2014-01

Computer-aided Discovery of Formal Specification Behavioral Requirements and Requirement to Implementation Mappings

Drusinsky, Doron

Monterey, California. Naval Postgraduate School

NPS-CS-14-001

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

COMPUTER-AIDED DISCOVERY OF FORMAL SPECIFICATION BEHAVIORAL REQUIREMENTS AND REQUIREMENT TO IMPLEMENTATION MAPPINGS

by

Doron Drusinsky

January 2014

Approved for public release; distribution is unlimited

Prepared for: Office of Naval Research (ONR)
One Liberty Center, 875 North Randolph Street, Suite 1425
Arlington, VA 22203-1995

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE ***(DD-MM-YYYY)*** 30/1/2014	**2. REPORT TYPE** Technical Report	**3. DATES COVERED** ***(From-To)*** April 2013 – Feb 2014

4. TITLE AND SUBTITLE Computer-aided Discovery of Formal Specification Behavioral Requirements and Requirement to Implementation Mappings	**5a. CONTRACT NUMBER** N0001413AF00002, N0001414AF00002
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Doron Drusinsky	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) AND ADDRESS(ES) Naval Postgraduate School 1411 Cunningham Road Monterey, CA 93943	**8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-14-001
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Code 822, Office of Naval Research (ONR) - One Liberty Center, 875 North Randolph Street, Suite 1425 , Arlington, VA	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

14. ABSTRACT

This paper presents two computer-aided techniques for discovering formal specification behavioral requirements and for mapping components and methods within an implementation to their driving requirements. The first technique uses a system reference model (SRM) abstraction and a set of existing formal specifications to discover implementation components that are not well covered by the formal specification set. This technique also provides a mapping between requirements and code segments driven by those requirements. The second technique uses a bounded constraint solver to match a set of tests with a generic formal specification taken from a small library.

15. SUBJECT TERMS
Requirements, Specification, Verification, UML, bounded constraint solving

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Doron Drusinsky
Unclassified	Unclassified	Unclassified	UU	24	**19b. TELEPHONE NUMBER** ***(include area code)*** 831 656 2168

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Ronald A. Route President	Douglas A. Hensler Provost

The report entitled "*Computer-aided Discovery of Formal Specification Behavioral Requirements and Requirement to Implementation Mappings*" was prepared for funded by Office of Naval Research (ONR), One Liberty Center, 875 North Randolph Street, Suite 1425, Arlington, VA.

Further distribution of all or part of this report is authorized.

This report was prepared by:

Doron Drusinsky
Associate Professor
Computer Science Department

Reviewed by:	**Released by:**
Peter J. Denning, Chairman Computer Science Department	Jeffrey D. Paduan Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This paper presents two computer-aided techniques for discovering formal specification behavioral requirements and for mapping components and methods within an implementation to their driving requirements. The first technique is an informal technique while the second technique is formal. The first technique uses a system reference model (SRM) abstraction and a set of existing formal specifications to discover implementation components that are not well covered by the formal specification set. This technique also provides a mapping between requirements and code segments driven by those requirements. The second technique uses a bounded constraint solver to match a set of tests with a generic formal specification taken from a small library

THIS PAGE INTENTIONALLY LEFT BLANK

Computer-aided Discovery of Formal Specification Behavioral Requirements and Requirement to Implementation Mappings[1]

Doron Drusinsky
Department of Computer Science
Naval Postgraduate School
1411 Cunningham Road, Monterey, CA 93943, USA
ddrusins@nps.edu

[1] This research was funded by a grant numbers N0001413AF00002 and N0001414AF00002 from the Office of Naval Research. The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

1 INTRODUCTION

It is widely claimed that formal methods help improve the quality of software [CW], [ELC]. Formal methods have received considerable academic attention during the last three decades, as reflected by the many technical papers published in the open literature. For example, IEEE Software (Sept. 1990), IEEE Computer (Sept. 1990), and IEEE Transactions on Software Engineering (Sept. 1990, May 1997, Aug. 2000) all have published special issues on formal methods. Formal Specifications (FS) are mathematically based techniques for assisting with the implementation and assurance of systems and software. Numerous FS languages have been proposed over the past four decades, primary of which are temporal logics [CE], and diagrammatic languages, such as statechart assertions [D1, D2], the FS formalism used by this paper. There are two primary categories of temporal logics, linear time and branching time, with Propositional Linear-time Temporal Logic (PLTL) being the better known linear time FS language, and Computational Tree Logic (CTL and CTL*) being the better known variants of branching time logics.

Run-time Verification (RV) of formal specification assertions (RV), also known as Run-time Execution Monitoring (REM), is a class of methods for monitoring the sequencing and temporal behavior of an underlying application and comparing it to the correct behavior as specified by a formal specification [D5, HR]. The lightweight technique described in this paper is an RV-based technique.

The SElf-awarE Computing (SEEC) Model was named one of ten “World Changing Ideas” by Scientific American in their December 2011 issue [SEEC]. SEEC is designed to address the challenge of programming modern and future computer systems that must meet conflicting goals (e.g. high performance with low energy consumption). SEEC meets this challenge by allowing developers to collaboratively create adaptive systems that understand user’s goals and requirements and takes appropriate actions in the face of unexpected events such as variations in application workload or a malicious attack. In the SEEC model programmers expose application, while system programmers and hardware developers expose actions. SEEC’s runtime decision engine monitors goals and progress and then uses a novel combination of control theory and machine learning to schedule actions that meet goals optimally. One outcome of this paper is a technique for mapping requirements and goals with implementation code, a useful step in making an application a SEEC application.

SAT-solving is defined as the process of deciding, given a set of propositional formulas in Conjunctive Normal Form (CNF), whether there is an assignment to the variables in the formula such that the formula is satisfied (true). While the core problem is a classical NP-complete problem [HU], recent advances have resulted in efficient SAT solvers capable of solving formulas with hundreds of thousands of variables and millions of clauses [MZ].

Bounded constraint solving [K] uses SAT-solving to discover an instance assignment to a set of *n*-ary relations (unary relations are sets, functions are binary relations) that satisfy a set of First Order Logic (FOL) constraints.

This paper presents two computer-aided techniques for discovering FS’s. The first technique is light-weight and informal; it uses a FS repository, a System Reference Model (SRM), and a code coverage tool to discover code segments that are not covered by FS’s. The second technique is a heavyweight formal method that uses bounded constraint solving to match a set of test scenarios with a generic FS taken from a FS library.

