Real-time Java API Specifications for High Coverage Test Generation[*]

Wolfgang Ahrendt
Chalmers University of Technology
Gothenburg, Sweden
ahrendt@chalmers.se

Wojciech Mostowski
University of Twente
Enschede
The Netherlands
w.mostowski@utwente.nl

Gabriele Paganelli
Chalmers University of Technology
Gothenburg, Sweden
gabpag@chalmers.se

ABSTRACT

We present the test case generation method and tool KeY-TestGen in the context of real-time Java applications and libraries. The generated tests feature strong coverage criteria, like the Modified Condition/Decision Criterion, by construction. This is achieved by basing the test generation on formal verification techniques, namely the KeY system for Java source code verification. Moreover, we present formal specifications for the classes and methods in the real-time Java API. These specifications are used for symbolic execution when generating tests for real-time Java applications, and for oracle construction when generating tests for real-time Java library implementations. The latter application exhibited a mismatch between a commercial library implementation and the official RTSJ documentation. Even if there is a rationale behind this particular inconsistency, it demonstrates the effectiveness of our method on production code.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.4.7 [**Software Engineering**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Real-time Java, Test generation, Coverage

1. INTRODUCTION

The rapidly growing complexity of software deployed on embedded systems seriously threatens the effectiveness of traditional approaches to verification and certification. This is a huge problem in a time where effective verification and certification of embedded software becomes ever more important, as software is increasingly used as a core building block in almost all safety critical application areas. The good news is, however, the growing trend in the embedded software area towards using higher-level programming languages and paradigms, for instance by partly using Java rather than C for development. Even if the original motivation for using higher-level languages is rather increased productivity, modularity, and maintainability, this trend also enables more innovative, powerful verification methods. Higher-level languages allow stronger automated analyses, through their higher level of abstraction, and a better separation between domain related concerns (handled on the source code level) and lower-level concerns (delegated almost entirely to compilers and virtual machines). Among the powerful analysis methods, probably the strongest, but also the heaviest, is *formal verification*, which statically reasons about the correctness of *all possible runs* of a program. This requires precise definitions of the correct behaviour in the first place, which is the role of *formal specification*, written in some mathematically precise *specification language*.

Even if the area of formal verification made tremendous progress and provided powerful tools in the last decade or so, these methods are still rather heavy for mainstream usage. However, along with these developments there emerged various *lightweight formal methods*, where formal verification is used as a base technology for a more lightweight purpose, like automated test generation [6, 14, 11, 2]. (In the remainder of the paper, we will simply say 'verification' when referring to 'formal verification', in particular also when using the phrase 'verification based test generation'.)

The main contributions of our work are a verification based test case generation method and tool for real-time Java, and the development of formal specifications for the classes and methods in the real-time Java API. Continuing the work by Engel, Hähnle, Beckert, and Gladisch [11, 2], we have developed KeYTestGen, which is based on the KeY tool [3], a software verification system for Java. From Java source code augmented with formal specification given in the Java Modelling Language (JML [7]), KeY generates proof obligations in a program logic, called 'dynamic logic' for Java. During verification with the KeY prover, the proof branches over the necessary case distinctions, largely triggered by Boolean decisions in the source code. On each proof branch, a certain path through the program is *executed symbolically*. KeYTestGen uses the same machinery for a different purpose, namely generating test cases (for

[*]This work was funded by the Artemis Joint Undertaking in the CHARTER project, grant-nr. 100039. See http://charterproject.ning.com.

the popular JUnit framework). The key idea is to let the prover build an unfinished proof tree, to then read off from each proof branch a *path constraint*, i.e., a constraint on the input parameters and initial state for this path being taken during execution of the source code. We generate concrete test input data satisfying each of these constraints, thereby achieving strong code coverage criteria, in particular the Modified Condition/Decision Criterion (MCDC, see Sect. 3.1.1), by construction.

In addition to the source code, KeYTestGen requires formal specifications, for two purposes. First of all, specifications are needed to complete the test cases with *oracles* to check the test's pass/fail status. We generate such oracles in particular from *postconditions* of the methods under test. The second, and in the context of this paper more important, role of specifications is to allow symbolic execution of method calls within the code under test. The prover will use the specification, rather than implementation, of called methods to continue symbolic execution. In particular, frequently used library methods need to be specified. On our target domain, real-time Java, the crucial library is the real-time Java API (javax.realtime), for which we developed JML specifications suitable for symbolic execution.

The third contribution of our work is to automatically generate (and run) test for a commercial *implementation* of the real-time Java API from the Jamaica RT virtual machine developed by aicas. Here, KeYTestGen automatically generated a failing test, thereby detecting a mismatch between the library implementation and the official RTSJ documentation. Even if there is a rationale behind this inconsistency (see 5.3 for a brief discussion), it demonstrates the effectiveness of our method on production code.

This work has been performed in the frame of CHARTER (Critical and High Assurance Requirements Transformed through Engineering Rigour, see title footnote), a project within the ARTEMIS Embedded Computing Systems Initiative. CHARTER provides a methodology and tool chain designed to ease, accelerate, and cost-reduce the certification of critical embedded systems by melding real-time Java, Model Driven Development, rule-based compilation, automated testing, and formal verification. KeYTestGen and the JML specifications of the real-time Java library are important cornerstones in this endeavour.

