4) before the call to the original renamed method, just like the check for invariants (line 3). On normal method termination, the event `m$normal` is added to the event list (line 9), before the check for method postconditions. In case the method throws an exception, the event `m$exceptional` is added to the even list (line 20), before the check for the method's exceptional postcondition. In both cases (i.e. normal or exceptional method termination), temporal specifications are checked (line 26) before leaving the wrapper, similar to the checks for invariants and history constraints (lines 25 and 27 respectively).

IV. Case Study: Specification of a Bank Account Class

Consider the temporal specification of a bank account class using temporalJML shown in Listing 5. It seeks to impose the contraint that once a bank account has been opened and activated, either its balance must always remain positive or that it must, at some point, be designated as a *swissType* account, unless the account itself has been suspended.[3]

Compare this temporalJML specification with the excerpts from a bank account class in Listing 3 which contains the equivalent specification in plain JML.[4] Note how complicated[5] this specification is compared to the one which uses temporalJML. Essentially, the specifications in Listing 3 had to be written such that ghost variables were used to keep track of the appropriate "state" the bank account object is in. It uses the values of the flags to determine when to check the functional properties and also to evaluate if any of them has been violated. This requires the cumbersome use of model methods, ghost and model fields, and even adding new fields (viz. `tp_bal`, `tp_swiss`, `never_called_update_tp_swiss`) to the class to aid specification and checking of trace properties. On the other hand, code written using temporalJML (Listing 5) can perform this task with a single line of specification.

[2]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.

[3]Complete code available at: http://www.cs.ucf.edu/~fhussain/temporaljml/TemporalSpecBankAC.java.

[4]Complete code available at: http://www.cs.ucf.edu/~fhussain/temporaljml/RegularSpecBankAC.java.

[5]The file containing the specification of the bank account class in plain JML is 4 times larger than the one which uses temporalJML.

Listing 4: The wrapper method approach [18, §4.3] used for checking temporal specifications

```
T m(T1 x1, : : :, Tn xn) {
       temporalEventList.add(m$called);
       checkInv$S();
       checkTemporalForumlas();
       checkPre$m$S(x1, : : :, xn);
       T rac$result;
        try {
              rac$result = orig$m(x1, :::, xn);
              temporalEventList.add(m$normal);
              checkPost$m$S(x1, : : :, xn,
                  rac$result);
              return rac$result;
        }
        catch (JMLEntryPreconditionError rac$e) {
              throw new
                  JMLInternalPreconditionError(rac$e);
        }
        catch (JMLAssertionError rac$e) {
              throw rac$e;
        }
        catch (Throwable rac$e) {
              temporalEventList.add(m$exceptional);
              checkXPost$m$S(x1, : : :, xn,
                  rac$e);
        }
        finally {
              if (/*no postcondition
                  violation?*/) {
                      checkInv$S();
                      checkTemporalFormulas();
                      checkHC$S();
              }
        }
      }
```

Listing 5: BankAC temporal specification

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

The main drawback of the plain-JML specification of the bank account class is that we had to manually create a finite state machine to represent the temporal properties. Its state transitions were simulated by changing the values of the ghost variables. Model methods were used to check the functional properties when the machine entered an appropriate state on program execution. These values were checked at program completion to indicate property violations.

A. Generated code and RAC for the Bank Account class

The runtime assertion checking of temporal specifications builds on the technique described in Yoonsik Cheon's Ph.D. thesis [18]. We earlier described (§III) how the wrapper method approach (Listing 4) has been used to also generate code for checking temporal specifications. For a detailed

Listing 6: Constructor in the generated code for temporal bank account class (Also see Listing 7)

```
public TemporalSpecBankAC() {
      :
 internal$$init$();
      :
 finally {
  listOfInstances$temporalspec.add(this);
  init$instance$temporalspecs
       $RuntimeTemporalMachines();
      :
}
```

explanation of the code generated[6] by temporaljmlc see [16, §3].

In the rest of this section, we use the code generated by temporaljmlc for the bank account class to demonstrate how we check temporal specifications.