The rest of the paper is organized as follows. Chapter 2 provides an overview of formal speciations, a FS library, FS validation, and bounded constraint solving. Section 3 describes the light weight requirement discovery technique, and section 4 describes the formal FS discovery technique.

2 PRELIMINARIES

2.1 FORMAL SPECIFICATIONS USING STATECHART ASSERTIONS

In section 3 we will be using the following Traffic Light Controller (TLC) example. The TLC controls lights at a junction of two streets, El Camino Real and B Street, in Silicon Valley. Sensors are positioned under both streets. The El Camino Real sensor dispatches a *newCar* or *newTruck* event to the TLC, when a new car or a new truck is detected. The system should contain a camera positioned in each direction of the El Camino Real. The TLC must conform to the following Natural Language (NL) requirements governing the behavior of the El Camino Real lights:

R1. *If the average weight of vehicles on record on El-Camino Real is lower than 2000 kg, then the El Camino Real lights should not remain green for more than one minute*.

R2: *If four or more cars are waiting while lights are red in the El Camino Real direction, then lights must turn green within at most 15 seconds of the arrival time of the fourth car*.

R3: *When lights turn red in the El Camino Real direction, they must not turn green for at least 15* *seconds*.

R4: *When red or green in eiterr direction, , then upon reset lights must turn yellow within five seconds*.

R5. If a car is in going through the junction at a speed of 1 m/sec or less then lights in both directions must turn red for at least 30 seconds.

Figure 1 depicts statechart assertions for requirement R3. The syntax, semantics, and application of statechart assertions to the Validation and Verification (V&V) of mission critical systems are elaborated in [D1, D2, DMTS]. The following is a short overview.

The statechart assertion of Fig. 1 starts of in the Init state where it waits for an event; UML events are annotated as *event[optional-condition]/optional-action*. When event *lightsTurnedRed* occurs, the assertion transitions to state *Red*, where it executes a *timer.restart()* Java action, restarting a 15 second timer. It then waits for either the *lightsTurnedGreen* or *timeoutFire* events, the later being the 15-second timer expiration event. The statechart-assertion continues hopping from state to state given the input events. Whenever it reaches the *Error* state its sets the built-in Boolean *bSuccess* flag to *false* (it is *true* by default), thereby announcing that the assertion has detected a violation of requirement R3.

2.2 VALIDATION

Validation is a testing process for assuring that the statechart-assertion is a good representative of its corresponding NL requirement. This is done by examining the assertion's response to hand crafted test scenarios and comparing the assertions actual Boolean response to the expected response - as prescribed by the NL requirement. An additional important aspect of validation is to examine, understand, and expose the true intention of the requirement and resolve potential ambiguities. For example, a validation test for requirement R5 exposes the fact that the period in which the speed must be lower than the specified threshold is not specified in R5. In other words, according to the NL specification it is OK to turn all lights red if the car speed drops below 1m/sec only for a very short time. Hence, we modify the requirement as follows.

R5a: *If a car is in going through the junction at a speed of 1 m/sec or less for at least 10 seconds then lights in both directions must turn red, within 5 seconds thereafter, for at least 30 seconds.*

Figure 2a is a statechart assertion for requirement R5a, and Figures 2b, 2c, and 2d contain timeline diagram depictions of three validation test scenarios for this assertion. The *setSpeed*() and *getSpeed*() method are used by test driver to set speed values, and by the assertion to obtain them. During validation, the JUnit[2] driver plays God and sets speed values, whereas during testing speed values are sampled from log file generated earlier, when executing the System Under Test (SUT) [D2]. The statechart assertion of Figure 2a uses a built-in 1Hz *tick* event, and two built in timers, one (named *timerT2*) for 2 seconds, and the other (named *timerT3*) for 30 seconds. Each timer is restarted using a *restart*() action; it fires the *timeoutFire* event when its time limit expires.