2. CODE CONTRACTS

The idea of *design by contract* was initially developed by Bertrand Meyer for the Eiffel object oriented language [21], and is very resemblant of the Hoare programming logic correctness triples [16]. In essence, a method (or procedure) contract states, using some kind of a formal language, mutual commitments between caller and callee. In method contracts, *'preconditions'* formulate conditions on the pre-state and inputs which the *caller* is obliged to fulfil, and which the *callee's* implementer can rely on. In turn, the *'postconditions'* formulate conditions on the post-state and output which the *callee* is obliged to fulfil, and which the *caller* can rely on. In general, no guarantees are made for method entry states and inputs that do not satisfy the precondition. A contract can be viewed as a formalised natural description of the method behaviour from the API documentation. For example, a method that sorts an array of integers can be informally specified as:

"The method sortArray expects a non-null array as its input. The outcome of the method is that the elements of the input array are rearranged in the ascending order according to integer number ordering."

This description does not say what happens if the method is provided with a null array. A corresponding formal contract for method sortArray(**int**[] a), using generic First Order Logic as the formal language, is the following:

$$\begin{array}{ll} Pre: & \mathsf{a} \neq null \\ Post: & \forall_{i:int} i > 0 \land i < |\mathsf{a}| \rightarrow \mathsf{a}[i-1] \leq \mathsf{a}[i] \end{array}$$

where $|\cdot|$ is the array length operator. Note that (a) we do not state what happens for null arrays; (b) the postcondition does not state that the output array is a permutation of the input array: in the extreme case, an implementation filling the array with zeros would also make this formal specification valid. To fix this we use quantifiers and state the following:

$$\begin{array}{ll} Post_2: & \forall_{i:int} i \geq 0 \land i < |\mathsf{a}| \rightarrow \\ & \qquad \exists_{j:int} j \geq 0 \land j < |\mathsf{a}| \land \mathsf{a}[i] = pre(\mathsf{a}[j]) \end{array}$$

The postcondition uses the *pre* operator to refer to values at the beginning of the call. This allows to relate states before and after the method call. Apart from pre- and postconditions, there are often conditions which should be maintained under all operations, like sortedness, being balanced, or consistency constraints on redundant structures. Such conditions are called *invariants*, and are required to hold at times when no method call is active. That is, methods can assume invariants when they start execution, may break them temporarily, and have to re-establish them upon exit.

The practical applications of formal contracts are the following. The first and most obvious one is merely for stronger documentation, in a language providing strong precision compared to natural descriptions. Going further, after translation the formal expressions can be used to do run-time checks of the executing code. In our example, a run-time checker would perform the following tasks upon every call of the sortArray method. Before the call it would (a) check that the expression contained in the precondition evaluates to true, and (b) record the contents of the inputs array in a temporary storage. Upon completion of the method the checker would evaluate the postcondition expression. In the case of our example it would need to iterate over the contents of the array and check the associated formulas. Simpler expressions' checks can be also easily performed with the help of built-in assert statements of the programming language.

The ability to perform run-time checks already hints on the possible application of formal specification in test generation, namely to (partly) serve as a test oracle, as we explain in Sect. 3. The ultimate application of formal contracts, however, lays in a deep analysis technique called *formal verification*. Using mechanised theorem provers and associated logic systems, programs are proven to be correct statically, i.e., before they are run or even compiled. During this process, execution paths of the program are analysed using symbolic values which are characterised by *formulas* capturing infinite or large sets of values concisely. Whenever different parts of the data domain cannot be treated uniformly, proofs branch into case distinctions. In particular, branching points in the control flow (e.g., **if**, **while**, exception handlers) lead to branches in the proof. Therefore, each path through the source code is represented by one (or more) proof branches.

Figure 1: Specification of sortArray in JML

```
     /*@ public normal_behavior
 2         requires a != null;
           ensures (\forall int i; i>0 && i<a.length ==> a[i−1] <= a[i]);
 4         ensures (\forall int i; i>=0 && i<a.length ==> (\exists int j; j>=0 && j<a.length && a[i] == \old(a[j])));
           assignable a[*];
 6       also public exceptional_behavior
           requires a == null;
 8         signals (NullPointerException npe) true;
           assignable \nothing; @*/
10   public void sortArray(int[] a);
```

Even unfinished verification attempts still represent an analysis of all possible execution paths up to the depth of the proof attempt. The resulting execution tree is another important building block in automated test generation to be discussed in Sect. 3. In particular, it enables the achieving of high coverage criteria.

Formal verification is in general considered difficult and time consuming. This is mostly caused by the fact that complete correctness proofs for complex code usually require human input and interaction with the verification tool. In turn, deep understanding of the tool and the underlying logic from the user is required. In this context modularisation is an important factor. The idea of reusable API libraries is lifted to the verification process. Each method is specified and verified in separation, and when another program to be verified calls some method only the called method's specification is considered, rather than the implementation that should be already verified. This is especially important when a program to be verified utilises proprietary APIs, for which the source code may not be available. Lacking this verification target, an API method is left unverified, and its specification provides the only means to describe its behaviour in the verification context of other programs. The benefits of modularisation apply to other verification based techniques as well, in particular to test generation to be described.

The lack of good quality specifications for commonly utilised API is in fact a shortcoming for formal verification approaches. For Java a community effort is continuously under way to provide specifications for a wide range of standard Java API [7]. The main application context of the CHARTER project is the real-time Java and safety critical programs. To evaluate our test generation efforts on the associated real-time Java case studies we developed formal specifications for the classes and methods in the javax.realtime package, which interfaces the core real-time functionality of the JVM to Java programs. We describe the exact role of these specifications in test generation (Sect. 3.1.2) as well as the specifications themselves (Sect. 4).