B. Temporal State Machine

A state machine is created for each temporal specification. In fact, for a *non-static* temporal specification, there is one machine per object and we therefore mantain a list of all objects created (Listing 6). In the code generated by the runtime assertion checker, these temporal machines are represented by the type JMLRuntimeTemporalStateMachine ([16, §A.5]). The machines are generated after parsing of temporal specifications.

The RAC code for initialization of the temporal state machine is shown in Listing 7. Machine initialization is realized using a rudimentary implementation of the *Observer Pattern* [19], which is required for appropriate calls to trace property update methods. Essentially, the this object (which is of type TemporalSpecBankAC) becomes an *observer* (line 5 in Listing 7) of the runtime temporal state machine represented by tsm$temporalspec$TF0. The file TemporalSpecBankAC.java has only one temporal specification (Listing 5) and the corresponding machine is represented by the variable tsm$temporalspec$TF0.

Temporal events are recorded using wrapper methods (Listing 4). These events are fed to the machines so that they make appropriate transitions. If the machine is in a trace-property-checking-state, the observer's update$temporalspec method is called, which in turn calls updater methods for all basic trace properties comprising that particular temporal formula's trace property.

The finite state machine generated by temporaljmlc on compiling the bank account class with the given temporal specification (Listing 5) is pictorially depicted in Figure 2. The code generated for the construction of this machine is shown in Listing 7. The start state (State0) is shown by an

[6]The code generated by temporaljmlc for the bank account class containing the temporal specification is available at: http://www.cs.ucf.edu/~fhussain/temporaljml/TemporalSpecBankAC.java.gen

Listing 7: Runtime Temporal State Machine initialization

```
public void
init$instance$temporalspecs$RuntimeTemporalMachines()
{
  //------Code for temporal formula  0---------
  tsm$temporalspec$TF0 = new
      JMLRuntimeTemporalStateMachine(this, 0 );
  ArrayList listOfStatesForMachineNumber0 = new
      ArrayList();
  listOfStatesForMachineNumber0.add(new
      TemporalState(0,false,true,"null",false) );
  listOfStatesForMachineNumber0.add(new
      TemporalState(1,false,true,"null",false) );
  listOfStatesForMachineNumber0.add(new
      TemporalState(2,true,true,"null",false) );
  listOfStatesForMachineNumber0.add(new
      TemporalState(3,false,true,"null",false) );
  tsm$temporalspec$TF0.setStateList(
        listOfStatesForMachineNumber0);
  tsm$temporalspec$TF0.setStartState(0);
  tsm$temporalspec$TF0.addTransition(2,
   "suspendACLParenRParenV$temporalspec$called",3);
  tsm$temporalspec$TF0.addTransition(1,
   "activateACLParenRParenV$temporalspec$normal",2);
  tsm$temporalspec$TF0.addTransition(3,
   "activateACLParenRParenV$temporalspec$normal",2);
  tsm$temporalspec$TF0.addTransition(0,
   "openACLParenRParenV$temporalspec$normal",1);
  //------End of temporal formula  0---------
}
```

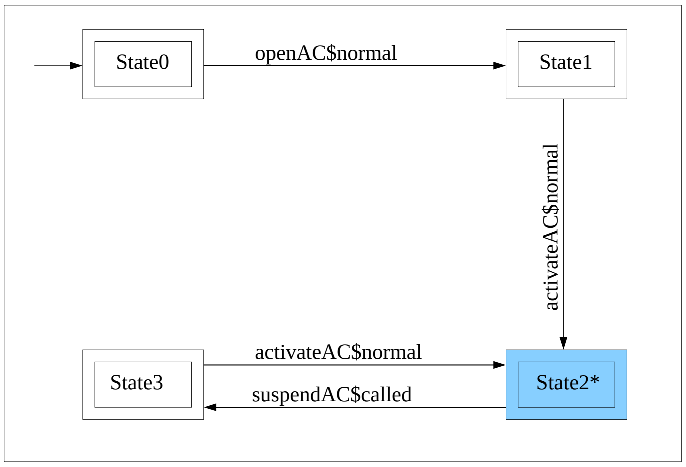


Fig. 2: BankAC Temporal State Machine Automaton

incoming arrow. The accepting states[7] (`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.