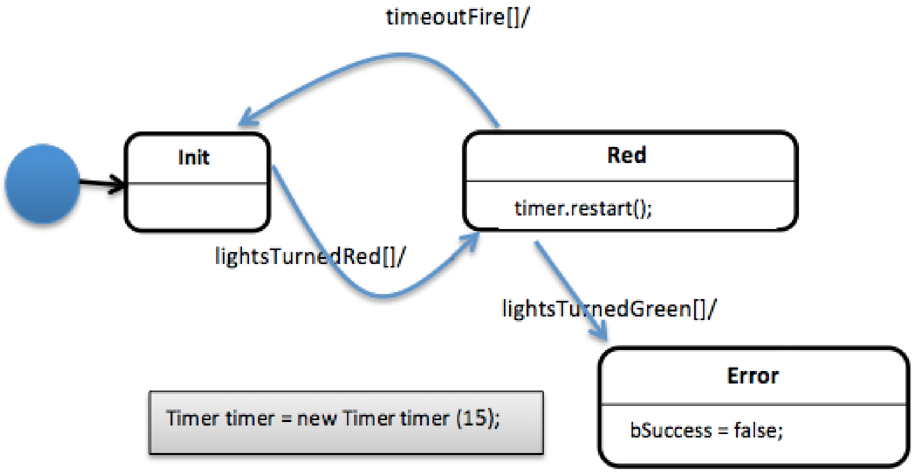
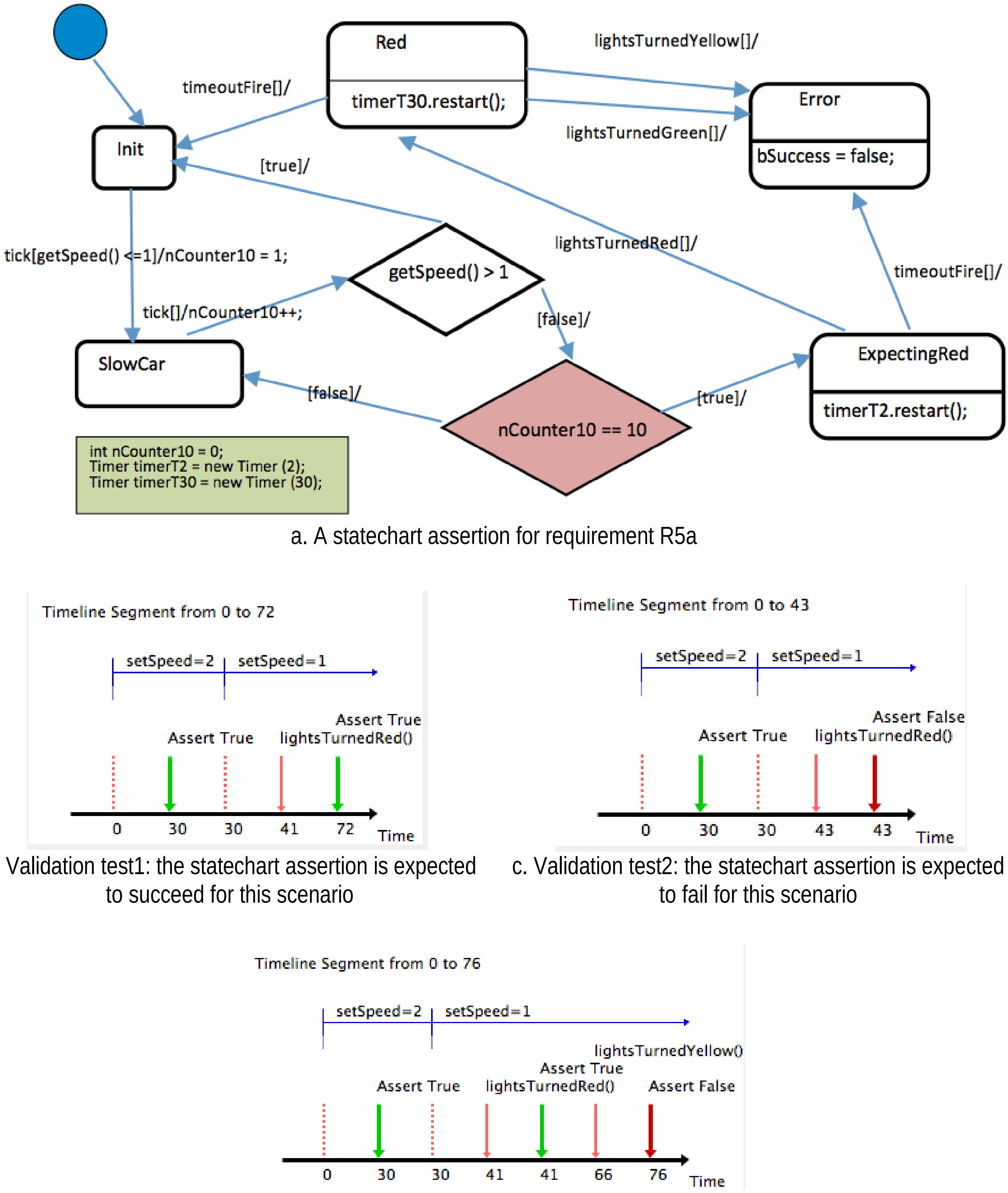


Figure 1. A statechart assertions for TLC requirement R3.

[2] JUnit is a well-known standard Java testing framework; see www.junit.org.



a. A statechart assertion for requirement R5a

b. Validation test1: the statechart assertion is expected to succeed for this scenario

c. Validation test2: the statechart assertion is expected to fail for this scenario

d. Validation test3: the statechart assertion is expected to fail for this scenario

Figure 2. A statechart assertion for requirement R5a and three corresponding validation tests

As described in [D2] the assertion repository tool contains assertions, their generated code, and validation test suite (one that is also used in a later stage for verification). Our lightweight technique assumes that assertions have been well validated, preferably using validation patterns, as suggested in [D2, DMTS].

2.3 A GENERIC FORMAL SPECIFICATION LIBRARY

A generic statechart assertion library [D2] is a collection of statechart assertions with the following generic parameters:

- Event names: generic assertions use names such as P, Q, and R; These generic event names are mapped to concrete system event names when the assertion is instantiated (e.g., when used for verification).
- Time bounds, such as the 15 second limit in Figure 1 (per requirement R3), are represented by generic bounds, such a T.
- Count values, such as the value four in requirement R2, are represented using generic values, such as N.

Hence, the generic statechart assertion of Figure 3 represents the NL requirement *Flag whenever event Q occurs more than N times between events P and R.*

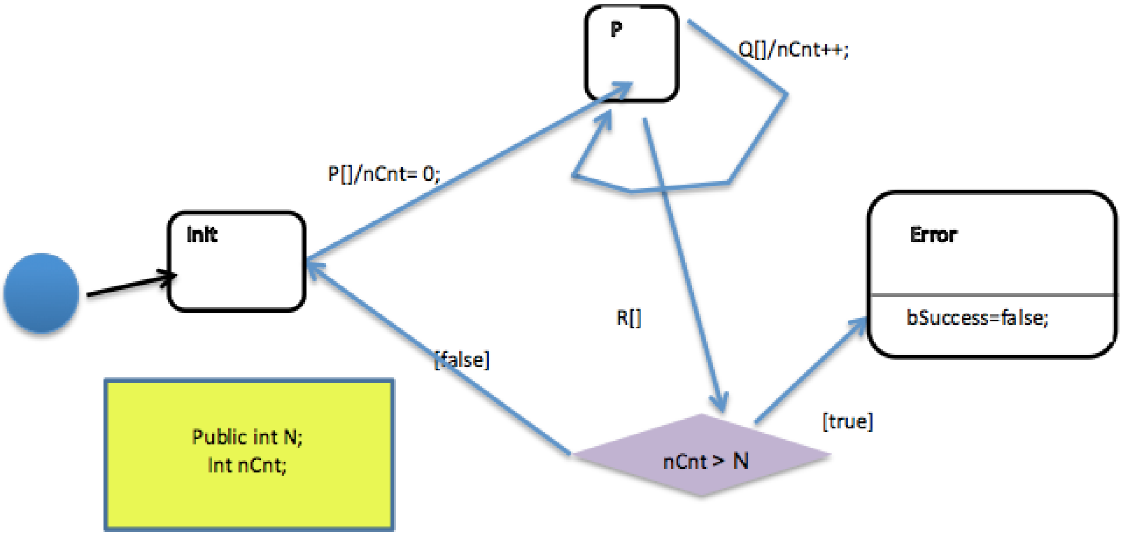


Figure 3. A Generic statechart assertion

A. 2.4 BOUNDED CONSTRAINT SOLVING

The MIT-Kodkod [K] is an efficient SAT-based constraint solver for First Order Logic (FOL) with relations, transitive closure, bit-vector arithmetic, and partial models. It enables the analysis of both satisfiable and unsatisfiable problems, while providing a finite model finder for the former and a minimal unsatisfiable core extractor for the latter. MIT-Kodkod is used in a wide range of applications, including code checking, test-case generation, declarative execution, declarative configuration, and lightweight analysis of Alloy, UML, and Isabelle/HOL.