The actual formal language used to specify method contracts primarily depends on the programming language itself, but may be also specific to the validation tool used (program verifier or run-time checker). The level of integration in the programming language can also differ. In Eiffel [21] contracts are an inherent part of the language with special syntax devoted to specify properties of the program on the same level as code. In Microsoft Code Contracts for C# [22] specifications are also part of the program, however, the C# syntax is not extended, rather contracts are embedded in calls to special *contract libraries*. The Object Constraint Language (OCL) [9] was developed to express formal properties about UML models, but here this is done externally to the UML model. For Java the de facto standard specification language is Java Modelling Language (JML) [20], a research community effort to provide a general, tool independent specification framework for Java. In contrast to other examples, JML is contained in the code it specifies, but is not part of the Java language. This is achieved by embedding JML specifications in special Java comments, similar to JavaDoc tags. We give a very brief introduction to JML in the following.

2.1 Java Modelling Language

JML specifications reside in Java files in comments marked with the @ sign as the first character of the comment. Thus, comments starting with /*@ or //@ indicate that a JML specification follows. Fig. 1 shows such a specification for the array sorting method we discussed above, to handle possible exceptions. A method specification is tagged with a marker indicating whether only non-exceptional behavior (**normal_behavior**, line 1 in the example), purely exceptional behavior (**exceptional_behavior**, line 6), or both (simply **behavior**) is specified. The core method specification can contain preconditions (marked with the **requires** keyword, lines 2&7), and normal (**ensures**, lines 3&4) or exceptional (**signals**, line 8) postconditions. Exceptional postconditions specify *both* the type of the exception that is thrown and the condition that should nevertheless hold after this event; in our example this is simply true.

In addition to behavioural descriptions, 'framing conditions' are specified with the **assignable** keyword. They characterize memory locations (object fields or array elements) that a method may *at most* modify. This information about the memory scope the method operates on is necessary for verification tools to correctly reason about data dependencies and possible object aliasing. Each method can have a number of specification cases, separated by the **also** keyword, which describe different behaviour cases under different preconditions. However, this should be only considered a syntactic convenience – multiple specification cases can always be combined into one specification case tagged with **behavior**. Fig. 2 shows such a combined specification case for our array sorting method. In particular, the combined precondition now is true, hence a trivial **requires** clause is removed from the specification altogether.

The JML specification expressions themselves are side effect free Java Boolean expression extended with classical

Figure 2: Combined JML specification of sortArray

```
/*@ public behavior
    ensures a != null ==> (\forall int i;
      i>0 && i<a.length ==> a[i−1] <= a[i]) && ...;
    signals (NullPointerException npe) a == null;
    assignable a[*]; @*/
public void sortArray(int[] a);
```

First Order Logic operators. Finally, class invariants are specified anywhere in the scope of the class using the **invariant** keyword. We discuss some more details of JML when presenting the Real-time Java API specifications in Sect. 4.

3. THEOREM PROVING BASED TEST GENERATION

In this section we introduce the usage of verification tools in the automated generation of test cases. We put a strong focus on test coverage, as it is one of the crucial aspects in software certification. Then we briefly describe how the KeYTestGen tool works and how it interfaces with other verification tools. We also discuss the role of formal specifications in automated test case generation.

3.1 Test Coverage from Verification

Verification based automated test case generators work with some form of symbolic execution of the code [28, 10, 11] or use model checking on models of the code [26, 29]. The goal is to guarantee that the generated test suite fulfils certain coverage requirements. Symbolic execution unwinds the paths and collects the (symbolic) constraints under which each path is taken. The constraints characterize the initial state and the input arguments of the Method Under Test (MUT); the constraints are solved using specific solvers that provide concrete test values, i.e., witnesses for the collected path constraints.

Test coverage is an important software quality metric, and meeting specific coverage criteria is a requirement for certification of safety-critical software, like in the European/American avionics standard ED-12/DO-178 [12]. In principle, verification based techniques using symbolic execution can achieve logical and graph coverage criteria [1] *by construction*, as the symbolic execution engine will evaluate symbolically Boolean expressions (ensuring logical coverage), and execute symbolically all the (feasible) paths inside the program (ensuring graph coverage). In practice this is not always the case due to the trade-off between precision and feasibility of the analysis discussed in the rest of the paper. In any case, test case generation (with guaranteed coverage) relieves the developer from manually constructing the test cases (and judging their coverage).

3.1.1 *MCDC coverage criterion*

Modified Condition/Decision Criterion (MCDC) [15] is a logical/graph coverage criterion mentioned in the ED-12/DO-178 standard as the required coverage for software whose failure may cause an air-crash of the aircraft on which it is running (*Level A software* in the standard). In the MCDC terminology, a *decision* is a top-level Boolean expression in the program, whereas a *condition* is an atomic Boolean expression not composed of other Boolean expressions. In short, a test suite fulfilling MCDC for a MUT (a) executes each statement in the MUT at least once; (b) exercises all entry and exit points in the MUT; (c) evaluates each non-constant decision D and condition c once to true and once to false; (d) shows for each condition c in decision D that c affects the evaluation of D independently of any other condition c' in decision D.

3.1.2 *The Role of JML Specification*

Formal specifications are crucial to automate the test generation process. They play a double role in the automated testing approach, as follows.