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

Listing 8: Runtime machine's temporal checks

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

C. Verifying temporal specifications: Checking trace properties

A list of temporal events is maintained in order to check temporal specifications. The wrapper method RAC code now also contains calls to check temporal specifications, in addition to regular JML specification checking, at method control points (Listing 4). These methods feed the temporal events that have occurred to the runtime temporal state machine. After making all necessary transitions depending on the temporal events that have occurred, the machine performs temporal specification checks (Listing 8), when appropriate.

As per the temporalJML grammar (Table I) 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*.

The trace property in the temporal specification in Listing 5 is a disjunction of two the two basic trace properties \always($balance > 0$) and \eventually(swissType). Method `performTemporalChecks` (Listing 8) calls the *observer's* `update$temporalspec` if the machine is currently in a trace property checking state. The observer's `temporalspec$update` method calls update methods for updating the values of the the variables representing both basic trace properties. The methods check the value of `balance>0` and `swissType` and appropriately update the corresponding corresponding basic trace property variables.

D. Temporal State machine post final state checking

Like `jmlc`, temporaljmlc generates a wrapper method for each method in the class, with calls to specification checking methods before the call and after the (normal or exceptional) return to the renamed original method, [18]. However, for `main`, temporaljmlc additionally generates code in the *post-state* which it doesn't for any other method. At this point it is clear that the original (renamed) `main` has completed execution, with or without throwing an exception. The wrapper for `main` will now call the appropriate method to check the final status of the trace properties of temporal specifications.

Furthermore, the `main` wrapper calls a method to check if the temporal state machine representing each specification is in an accepting state; if not, an exception thrown, which indicates the reason that the specification was violated.

Listing 9: Bank account main driver –1

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

Listing 10: Bank account main driver –2

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

The generated code for the bank account class with the temporal specification written using temporalJML can be accessed from http://www.cs.ucf.edu/~fhussain/temporaljml/TemporalSpecBankAC.java.gen.

Sample runs of the BankAccount class

Consider the main driver in Listing 9 for the bank account class. The output produced (Figure 3) shows that the trace property for the temporal specification was violated. This occurred since the balance was negative even after the account was activated and the account was never changed to be of *swissType*.

```
$ temporaljmlrac 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)
$
```

Fig. 3: Running temporaljmlrac on Bank Account class driver-1 causes an exception

Now consider the main driver in Listing 10 for the bank account class. No output (in particular, no temporal specification violation exception) is produced by temporaljmlc. This behavior is expected because the trace property is not violated anymore since the flag `swissType` is set to `true` by the driver (Listing 10).

Post final state checking: To demonstrate the necessity of checking if the state machine's final state is an accepting state, consider the bank accont class with the original specification (Listing 5) modified to use an `\until` formula (Listing 11).

For the same driver as used before (Listing 10), which showed no output earlier, an exception is now thrown (Figure 4) which indicates that a required event did not occur. Note that it is not possible to throw this exception until after `main` has completed execution because only then can we be sure that the expected event (`\call suspendAC`) did not occur at all. Hence the requirement for the extra temporal specification check in `main's` post-state.

Listing 11: BankAC temporal specification

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

```
$jmlrac2 TemporalSpecBankAC
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLTemporalSpecificationError:
Temporal Specification at location
<File "TemporalSpecBankAC.java", line 7, character 24>
violated: Temporal Formula TF0 contains a
TemporalUntilExpression: Expecting one of the
following temporal events: [suspendAC]: at
TemporalSpecBankAC.checkTemporalMachineFinalState$instance
(TemporalSpecBankAC.java:281) at
TemporalSpecBankAC.checkTemporalMachineFinalState$instances
(TemporalSpecBankAC.java:290) at
TemporalSpecBankAC.main(TemporalSpecBankAC.java:1453)
$
```

Fig. 4: Running temporaljmlrac on the Bank Account class driver-2 with the specification in Listing 11 causes an exception

V. Translation of Temporal Specifications into State Machines

We present here our method of converting temporalJML temporal formulas (Table I) into finite state machines. Note that a `\before` temporal formula can be converted into an `\after`-`\until` temporal formula, a `\between` temporal formula can be written as an `\after`-`\unless` temporal formula and an `\atmost` temporal formula can be written as an `\after` temporal formula ([9, §4]). Therefore, it is sufficient to show how to translate `\unless` (Figure 5), `\until` (Figure 6) and `\after` (Figure 7, Figure 8, Figure 9, Figure 10) temporal specifications into finite automata.