The MIT-Kodkod has a Java API on its front-end, and uses one of a plurality of possible SAT-solvers on the back-end. The Kodkod definition of a constraint set uses four primary Java objects named *Relation*, *Expression*, *Formula*, *and Atoms*. Atoms define the (bounded) domain of discourse. Relation and Expression objects are sets of tuples taken from a collection of predefined atoms; Relation objects are explicitly bound to some finite collection whereas Expression objects are calculated from other Relation and Expression objects. Formula objects are FOL formula written in Java, whose domain of discourse is the abovementioned collection of Relation and Expression objects.

The top level MIT-Kodkod Formula is a set of logical constraints that is translated to a CNF and handed of to a SAT solver. The SAT solver then searches for an instance solution to the constraint set. Clearly, the absence of a solution within the bounded region does not necessarily imply that an unbounded version of the problem has no solution.

In our Kodkod representation of a FS we will be using unary Relations - which amount to simple sets, and binary relations, which correspond to arrays. For example, the equivalent of a *preset-state* state-machine variable is a binary relation *stateAtTime*, $stateAtTime \subseteq state \times time$. We use the conventional programming notation $stateAtTime[t]$ to denote the state(s) at time t. The Kodkod approach for calculating for $stateAtTime[t]$ is stateAtTime.join(t).

3 AN INFORMAL TECHNIQUE FOR DETECTING CODE SEGMENTS WITH INSUFFICIENT SPECIFICATION

3.1 CONSTRUCTING THE SRM

An SRM is a collection of UML documents and models. It consists of executable and non-executable components. For example, a technical-report publishing system contains case diagrams (UC's), activity diagrams (AD's), sequence diagrams (SD's), and possibly other architectural-level UML diagrams in its non-executable part. The executable SRM consists of class diagrams (CD's), an object model, statechart assertion diagrams, and a JUnit validation test suite.

Figure 4 shows the class diagram of the executable SRM for the TLC example. It consists of the repository of statechart assertions (diagrams and resulting auto-generated Java code) discussed in previous sections, a domain model (DM), and a JUnit validation test suite driver – derived from the test suite discussed in the previous section.

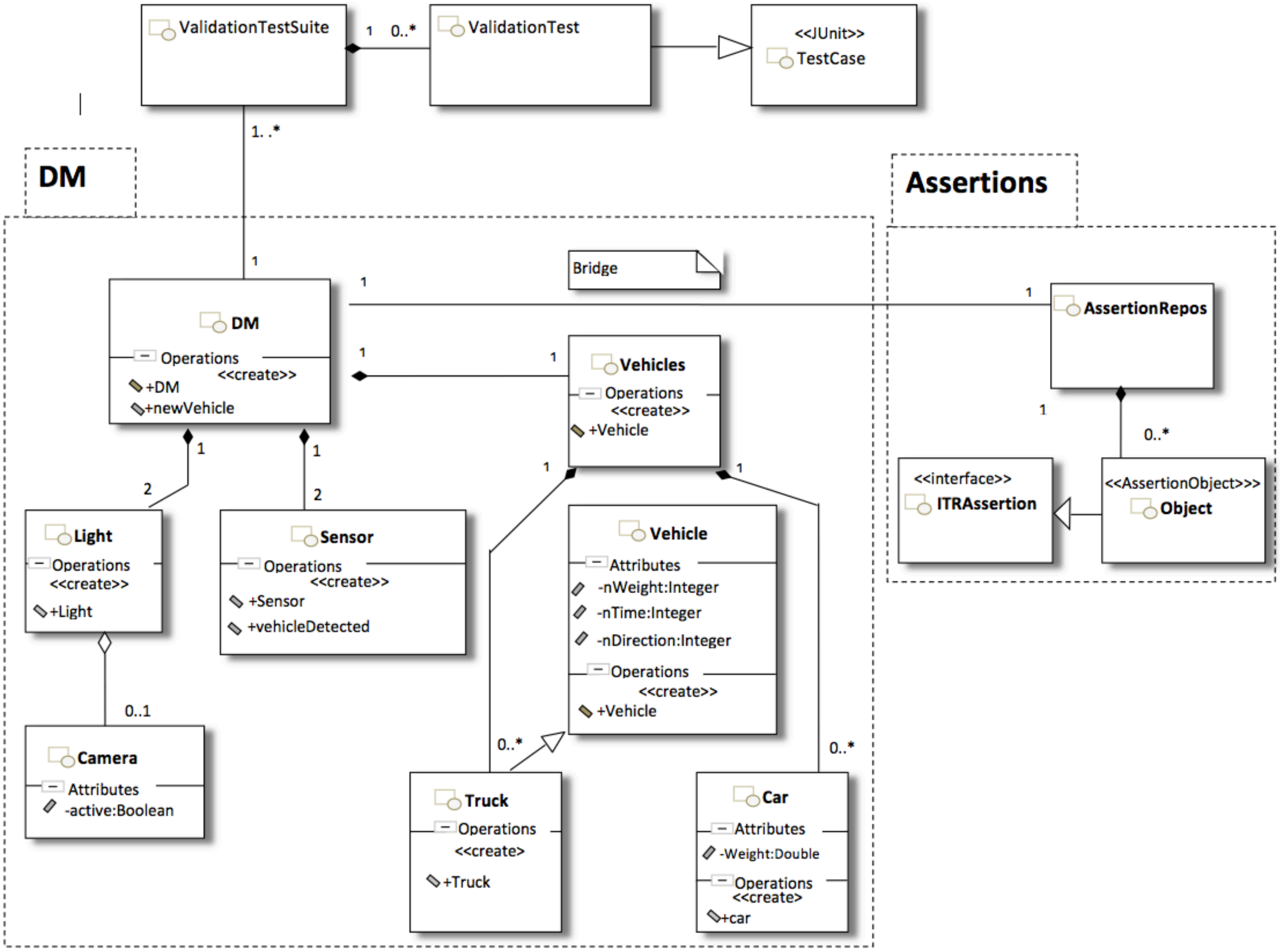


Figure 4. The UML class diagram of an executable SRM for the TLC example.