Conjectural use: specification of MUT.

In the specification of the MUT (a) the precondition constrains the setup of the test, and (b) the postcondition describes expected properties of the method's result and final state, from which the test oracle is derived. The specification of the MUT acts as a conjecture about its behaviour, the property we test against.

Axiomatic use: specification for method calls in MUT.

For symbolic execution to be modular and scalable, a call to method m from within the MUT is processed using the specification, rather than the implementation, of m. Expanding the implementation of called methods would lead to high proof complexity, and would require full availability of all callees' code (including libraries, third party code, and native methods). Also, implementation changes would not be manageable. Specifications of called methods can be seen as axioms during symbolic execution of the MUT, both in the context of verification and verification based test generation. The challenge is, however, that formal specifications for called methods must be powerful enough to allow meaningful symbolic execution. One of the contributions of this work lies exactly here, in JML specifications of the javax.realtime package which are suitable for symbolic execution, see Sect. 4.

3.2 KeYTestGen

KeYTestGen uses KeY [3] as the underlying verification technology. KeY is a Java source code verification tool which can formally prove that program units comply with their JML specification. The construction of proofs is dominated by *symbolic execution* of the code. During test generation, KeY uses the symbolic execution not for proving any postcondition correct, but only to collect path constraints. Therefore, a particular strategy for manipulating the program is implemented and specialised for test case generation. Path constraints are updated as symbolic execution proceeds, recording the location changes coming from assignments, and the case splits corresponding to conditional statements (including loops, which are unwound a fixed number of times). The information relevant for test case generation is filtered from the proof tree, yielding an *execution tree*. Its leaves contain path constraints, characterising conditions for one path in the program to be taken. Concrete test inputs are generated by solving such constraints.

Along a simple example given in Fig. 3, we describe how KeYTestGen generates test cases and achieves MCDC. The method scan(**int** t, **int** i) iterates on calling the method read (line 7) that returns a value between r and 2r+1. This can

Figure 3: Path constraint collection example

```
   /*@ public normal_behavior
2    requires t>=0 && i>=0;
     ensures \result > 0; @*/
4  public int scan(int t, int i){
     int r = 0, j = 0;
6    while(r<t && j<i) {
            r = read(r);
8           j++;
     }
10   return j;
   }
12
   /*@ public normal_behavior
14   ensures i <= \result && \result <= 2*i+1; @*/
   public int read(int i);
```

be, e.g., a read from a file, or from a sensor. The loop terminates when one of the following conditions holds: (a) the value read equals or exceeds the threshold t; (b) the number of iterations equals or exceeds the value i.

Execution tree: Symbolic execution.

The KeY prover symbolically executes the code and provides path constraints for every leaf in the execution tree. The execution tree in Fig. 4 highlights a certain path by solid arrows. It is the path where all Boolean conditions are evaluated to true. Note that: (1) all possible outcomes of the decision in line 6 in Fig. 3 are considered; (2) KeYTestGen unwinds loops a user-specified number of times, in our example just once; (3) the method read(int i) is not expanded but its contract is used instead.

Path constraints: SMT solving.

The path constraints are shown top-right of the return statements in the different paths of the execution tree. Such constraints are solved using a Satisfiability Modulo Theory (SMT) solver. In short, it checks the satisfiability/validity of a logical formula with respect to background theories with equality expressed in First Order Logic. KeYTestGen uses Simplify [27] iteratively to find concrete models for the path constraints.

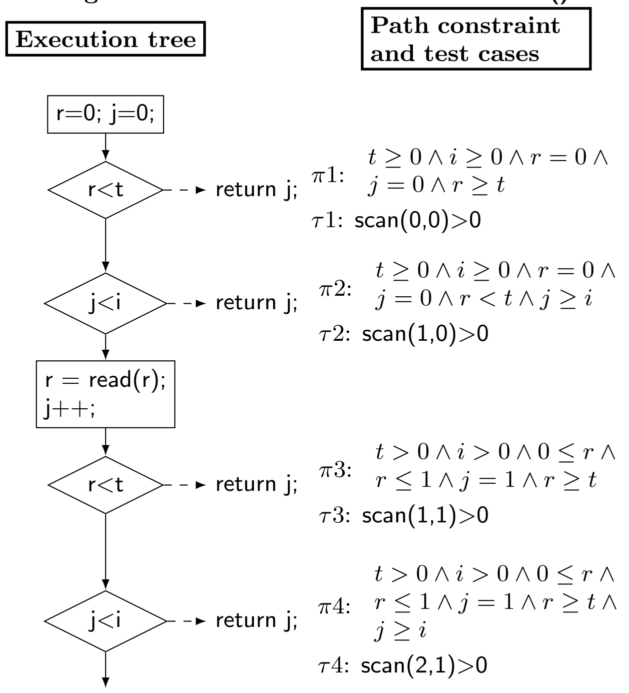
Test cases.

Test cases in the JUnit format are generated for each leaf using the concrete values computed in solving the path constraint. We discuss three main aspects in the following:

Test inputs are generated from path constraints like, in the example, $\pi1$,$\pi2$,$\pi3$,$\pi4$ (Fig. 4), by finding witnesses, satisfying the constraints. In the example, these witnesses appear as arguments of scan in $\tau1$,$\tau2$,$\tau3$,$\tau4$. In the case of primitive types (as in the example) or simple reference types like standard objects (with certain visibility restrictions on fields and methods), the translation is fully automatic and consists in assignments, calls to mock object libraries [23] (to overcome visibility modifiers) or the java.lang.reflect.* API. When the architecture of the reference type inputs becomes more intricate this is often hard to automate as explained later in Sect. 6.