States in which trace properties are checked (§IV-C) are marked with an asterisk. Temporal events, represented by e, $e1$ and $e2$ in these figures, may refer to either a single temporal event or a sequence of temporal events. Nodes which represent accepting states are shown with two concentric circles; should the machine be in a non-accepting state at the end of program execution, a temporal specification error is thrown (§IV-D).

For a simple `\unless` formula of the form "<TraceProp> `\unless` e", the state machine (Figure 5) starts with a trace property checking state. If the event occurs, a transition is made to *State A*. Both states are accepting states. The automaton representing the formula "<TraceProp> **\until** e" (Figure 6) is similar, with the only difference that the initial trace property checking state is non-accepting, because the semantics of an **\until** formula states that the given event must occur.

Fig. 5: <TraceProp> **\unless** e

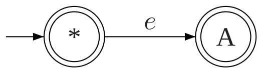
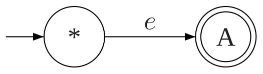


Fig. 6: <TraceProp> **\until** e

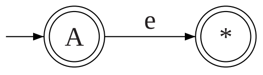


The case for **\after** specifications is more complicated because this is the only kind of temporal formula which may contain other temporal formulas inside it. From the temporalJML grammar (Table I) and the simplification rules for temporal formulas ([9, §4]) discussed earlier in this section, we divide specifications of the form "**\after** e ; <TempForm>" into the following four disjoint parts, based on what kind of temporal formula they contain:

- **\after** formulas containing only a trace property
- **\after** formulas containing another **\after** formula
- **\after** formulas containing an **\unless** formula
- **\after** formulas containing an **\until** formula

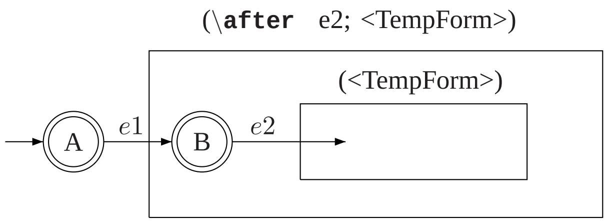
Consider the **\after** specification, "**\after** e; <TraceProp>", which contains only a trace property. Its automaton (Figure 7) consists of two accepting states. The automaton transitions to the trace property checking state when the event e occurs.

Fig. 7: (**\after** e; <TraceProp>)



Next, consider an **\after** specification such that its contained temporal formula is another **\after** formula. An example of the kind of automaton produced is shown in Figure 8. The machine starts in *State A*. The figure shows the automaton for the contained formula "**\after** $e2$; <TempForm>" in the outer rectangle. The inner rectangle represents the automaton for "<TempForm>". The temporal event $e2$ causes the machine to transition from *State B* to the initial state (not shown here) of the automaton for "<TempForm>". On occurrence of $e1$, the machine moves to *State B* (which is the initial state of the automaton representing "(**\after** $e2$; <TempForm>)"). Essentially, we are able to construct an automaton for a temporal specification of the form "(**\after** $e1$; (**\after** $e2$; <TempForm>))", by taking the automaton for the enclosed **\after** formula, namely (**\after** $e2$; <TempForm>), adding a new state, *State A* (which becomes the new initial state), and adding a transition from *State A* to the initial state of the original automaton (*State B*).

Fig. 8: (**\after** $e1$; (**\after** $e2$; <TempForm>))



Now, consider the temporal specification "**\after** $e1$; <TraceProp> **\unless** $e2$". This is translated into an automaton with three states: the initial *State A*, a trace property checking state (marked with an asterisk), and *State B* (Figure 9). The machine transitions from *State A* to the trace property checking state on occurrence of $e1$, and from this state to *State B* on occurrence of $e2$. What is worthy of note is that there is an ϵ-transition from *State B* to *State A*. Therefore, the trace property is checked each time $e1$ occurs, as long as $e2$ does not occur. All states here are accepting states.