The assertion repository part of the executable SRM contains Java object implementation code for all statechart assertions, as well as code for propositional assertions, which are outside the scope of this paper (see [D5] for further information). In the case of the executable SRM for the TLC example, the assertion repository consists of code for the statechart assertions of NL requirements R1 through R4, and R5a The assertion repository is implemented as a wrapper that encapsulates a collection of Java-object instance implementations of statechart assertions; each statechart assertion is implemented as a Java (or C++) class [D1]. The assertion repository dispatches events received via the bridge to all assertions in the collection and announces failure when one or more assertions fail.

The DM part of the executable SRM is developed according to the specification and marketing documents. Hence, for example, the *Vehicles*, *Car*, and *Truck* classes in Figure 4 exist as specified by the nonfunctional requirement R3 of the TLC example.

Finally, let's consider the bridge connecting the DM to the assertions. The bridge is responsible for passing, from the executing DM to the assertions, all events and data artifacts the assertions assert about, as well as timing information (the *incrTime* method calls in the validation test driver). For example, when a *Car* object fires its constructor, a *newCar* event notification is sent via the bridge to the assertion repository. We have implemented two types of bridges: (i) hard-coded, and (ii) Aspect-oriented (e.g., using *AspectJ*). A hard-coded bridge depends on hard-coded source code instrumentation that sends all events from the DM to the assertion repository. The Aspect-oriented approach, on the other hand, as described in [DMOS], does not require such verbatim code instrumentation.

3.2 DRIVING VALIDATION TESTS VIA THE SRM

Using the executable SRM, validation test artifacts (events and data values) drive the assertion repository *via the DM*, as done by the JUnit test of Listing 1a. This is in contrast with the *direct* validation test of Listing 1b, which is of the kind described in the previous section, i.e., tests that exercise assertions directly. Hence for example, instead of a validation test issuing a *newCar* event to a particular assertion or to the entire assertion repository, as in the case of direct validation, the DM-based test driver operates via the DM: it creates a scenario in which Sensor objects sense a *vehicleDetected* event, thereby inducing the construction of a new Car object, which in turn sends a *newCar* event to the assertions.

a. A v*ia-DM validation* test; the assertion for requirement R2 is expected to fail, therefore the entire assertion repository is expected to report a failure.

```
public void testMe() {
  EClight.lightsTurnedRed();
  // argument is weight
  // 1000 to 4000 Kg. is a car
  sensor.vehicleDetected(2200);
  sensor.vehicleDetected(2800);
  sensor.vehicleDetected(1450);
  sensor.vehicleDetected(3700);
  timer.incrTime(16);
  EClight.lightsTurnedGreen();
  assertFalse(dM.isSuccess());
}
```

b. A *direct validation* of assertion for requirement R2

```
public void testMe() {
  assertion.lightsTurnedRed ();

  assertion.newCar();
  assertion.newCar();
  assertion.newCar();
  assertion.newCar();
  assertion.incrTime(16);
  assertion.lightsTurnedGreen();

assertFalse(assertion.isSuccess());
 }
```

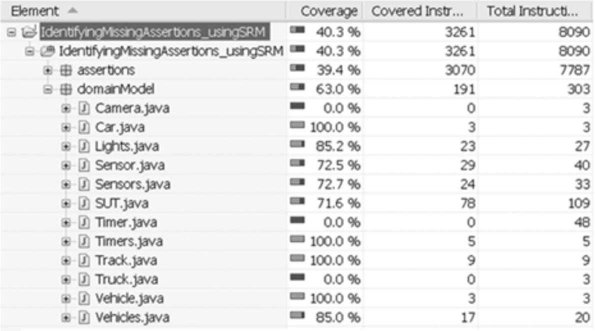
Listing 1. JUnit via-DM validation test vs. direct-validation test for the TLC.

Note that the DM in Figure 4 is not an implementation model. Although it contains sufficient detail to enable via-DM testing as in Listing 1a, it does not contain a controller that makes specific deterministic decisions, such as when the lights actually change status, as is evident from Listing 1a, where the JUnit test case is the driver of those light-color values.

Via-DM validation testing helps ensure that the DM is not under-modeled. For example, the via-DM validation test in Listing 1a refers to a timer object to advance the clock, because via-DM testing implies that the DM is responsible for generating all artifacts witnessed by the assertions, including time; the DM in Figure 4 and its corresponding Java implementation, however, contains no such Timer class, an obvious underrepresentation.

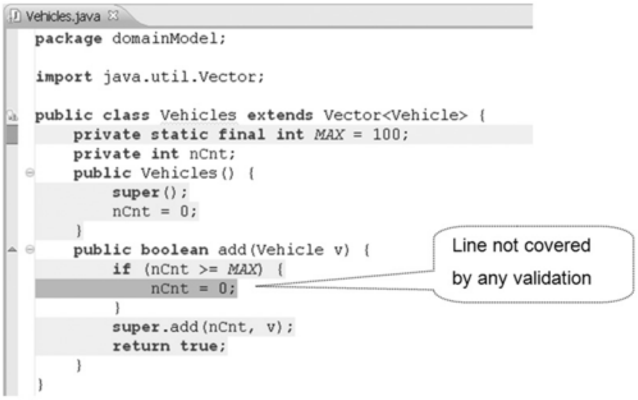
3.3 USING CODE COVERAGE TO DETECT UNCOVERED CODE SEGMENTS

When validation tests are all via-DM validation tests, we perform the last phase of our light-weight FS discovery technique by running test coverage analysis. Test coverage provides information about the extent to which the test suite exercises source code classes, methods, or individual lines. A test coverage report for the TLC model example is illustrated in Figure 5.[3] The coverage report in Figure 5a indicates that the *Truck* and *Camera* classes have 0% coverage; i.e., no validation test touches those classes. Recall, however, that every FS assertion must be validated before we reach this stage. Consequently, we can deduce that there are no FS assertions about the behavior of the TLC that pertain to the behavior of the *Camera* class or the behavior of the *Truck* class, and we should therefore consider adding requirements to that effect.