Oracles are generated from the MUT contract's postcondition, turned into executable Java code. In the example, the oracle accounts to check if the returned value is bigger than zero (Fig. 4, test cases $\tau1$,$\tau2$,$\tau3$,$\tau4$).

Figure 4: Constraint Collection for scan().


Execution tree
Path constraint and test cases
r=0; j=0;
r<t
return j;
$\pi1$: $t \geq 0 \wedge i \geq 0 \wedge r = 0 \wedge j = 0 \wedge r \geq t$
$\tau1$: scan(0,0)>0
j<i
return j;
$\pi2$: $t \geq 0 \wedge i \geq 0 \wedge r = 0 \wedge j = 0 \wedge r < t \wedge j \geq i$
$\tau2$: scan(1,0)>0
r = read(r); j++;
r<t
return j;
$\pi3$: $t > 0 \wedge i > 0 \wedge 0 \leq r \wedge r \leq 1 \wedge j = 1 \wedge r \geq t$
$\tau3$: scan(1,1)>0
j<i
return j;
$\pi4$: $t > 0 \wedge i > 0 \wedge 0 \leq r \wedge r \leq 1 \wedge j = 1 \wedge r \geq t \wedge j \geq i$
$\tau4$: scan(2,1)>0

MCDC Coverage is guaranteed by construction because of the way symbolic execution evaluates decisions; in particular, the decision in line 6 in Fig. 3 is the only decision D of the program with two conditions q = r<t and c = j<i. To show that c affects independently of q the value of decision D (item (d) in 3.1.1), the test case pair ($\tau2$,$\tau4$) is sufficient since in test case $\tau2$ c evaluates to false, causing D to evaluate to false while q is true; and in test case $\tau4$ c evaluates to true, while keeping q true, causing D to evaluate to true and enter the loop. Similar reasoning applies to condition q.

4. JAVA RT API SPECIFICATIONS

The development of the real-time Java API specifications was driven by the needs of the corresponding tools in the CHARTER processes, in particular the test generation we have just described. The questions to answer first are which parts of the API should one specify as the first priority, and what should be the specification style and detail of the specifications. In other words, *what* exactly should we specify, and *how*?

4.1 What to Specify

Since the main focus of our work is the application area of real-time Java, the primary goal is to specify the core classes in the real-time interface of the API, i.e., classes and interfaces residing in the javax.realtime package. Furthermore, during our work we used two particular case studies to evaluate our efforts:

- An aviation collision detector program, CDx – a free real-time Java benchmark suite, mostly targeted at performance evaluation of real-time virtual machine implementations[1] [18]. CDx is a relatively small application, making it a perfect target for evaluation.

- A gaming device (light gun) driver – this case study was developed by us as a preliminary evaluation target for our tool. It provides two simple implementations for detecting "target hit" events when the light gun is fired at a game screen. One implementation is based on tracking and timing the CRT beam, the second one on detecting a white square on the screen during a synchronised display of one special frame on the screen.

The analysis of both case studies confirmed that the realtime package should be in the centre of attention for our formal specifications. Both applications rely almost entirely on the real-time package, the collision detector additionally utilises a few collection classes from the java.util package, but these are used outside of the core real-time functionality of the suite. For these utility classes it is sufficient to reuse the existing API specifications developed by the JML community.[2] We come back to the evaluation of the test generator and the real-time API specifications with these two case studies in Sect. 5.

4.2 How to Specify

The first aspect of the preciseness of the specifications is dictated by the KeY prover and the SMT solver Simplify – the tools driving our test generation procedure. That is, the specifications should be written in the subset of JML supported by the version of the KeY system that we use, and at a detail level that can be easily managed by the KeY prover and the SMT solver. In particular, overly complex specifications can cause the KeY prover to produce a proof tree of an unmanageable size. In turn, the SMT solver will not manage to resolve the resulting data instantiation problem.

This leads us to the second aspect of specification writing, namely the double role the specifications play in the test generation process as described in Sect. 3.1.2. Let us start with the conjectural use of specification. For this, the specifications should provide assertions that can be relatively easily checked during run-time. One type of specification constructs difficult to evaluate at run-time are all sorts of quantifiers. They do not have to be avoided altogether, however, one should keep in mind that in the test oracle context they may be left unchecked, thus less effort should be put into specifying properties that may involve quantifications. Then, following the implementation independence paradigm, properties should be expressed using the public interface of objects through the getter methods, rather than using any internal representation that objects may have. The generated test cases based on such cleanly expressed properties will be reusable between different API implementations. Also, for this role the method frame conditions (JML assignable clauses) are not that vital, because again they are usually expressed in terms of the internal object representation. Finally, the specifications should cover a complete behaviour of the given method, to provide a wide test oracle. In other words, the specification should account for all possible inputs that the method may be called with, including the ones causing any exceptional behaviour, so that the resulting test case covers a wide range of possible outcomes of the method.

In the context of the axiomatic use of the method specification for modularisation, we need to accommodate further elements in our formal contracts. The key concept of modular verification is that upon method calls the symbolic state of the verified program is modified according to the method specification, rather than the method implementation. In this situation the reasoning provides more precise results when the method specifications also provide framing conditions we mentioned earlier. Such conditions enable the prover to disregard the parts of the state of the program that is *not* changed [4], this in turn allows for more precise analysis of the different program branches that takes into account only the part of the symbolic state that *does* change.