Fig. 9: (**\after** $e1$; (<TraceProp> **\unless** $e2$))

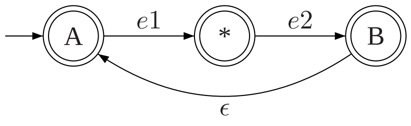
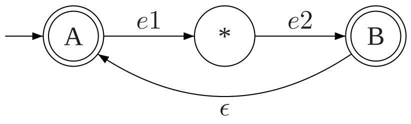


Fig. 10: (**\after** $e1$; (<TraceProp> **\until** $e2$))



The automaton for "**\after** $e1$; <TraceProp> **\until** $e2$" (Figure 10) is similar, except that the trace property checking state (marked with an asterisk) is a non-accepting state because the semantics of the specification "<TraceProp> **\until** $e2$" is that the trace property must hold as long as $e2$ doesn't occur, but that $e2$ *must* occur at some point during execution.

Thus, we have shown how an arbitrary temporalJML specification can be converted into a finite state machine.

VI. Notes on Semantics

Note from the grammar (Table I), that a trace property can only describe functional properties over occurrence patterns (**\always**, **\eventually** and **\never**). However, a temporal formula can describe complete temporal properties because they can be used to specify trace properties delimited by temporal events.

General notes about temporal formula subrules: A **\before** formula specification is equivalent to an **\always-\until** specification. Note that a **\before** formula is fundamentally different from an **\after** formula in that it cannot contain another top-level temporal formula, but only a temporal trace property, as specified in the temporalJML grammar (Table I). 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 none of the following temporal events should occur for the formula to be true, in which case the **\unless** formula holds if the underlying trace property holds.

Attempted specification of an internal state: Consider the following specification:

(**\after** **\call**(m); (**\before** **\normal**(m); **\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 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 `m` is called, so the success or failure of this temporal formula depends on whether P holds right at the point of the invocation of `m`. 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 temporalJML grammar (Table I), the **\atmost** formula describes the number of times an event can happen using a natural number. We follow the convention where natural numbers include zero, so an **\atmost** formula can be used to prohibit the occurrence of a temporal event (or a list of such events).

For a state-based semantics of the temporal logic extension on which temporalJML is based, see [9, §5.1].

VII. Related Work

The grammar for temporalJML is based on the temporal logic extension to JML suggested by Trentelman and Huisman [9]. They also propose translating a subset (viz. the formulas which express *safety properties*) of the new constructs of their temporal extension of JML back into standard JML expressions [9, §5.2]. Groslambert et. al. [20] propose a method for the verification of *liveness properties* in the temporal extension of JML in [9]. The JML Annotation Generator (JAG) [21] translates formulas expressed in the extension described in [9] into JML annotations. Although the translation mechanisms we use are similar, these approaches differs from our work because temporaljmlc translates the Java code annotated with temporal (and normal JML) specifications into regular Java.

Cheon and Perumandla propose a JML extension that allows the specification of sequences of method calls (*protocols*) [22]. They 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 inserting protocol checking into the CFG implementation [23]. This technique helps in specifying protocol properties of Java types. Their approach provides (and demands) separation of temporal properties (*protocols*) from functional behavior whereas our approach allows integration of the two using Bandera-style patterns to describe temporal behavior and trace properties to specify functional behavior.

Temporal Rover [24] is a verification tool that allows specifications written in an extension of LTL and Metric Temporal Logic (MTL) to be annotated to code written in C, C++, Java, Verilog and VHDL. This tool, developed by Time-Rover Software generates code from the written specifications which is linked to the application that its part of. However, it requires the programmer be well versed in formal temporal specification languages like LTL.

Java with Assertions (Jass) [25] is a Java extension which translates Java code annotated with specifications into pure Java and checks compliance with the specifications dynamically. It supports specification of *trace assertions* that describe the ordering of method calls. However, as pointed out in [9, §1], Jass trace assertions cannot be integrated with functional specifications.

The Bandera Specification Language [4] is a "source-level, model-checker independent language" that allows writing temporal specifications by avoiding logics like CTL and LTL. It is different from our approach in that its based on model checking, whereas we follow primarily a design by contract approach [1]. We consider temporal property specifications in the above mentioned logics as overly formal for most programmers and want to also avoid the state-explosion problem that often arises when large applications are to be represented as a mathematical structure which is required for model checking. Moreover, BSL does not allow specifications which describe properties of exceptions, as noted in [9, §1].