Element	Coverage	Covered Instr...	Total Instructi...
IdentifyingMissingAssertions_usingSRM	40.3 %	3261	8090
IdentifyingMissingAssertions_usingSRM	40.3 %	3261	8090
assertions	39.4 %	3070	7787
domainModel	63.0 %	191	303
Camera.java	0.0 %	0	3
Car.java	100.0 %	3	3
Lights.java	85.2 %	23	27
Sensor.java	72.5 %	29	40
Sensors.java	72.7 %	24	33
SUT.java	71.6 %	78	109
Timer.java	0.0 %	0	48
Timers.java	100.0 %	5	5
Track.java	100.0 %	9	9
Truck.java	0.0 %	0	3
Vehicle.java	100.0 %	3	3
Vehicles.java	85.0 %	17	20

a. The DM coverage report after executing the via-DM validation suite.



```
package domainModel;

import java.util.Vector;

public class Vehicles extends Vector<Vehicle> {
    private static final int MAX = 100;
    private int nCnt;
    public Vehicles() {
        super();
        nCnt = 0;
    }
    public boolean add(Vehicle v) {
        if (nCnt >= MAX) {
            nCnt = 0;
        }
        super.add(nCnt, v);
        return true;
    }
}
```

b. Coverage of the Vehicles collection. Clearly there is no validation test and therefore no assertion that pertains to buffer overflow

Similarly, the coverage report in Figure 5b indicates that there is no validation test, i.e., no assertion that is concerned with the behavior of the *Vehicles* collection when more than 100 vehicles are detected.

Figure 5. Source code coverage discovery of missing assertions.

3.4 DISCOVERING REQUIREMENT TO IMPLEMENTATION MAPPING

Discovering the mapping between requirement and the implementation is a process similar to the abovementioned process, except that the implementation now substitutes the DM. Two additional changes are made to the process:

1. Instead of using the via-DM (now via-implementation) test suite to exercise all assertions in the assertion repository, we exercise a single assertion at a time, i.e., we exercise a single requirement at a time.
2. For each such assertion and its associated validation test-suite, we use the coverage tool to detect implementation code snippets that have a high coverage score. Such high coverage means the requirement assertion is tightly coupled with that code location.

4 USING BOUNDED CONSTRAINT SOLVING TO DISCOVER FORMAL SPECIFICATIONS

4.1 A FOL REPRESENTATION OF A STATECHART ASSERTION FS

The bounded Kodkod representation of a statechart assertion FS's *structure*, contains the following basic declarations and bounds, described for a concrete version of the statechart-assertion of Figure 3:

- Atoms, are strings that represent artifacts of the FS, including its state names (e.g., "Init"), transition names (e.g., "Tr1", "Tr2"), and event names (e.g., "P"). In addition, the atom set contains two atoms, named nCntGtN and nCntNotGtN, for the condition nCnt>N being true or false, respectively. In addition, the atom set contains two atoms that represent the actions nCnt++ and timer.restart (for generic assertions with time constraints). Finally, the numbers 0 through MAX_INT are also atoms, with MAX_INT being a customizable number such as 100. These numbers are used for the timer and nCnt counter.
- A *timeDomain* Expression (i.e., a set), which contains integer values representing time. Similarly, *countDomain* represent the values the FS's variable (nCnt) can hold.

[3] Coverage in this example was done using the Emma test coverage tool (www.eclemma.com).

- A *restartTimes* Relation, that represents time slots in which a FS timer restarts (note that the assertion of Figure 3 has no timer, but an assertion for R1 would have one). Separate constraints will force *t* to be in *restartTimes* if and only if *stateAtTime*[*t*] is a state with an action timer.restart.
- A *states* unary relation (i.e., a set) declared as in: states = Relation.unary("states"). This relation is then bound to of the states of the FS, as in:

  bounds.boundExactly(states, tupleFactory.setOf("Init","P", "Error")).

  Note that the relation is bounded *exactly* to the set of state atoms, so to prevent the constraint solver from finding a solution that assumes a FS with fewer states than the abovementioned set.
- A *transitionIDs* unary relation bound (exactly) to: " Tr1", "Tr2", "Tr3", and "Tr4". Note that the diamond decision polygon is reduced to two transitions: P$\rightarrow_{R[nCnt<=N]}$Init and P$\rightarrow_{R[nCnt>N]}$Error.
- A *transitions* ternary relation: *transitions* ⊆ *states* x *transitionIDs* x *states*, such as <"Init", "Tr1", "stP">
- A *conditions* unary relation bound to "nCntGtN" , "nCntNotGtN".
- A *localEvents* unary relation bound to "P","R","Q". This set represents the events labeling the transitions of the statechart assertion FS.
- A special event named *stutter*, that represents the case that this FS cannot respond to the input event when in its present-state, and therefore stutters in its present state.
- A special event named *timeoutFire*, that represents the case that this timer fires. Recall that statechart-assertion semantics are such that when it is in a state with an outgoing transition labeled *timeoutFire* (e.g. *Red* in Figure 1), then this transition is traversed when the timer fires. This semantics is enforced below.
- Two action relations: *beforeActions* - for actions that execute before the associated transition fires (as with nCnt++ action, binding the operation to "incrNCnt"), and *afterActions* one that executes afterwards (as with the timer.restart action, binding the operation to timerRestart").
- A *firstState* and *flaggedState* relations that are bound to the FS's *initial* and *Error* states, respectively.
- Binary relations *eventToTransition* and *conditionToTransition* that map *localEvents* and *conditions*, respectively, to *transitionIDs*.
- Binary relations *beforeActionToTransition* and *afterActionToTransition* that map *beforeActions* and *afterActions*, respectively, to *transitionIDs*. For example "incrNCnt" is mapped to the transition whose ID is "Tr2" (i.e., the Figure 3 transition P→P).
- A *stateAtTime* binary relation that maps *timeDomain* to *states*.
- A *countAtTime* binary relation that maps *timeDomain* to the integers.
- A *localEventAtTime* binary relation that maps *timeDomain* to *localEvents* ∪ {*stutter*} ∪ {*timeoutFire*}.

The structural part also contains a Formula named *declarations* for constraining the following:

- *stateAtTime* and *localEventAtTime* are *functions*, rather than generic relations. In Kodkod this constraint is written as the Formula: stateAtTime.function(timeDomain, states).
- *localEventAtTime* is a *function* from *timeDomain* to *localEvents.*
- *restartTimes* is a subset of *timeDomain.* In Kodkod it is written as the Formula: restartTimes.in(timeDomain).