Hence, we would like to represent the internal state of objects without actually referring to concrete implementation details. JML provides two mechanisms for that, very similar to each other, called model fields and ghost fields [5]. The actual difference between the two is very fine-lined and detailed explanations are beyond the scope of this paper. Furthermore, the stable version of the KeY prover that our test generator is currently based on only provides good support for ghost fields, thus this is what we used and briefly describe next. The possibility to use model fields instead of ghost fields is discussed later in Sect. 6 on future work.

A ghost field is a specification-only field declared in a class with any Java type, either primitive or reference. From the specification point of view, such a field is part of the state of the object. In verification contexts, where the implementation code is available, special specification only **@set** statements are placed in the method code to update the values of ghost fields and maintain consistent state of the object extended with corresponding ghost fields. In our case the API specifications are developed separate from any implementing code, hence ghost fields are only referenced in method pre- and postconditions as well as assignable clauses. In this situation ghost fields can be viewed as a mock representation of the actual state of the object. The KeY prover can then perform the client code analysis based on the symbolic representation of this state.

Having such a representation of the object state we can now write more detailed and accurate specifications making case splits over the possible states of the object. This again is important for test coverage. The more different specification cases we provide for an API method, the more different branches will emerge in the proof tree as a result of discharging this API call in the client code. This can possibly provide additional data partitioning on top of the one that results from considering different execution paths of the other Java statements, i.e., loops, if-statements, etc. described in Sect. 3.

In the following section we give a few modest samples of the real-time Java API specifications we developed. The formal specifications are based on the documentation of the Jamaica virtual machine API version 3.4[3] and the official Real-time Specification for Java (RTSJ) documentation.[4]

4.3 Specification Samples

The complete set of the real-time Java API specifications covers all of the over 70 classes in the javax.realtime package. More than 800 methods were specified with the total

[1] http://adam.lille.inria.fr/soleil/rcd
[2] http://formalmethods.insttech.washington.edu/specathons/
[3] http://www.aicas.com/jamaica/3.4/doc/jamaica_api/
[4] http://www.rtsj.org/specjavadoc/book_index.html

of almost 4000 lines of JML code. The specifications are available on the CHARTER resources web-page,[5] here we shortly discuss two examples: event handler management in the AsyncEvent class, and a constructor for the MemoryArea class.

4.3.1 Class AsyncEvent

This class represents events that can be triggered in the real-time system asynchronously. Each such event has a set of associated handlers. Upon event occurrence the handlers are triggered to service the event. The handlers management in the AsyncEvent class is specified in the following way:

```
public class AsyncEvent extends Object {

  //@ public instance ghost AsyncEventHandler[] _handlers;

  /*@ behavior
        ensures handler != null && handledBy(handler);
        signals (IllegalArgumentException iae)
          handler == null;
        assignable this._handlers[*]; @*/
  public void addHandler(
      /*@ nullable @*/ AsyncEventHandler handler)
    throws IllegalArgumentException;

  /*@ normal_behavior
        ensures handler == null || !handledBy(handler);
        assignable this._handlers[*]; @*/
  public void removeHandler(
      /*@ nullable @*/ AsyncEventHandler handler);

  /*@ normal_behavior
        ensures handler == null ==> \result == false;
        ensures handler != null ==>
          \result == (\exists int i;
            i >= 0 && i < this._handlers.length;
            this._handlers[i] == handler); @*/
  /*@ pure @*/ public boolean handledBy(
      /*@ nullable @*/ AsyncEventHandler handler);
}
```

The **nullable** JML tag states that we allow the associated reference to have null values. The current semantics of JML is that all references are by default non-null, so nullable references have to be explicitly tagged. The ghost field _handlers represents the storage of registered event handlers using a Java array. The specification of the methods follow the API documentation. An attempt to add a null handler causes an IllegalArgumentException, while adding a non-null handler causes all subsequent calls to a corresponding handledBy() method to return true. The handledBy() method is tagged as **pure**, meaning it is a getter method and does not change the state of the object. Its specification expresses the association between registering a handler and the contents of the _handlers array.

4.3.2 Class MemoryArea

This class is the base class for defining the different types of memory that the real-time Java environment supports, in particular the scoped memory [19]. Each memory area has a runnable logic associated with it, but this logic does not have to be defined until the memory is actually used, i.e., on construction it can remain null:

```
public abstract class MemoryArea extends Object {

  //@ public instance ghost nullable Runnable _logic;
  ...
```

The very important aspect of the memory areas is that the user defined classes cannot extend MemoryArea class, even though it is not final, i.e., only memory areas predefined in the real-time virtual machine are allowed as subclasses of MemoryArea. We express this with the following specification for the constructor:

```
/*@ behavior
      ensures size != null && this._logic == logic;
      signals (IllegalArgumentException iae)
          size == null ||
        !((this instanceof LTMemory) ||
          (this instanceof LTPhysicalMemory) ||
          (this instanceof VTMemory) ||
          (this instanceof VTPhysicalMemory) ||
          (this instanceof ImmortalMemory) ||
          (this instanceof ImmortalPhysicalMemory) ||
          (this instanceof HeapMemory));
      signals (OutOfMemoryError oome) true;
      assignable this._logic; @*/
protected MemoryArea(
  /*@ nullable @*/ SizeEstimator size,
  /*@ nullable @*/ Runnable logic)
  throws IllegalArgumentException, OutOfMemoryError;
```

Following the API documentation, an IllegalArgumentException is thrown if an attempt to create a memory area outside of the predefined set is made. This is expressed with the cascade of the **instanceof** expressions. A null size parameter causes the same exception. On top of that, the constructor can throw an exception caused by memory exhaustion, in this case no further conditions are specified.