VIII. Limitations and Future Work

Currently, 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.

As further work, we plan to extend the tool to handle temporal specifications written in interfaces and provide support for temporal specification inheritance. Future work may also involve allowing temporal specifications of concurrent Java programs.

IX. Conclusion

Our contribution is the addition of temporal specification capability using Bandera-style patterns to JML, which we call 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, because temporalJML allows the integration of temporal and functional specifications.

Acknowledgement

The authors would like to thank Marieke Huisman for clarifications regarding the semantics of the JML temporal extension proposed in [9] and the anonymous reviewers for very helpful comments. The work of both the authors was supported in part by NSF grants CNS 08-08913 and CCF-0916350.

References

[1] B. Meyer, *Object-oriented Software Construction*. Prentice Hall, 1988.

[2] 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.*, vol. 8, no. 2, pp. 244–263, 1986.

[3] G. J. Holzmann, *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&amp;path=ASIN/0321228626

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby, "Expressing checkable properties of dynamic systems: The Bandera Specification Language," *STTT*, vol. 4, no. 1, pp. 34–56, 2002.

[5] Robby, "Bandera Specification Language: A specification language for software model checking," 2000, master's thesis, Kansas State University. [Online]. Available: citeseer.ist.psu.edu/498244.html

[6] J. Hatcliff and M. 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*. London, UK: Springer-Verlag, 2001, pp. 39–58.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting finite-state models from java source code," in *International Conference on Software Engineering*, 2000, pp. 439–448. [Online]. Available: citeseer.ist.psu.edu/corbett00bandera.html

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," Kansas State University, University of Massachusetts, University of Hawai'i, Tech. Rep. UM-CS-1998-035, , 1998. [Online]. Available: citeseer.ist.psu.edu/dwyer99patterns.html

[9] K. Trentelman and M. Huisman, "Extending JML specifications with temporal logic," in *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*. London, UK: Springer-Verlag, 2002, pp. 334–348.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," in *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*. New York, NY, USA: ACM, 1998, pp. 7–15.

[11] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," in *FMCO*, 2005, pp. 342–363.

[12] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 3, pp. 212–232, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1007/s10009-004-0167-4

[13] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," Iowa State University, Department of Computer Science, Tech. Rep. 98-06-rev29, Jan. 2006, also *ACM SIGSOFT Software Engineering Notes*, 31(3):1-38, March 2006. [Online]. Available: ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.pdf

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

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

[16] F. Hussain, "Enhancing a behavioral interface specification language with temporal logic features," Master's thesis, Department of Computer Science, Iowa State University, Ames, IA, Apr. 2009. [Online]. Available: http://archives.cs.iastate.edu/documents/disk0/00/00/06/00/index.html

[17] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers, "MultiJava: Design rationale, compiler implementation, and applications," vol. 28, no. 3, pp. 517–575, May 2006. [Online]. Available: ftp://ftp.cs.iastate.edu/pub/techreports/TR04-01/TR.pdf

[18] Y. Cheon, "A runtime assertion checker for the java modeling language," Ph.D. dissertation, Apr. 2003, technical Report 03-09, Department of Computer Science, Iowa State University.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[20] F. Bellegarde, J. Groslambert, M. Huisman, O. Kouchnarenko, and J. Julliand, "Verification of liveness properties with JML," INRIA, Tech. Rep. RR-5331, 2004.

[21] A. Giorgetti and J. Groslambert, "JAG: JML Annotation Generation for verifying temporal properties," in *FASE'2006, Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 3922. Vienna, Austria: Springer, Mar. 2006, pp. 373–376. [Online]. Available: http://dx.doi.org/10.1007/11693017_27

[22] Y. Cheon and A. Perumandla, "Specifying and checking method call sequences of Java programs," *Software Quality Journal*, vol. 15, no. 1, pp. 7–25, Mar. 2007.

[23] Y. 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)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 475–482.

[24] D. Drusinsky, "The Temporal Rover and the ATG rover," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK: Springer-Verlag, 2000, pp. 323–330.

[25] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass – Java with Assertions," in *Electronic Notes in Computer Science*, K. Havelund and G. R. su, Eds., vol. 55(2). Elsevier Science BV, 2001.