The semantics of an FS are enforced as follows. We use a Formula denoted *existsCompuation* that forces the FS to have a legal computation. Listing 2 contains the corresponding Kodkod predicate method. Note that the *existsCompuation* method returns a Formula; this return value is subsequently conjuncted with all other constraints and passed along to the constraint solver. Clearly, *existsCompuation* is but a conjunction of single step advances of the FS state machine.

```
Formula existsComputationFromTo(int startIndex, int endIndex) {
  if (startIndex >= endIndex) return Formula.TRUE;
  Formula pred = Formula.TRUE;
  int prev_i = startIndex;
  for (I = 0; i < endIndex; i++) {
                  Formula pred_i = existsComputationOneStep(prev_i,  i);
                  pred = pred.and(pred_i);
                  prev_i = i;
  }
  return pred;
```

```
          }
```

Listing 2. The predicate method that constrains a statechart assertion FS to legal behavior according to its semantics

The *existsComputationOneStep* single step predicate method forces the FS to have a computation from time-step *t*-1 to time-step *t*. It does so by enforcing constraints on all the *at time* relations (e.g., *stateAtTime*) that enable only legal transitions or stuttering. Hence, it is a conjunction of the following predicates:

- *stutterConstraint*[*t*]: which assures the FS stutters at time *t* (i.e., doesn't change states and does not perform any action) when the local-event is *stutter*.
- *nonStutterConstraint*[*t*]: which assures that if the FS does not stutter at time *t*, then it makes a state change based on one of the events annotating its transitions.
- *forcePossibleNextState*[*t*]: which assures that the FS advances at time *t* to the next state according to given *localEventAtTime*[*t*], *eventToTransition*[*t*] and *conditionAtTime*[*t*]. Conditions constrained in a straightforward manner, such as: *conditionAtTime*[*t*] for the FS of Figure 3 maps time *t* to `nCntGtN` if and only if *countAtTime*[*t*] > N; *timeoutFire* transitions are enabled for every time slot *t* in *restartTimes*. Constraints for the time and counting parameters, such N of figure 3, are discussed in the sequel.

  In addition, *existsComputationOneStep* assures the FS is not lazy, i.e., it stutters only if there is no alternative, i.e., when there is no enabled transition for the current event.
- *forceActionsOnTransition*[*t*]: assures the execution of actions when transitions are traversed. For example, *beforeActionToTransition* maps “incrNCnt” to “Tr2”, hence *countAtTime* is constrained so that: *countAtTime*[*t*]=*countAtTime*[*t*-1]+1 when a transition *Tr* has *beforeActionToTransition[Tr]=incrNCnt*, and *countAtTime*[*t*]=*countAtTime*[*t*-1] otherwise. Similarly, this predicate constrains *t* to be in *restartTimes* if and only if *stateAtTime*[*t*] is a FS state with an action timer.restart.
- *timeOut*[*t*]*:* imposes the following constraint on *localEventAtTime*[*t*]. Let *t*’ be *max*{*i*∈*restartTime* | *i*<*t* }; then *localEventAtTime*[*t*]=*timeoutFire* if and only if *t-t*’ equals the timer’s initial value (e.g., 5 time units in Figure 1).

The constraint set representation of a FS is the conjunction of its structural and semantic Formulae.

4.2 USING BOUNDED CONSTRAINT SOLVING TO MATCH TESTS WITH A CONCRETE FS-ASSERTION

Concrete FS assertions are ones in which:

- The test event namespace is the same namespace used in the assertion, such as P, Q, and R in Figure 3.
- Time and counting parameters, such as N is Figure 3, are constrained with tight bounds, such as N=3.

Absent a constraint representing the input sequence, the Kodkod solver will solve the FS constraint set by finding an instance of some legal sequence of local-events and state changes for the FS. This instance sequence is provided in the form of an instance assignment to the *at time* relations, i.e., the *localEventAtTime*, *stateAtTime* and *countAtTime* relations.

A specific input sequence is added to the constraint set by enforcing tight bounds to the *localEventAtTime* relation, such as the test of Listing 3.

```
          TupleSet eventAtTimeTS = tf.noneOf(2);
          Tuple tuple = tf.tuple(""+0, "P");
eventAtTimeTS.add(tuple);
          tuple = tf.tuple(""+3, "Q"); eventAtTimeTS.add(tuple);
  tuple = tf.tuple(""+5, "Q"); eventAtTimeTS.add(tuple);
          tuple = tf.tuple(""+14, "Q"); eventAtTimeTS.add(tuple);
          tuple = tf.tuple(""+15, "Q"); eventAtTimeTS.add(tuple);
          bounds.boundExactly(localEventAtTime, eventAtTimeTS);
```

Listing 3. Representing a specific test scenario using tightly bounded constraints.

Listing 4 contains the top-level formula for a test that is expected to lead the FS to the *Error* state. It contains three conjuncts: a *declarations* formula enforcing the FS’s structural constraints as specified in section 4.1, the *existsCompuation* formula enforcing its semantics, and the *endsInErrorState* formula. Given this top-level formula, the constraint solver searches for an instance assignment to the FS specification (e.g., a *stateAtTime* assignment) that satisfies this formula.

```
public Formula assertFalseIsSuccess() {
          return declarations()
          .and((existsComputation()))
          .and(endsInErrorState ())
          ;
     }
```

Listing 4. The top-level formula for a FS and a test that is expected to lead the FS to the Error state

A similar top-level formula, that uses *endsInErrorState.not*() instead of *endsInErrorState* , is used for a test that is expected to lead the FS to a good (non-*Error*) state.

4.3 MATCHING TESTS WITH A GENERIC FS-ASSERTION

The FS representation of section 4.1 uses binary relations that map events, states, and count values to time. Therefore, this representation is only capable of solving constraint pertaining to a *single* input sequence. Our technique however, matches a FS to a *collection* of tests, called a test-suite. A test-suite is broken into two parts, *acceptable* tests - that represent acceptable system behavior, and *failing* tests - that represent unacceptable system behavior.

To represent the behavior of a FS in the context of a test-suite, we modify the three "at time" relations of section 4.1, namely *stateAtTime, countAtTime,* and *localEventAtTime* to be ternary relations, with the third dimension of each relation being a test identifier, such as the test number.