5. EVALUATION ON CASE STUDIES

In the following we shortly describe three different use cases for our work. The difference is not only in the particular code that was validated, but also in the validation method itself. This shows the different ways in which formal specifications can be utilised with the test generator and the KeY verifier itself. Due to space restrictions we have to be very modest in quoting the program code, both of the use cases and of the generated tests. We also abbreviate some method names.

5.1 Test Generation for Light Gun Driver

The light gun driver provides two real-time algorithms for detecting a "target hit" event in some gaming system. The real-time aspect of these algorithms is that in both cases the procedure has to terminate and provide the result within the time needed to display two frames on a CRT screen. The following is the snippet of simple code responsible for delegating the registration of a handler in the v-sync timer object in a Screen class:

```
/** Sets the repainting behavior of this screen
    @param asi the handler to run whenever the
      vSync timer expires */
```

[5] http://charterproject.ning.com/page/resources-3

```
/*@ public behavior
      ensures (asi != null) ==> vSync.handledBy(asi);
      ensures (asi == null) ==> true;
      signals (NullPointerException npe) vSync == null; @*/
public void setVSyncHandler(AsyncEventHandler asi){
  vSync.setHandler(asi); }

public /*@ nullable @*/ Timer vSync;
```

The API method setHandler is very similar to the addHandler method we discussed in Sect. 4, only that when given a null argument it empties the set of handlers of the AsyncEvent object, and when given a non-null argument it sets the passed argument as the only handler, removing any previously registered handlers. The symbolic execution of the setVSyncHandler provides the following information to our test generator. From the specification of the API's setHandler, two cases for the asi parameter are provided, a null and non-null one. Then, the symbolic execution of the API call on the vSync field provides another case split, again, that the field can be also null or non-null. KeYTestGen generated four test cases for the setVSyncHandler method, all of which follow the same pattern, like the following:

```
public void testScreen_setVSyncHandler0 () {
  Throwable exc = null;
  AsyncEventHandler asi = RFL.new_AsyncEventHandler();
  Screen self = RFL.new_Screen();
  String exceptionTrace = "";
  String inputsBefore = "Value␣of␣asi:␣"+asi+
    "\nValue␣of␣vSync:␣"+self.vSync+"\n\n";
  try {
    self.setVSyncHandler(asi);
  } catch (Throwable e) {
    exc=e; exceptionTrace=e.toString();
  }
  StringBuffer buffer = new StringBuffer();
  boolean oracleResult =
    TestScreen_setVSyncHandler0.formula(exc,asi,self,buffer);
  String inputsAfter = "Value␣of␣asi:␣"+asi+
    "\nValue␣of␣vSync:␣"+self.vSync+"\n\n";
  assertTrue("\nPost␣evaluated␣to␣false."+
    "\nEvaluation␣of␣formulas␣so␣far:␣"+buffer.toString()+
    "\n\nInput␣values␣before␣method␣call:␣\n\n"+
    inputsBefore+
    "\n\nInput␣values␣after␣method␣call:␣\n\n"+
    inputsAfter+exceptionTrace+"\n\n", oracleResult);
}
```

The formula() method is an automatically generated evaluator of the top-level JML specification of setVSyncHandler and provides the actual test oracle. The class RFL is a factory responsible for generating reference type test data. The generated test case trivially fulfils MCDC since there are no decisions in the code. For this concrete instance we generate test cases in a completely automated way, but the general case where an arbitrary set of handlers is to be created is not straightforward to automate. We address this in Sect. 6.

5.2 Verification of Collision Detector

Apart from test generation our API specifications can be also used for full formal verification with the KeY prover. Similarly to test generation the method in question along with its specification is loaded into the tool and executed symbolically. However, instead of extracting symbolic path conditions from the proof branches, the tool attempts to statically evaluate the methods top-level specification to establish complete correctness of the code. If the tool can show the specification to be valid on all proof branches the method is considered fully verified with respect to the specified property. We illustrate this with the method roundUp from the Collision Detector benchmark:

```
public static AbsoluteTime roundUp(AbsoluteTime t) { ... }
```

This method is responsible for rounding up the provided time to the next sampling period. The JML specification that expresses this property for non-null input time t is the following:

```
/*@ public normal_behavior
  requires t != null;
  ensures
   1000000 * \result.getMillis() + \result.getNanos() >=
   1000000 * t.getMillis() + t.getNanos();
  ensures \result.getNanos() == 0;
  ensures \result.getMillis() % Const.PERIOD == 0; @*/
```

In natural language: the resulting time should be rounded to milliseconds, should be greater than the input time, and it should be evenly divisible by the applications sampling period. Intuitively this is a straightforward judgement. However, because of the arithmetic involved both in the code and the specifications, the task of establishing this property statically for all possible inputs is rather demanding for the KeY prover. The complete correctness proof is generated fully automatically by KeY in about 3 minutes, and consists of ca. 54 000 single proof steps.