In addition, we introduce the following additions and modifications to the FS structural constraints of section 4.1:

1. A new ternary relation named *realEventAtTime* is used to represent the *concrete* test-suite, i.e., a collection of *concrete*-event sequences generated by a given test scenario. The domain of these events is a set called *realEvents*. This relation is in addition to *localEventAtTime* which represents *generic*-event sequences. Hence for example, the tuple <1, 5, *pumpReset*> of *realEventAtTime* represents the concrete event *pumpReset* (assuming an infusion pump domain of discourse for the real events) being generated at time slot 5 of test number 1. Similarly, the tuple <1, 5, *P*> of *localEventAtTime* represents the generic event *P* occurring at time slot 5 of test number 1.
2. Time-bounds and count-limits within generic FS's, such as the limit N in the FS of Figure 3, are represented with a relaxed constrained, allowing them to range between prescribed lower and upper bounds, such as [1,10].
3. Newly added structural constraints specify requirements for a mapping *S*: *realEventAtTime* → *localEventAtTime* as follows:
    a. We define surjective function *R*: *realEvents* → *localEvents*.
    b. For every test in the test-suite, the length of the test sequence in *realEventAtTime* is equal to the length of that test in *localEventAtTime*.
    c. Let *tn* represent a test number, *t* a time slot, *Er* a real-event, and *Eg* a generic event. *S* specifies that for every tuple <*tn, n, Er*> in *realEventAtTime,* there exists <*tn, n, Eg*> in *localEventAtTime such that Eg* = *R*(*Er*).

The semantic constraints for a generic FS computes a *existsCompuation* Formula per test, thereby enforcing an FS computation for that particular test.

Given a library of generic FS assertions and a test suite, the discovery of a concrete FS that matches a given test suite is done by checking individual generic FS's from the library for a match, including the discovery of concrete parameters that make that FS concrete, as discussed below.

Given a single generic FS *F* and a test-suite *TS* with *Na acceptable* tests and *Nf* failing tests, the Formula of Listing 5 provides the top-level constraints for the constraint solver when checking whether *F* conforms to *TS*. If it does, the instance solution discovered by the constraint solver induces a concrete version of the generic FS by providing the function R, the mapping S, and specific time and counting parameter values.

```
public Formula assertWithTestCollection
    (AcceptableTests acceptableTests, FailingTests failingTests) {
         Formula f = Formula.TRUE;
         for (Expression tn: acceptableTests) {
              f = f
                   .and(declarations(tn))
                   .and((existsComputation(tn)))
                   .and(endsInErrorState(tn).not())
                   ;
         }
         for (Expression tn: failingTests) {
              f = f
                   .and(declarations(tn))
                   .and((existsComputation(tn)))
                   .and(endsInErrorState(tn))
                   ;
         }
     return f;
    }
```

Listing 5. The top-level formula for a discovering whether a candidate FS matches a test suite

5 CONCLUSION

It is safe to say that say that completeness problems in computer science are notoriously difficult, starting from Gödel's incompleteness theorem and its Halting problem counterpart. Detecting missing assertions belongs to this category because one does not know a requirement one has not thought of. Hence the importance of the first technique, albeit lightweight.

As for the second, formal technique, more research is needed to extend the technique to work in the presence of existing concrete formal specifications that might be in conflict with the ones deduced using our technique. Similarly, more research is needed to extend the technique to prune unreasonable automatically generated concrete FS’s.

REFERENCES

[CW] E. Clarke, J. Wing, et. al., “Formal Methods: State of the Art and Future Direction,” ACM Computing Surveys, vol. 28, no. 4, pp. 626-643, Dec. 1996.

[CE] E. M. Clarke and E. A. Emerson, “Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic,” in *Proc. Workshop on Logic of Programs*, 1981, D. Kozen, ed., *LNCS 131*, Springer-Verlag, pp. 52-71

[D1] D. Drusinsky, *Modeling and Verification Using UML Statecharts – A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, Elsevier, 2006.

[D2] D. Drusinsky, *Practical UML-based Specification, Validation, and Verification of Mission-critical Software. Space Exploration and Defense Software Examples in Practice*. ISBN: 978-145750-494-5.

[DMTS] D. Drusinsky, J.B. Michael, T. Otani, and M. Shing, “Validating UML Statechart-Based Assertions Libraries for Improved Reliability and Assurance,” *Proc. 2nd International Conf. on Secure System Integration and Reliability Improvement*, Yokohama, Japan, 14-17 July 2008, pp. 47-51).

[DMOS] D. Drusinsky, B. Michael, T. Otani, M. Shing, Validating UML Statechart-Based Assertions Libraries for Improved Reliability and Assurance,. Proceedings of the Second International Conference on Secure System Integration and Reliability Improvement (SSIRI 2008), Yokohama, Japan, 14-17 July 2008, pp. 47-51. Best paper award.

[ELC] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, “Experiences Using Lightweight Formal Methods for Requirements Modeling,” IEEE Trans. Software Eng., vol. 24, no. 1, pp. 4-14, Jan. 1998.

[HR] K. Havelund and G. Rosu, “An Overview of the Runtime Verification Tool Java PathExplorer,” *Formal Methods in System Design*, Vol. 24, Springer Netherlands, 2004, pp. 189-215.

[HU] J. E. Hopcroft, J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 2006.

[K] Kodkod, A Constraint Solver for Relational Logic, http://alloy.mit.edu/kodkod/

[SEEC] http://www.scientificamerican.com/article.cfm?id=world-changing-ideas-2011

[MZ] S. Malik and L. Zhang, “The Quest for Efficient Boolean Satisfiability Solvers,” *Proc. 14th Conf. on Computer Aided Verification (CAV2002),* Copenhagen, Denmark, July 2002, pp. 17-36.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Research Sponsored Programs Office, Code 41
   Naval Postgraduate School
   Monterey, CA 93943

4. Dr. Sukarno Mertoguno, Office of Naval Research
   Office of Naval Research (ONR) -
   One Liberty Center, 875 North Randolph Street, Suite 1425
   Arlington, VA 22203-1995

5. Professor Doron Drusinsky
   Naval Postgraduate School
   Monterey, California

THIS PAGE INTENTIONALLY LEFT BLANK