This simple example also shows how complex the dependencies between the code and the API methods can be in the context of our work. The implementation of the roundUp method makes calls to API methods, the API methods are also used in the specifications, both of the roundUp method and the API methods used by the code. Fig. 5 shows a complete dependency tree. For fully modular test generation or formal verification all these methods have to be formally specified. Here, a fully modular processing of the roundUp method requires JML specifications of 6 different real-time API methods.

5.3 Test Generation for Jamaica RT API

Finally, the real-time API specifications were also used to generate tests for the API *implementation* itself. The particular API under test was one from the Jamaica RT virtual machine, the choice of real-time VM for the CHARTER project.

The effectiveness of the KeYTestGen was successfully demonstrated, by it generating tests which revealed an inconsistency for one of the methods in the commercial API implementation with respect to the official RTSJ documentation. More concretely, a test has been generated for the method absolute in class AbsoluteTime:

```
public AbsoluteTime absolute(Clock clock);
```

The discovered point of inconsistency is the clock association that this method should establish between the result and the clock parameter. The RTSJ documentation states that for a null clock parameter the resulting time object is associated with the system's default real-time clock, for a non-null parameter the association is made with the clock

Figure 5: Specification dependencies for roundUp()

```
roundUp(...)
 └ method code:
     └ HighResolutionTime.getMilliseconds()
     └ HighResolutionTime.getNanoseconds()
     └ AbsoluteTime.AbsoluteTime(...)
         └ specification:
             └ HighResolutionTime.getMilliseconds()
             └ HighResolutionTime.getNanoseconds()
             └ HighResolutionTime.getClock()
             └ Clock.getRealtimeClock()
                 └ MemoryArea.getMemoryArea()
 └ method specification:
     └ HighResolutionTime.getMilliseconds()
     └ HighResolutionTime.getNanoseconds()
```

parameter itself. This property has been specified with JML in the following way:

```
ensures clock != null ==> \result.getClock() == clock;
ensures clock == null ==>
  \result.getClock() == Clock.getRealtimeClock();
```

The running of the test generated for the first specification case above revealed a failure, i.e., no association to the passed clock is made in the resulting time object. Further study of the API implementation and its internal documentation showed that this behaviour is actually intended and does not cause any dysfunction of the API implementation. The rationale behind "ignoring" the clock parameter is that the Jamaica VM implements only one single real-time clock, moreover, the RTSJ specification does not declare any factory classes or methods for obtaining valid system clocks, which *probably* should be the only ones allowed in the system prohibiting user instantiated clock objects. We say probably, because this aspect is not mentioned in the RTSJ documentation, and actually opens a discussion that goes beyond the scope of this paper about clock usage in the Java RT API [30]. Regardless of that, this clearly shows how automated test generation through formal specifications can be used to discover problems.

6. CHALLENGES AND FUTURE WORK

Testing is one of the means of certifying software. In safety-critical applications white box testing coverage criteria are mandated – tests are required to exercise certain parts of the actual code. The ultimate property of a test case generation tool is to have a test suite fulfilling the required coverage criteria *by construction*. For this reason we believe that the most appropriate tools for this kind of process are those based on symbolic execution or similar technologies which ensure full analysis of the code structure. Black box testing tools, like JMLUnitNG [31], lack this feature and cannot reach our ultimate goal. Concerning the creation of initial (heap) states, JMLUnitNG faces similar challenges as KeY. They also have in common the conjectural use of specification (Sect. 3.1.2), which provides partitioning of test inputs and an oracle function, and in the case of JMLUnitNG this can be seen also as an approximation of path constraint. Initial state generation has two main difficulties:

Solving path constraints with quantifiers.

Gladisch [13] explored the limitations of using SMT solving to create reference type data, by proposing a preprocessing step and a search algorithm using SMT solvers. This is implemented in an experimental version of KeY. This model generation technique is currently interactive. The author claims that full automation is possible.

Concrete instantiation of initial state.

Solving path constraints tells what the input *is* but not *how to create it*, so in general the solution to the path constraint cannot be used directly. Suppose the analysis of the code given in Sect. 5.1 would mandate the creation of an AsyncEvent obj with an initialized list containing n handlers. Setting up the test requires a sequence of n calls to addHandler to fill the list of handlers. In general it is not immediate to infer the actual sequence of method calls from its API to build an object o of class C such that it has some property P. Currently KeYTestGen allows, similarly to other test generation tools, manual input of test data by writing it in the auxiliary files generated by the tool. A first approach to automate this could be to annotate appropriately methods that can be used to generate inputs, inspired by the data generators of QuickCheck [8, 17], and let KeYTestGen reason about them. Other improvements on this front will be to implement better integration with testing libraries (more mock libraries, QuickCheck for Java [24]) and concepts such as caching objects when testing constructors, as done in JMLUnitNG [31].

Currently KeYTestGen is still at a prototypical state and is distributed as Eclipse plugin[6] which allows test case generation for projects and classes. It will also feature a command line interface, for better integration into other tools and IDEs. Another future step is to adapt the test generator and the API specifications to the newer generation version of the KeY system. This new version, currently also still at the development stage, offers big improvements for abstracting the state in specifications using model fields and 'dynamic frames' [25]. These mechanisms are much more flexible and elegant than the ghost fields we discussed in this paper. The changes in the KeY system to support this new way of specifying contracts require revisiting our test generation technique and implementation. Finally, we plan to expand our collection of real-time case studies to further evaluate our work.

Acknowledgements

We would like to thank James Hunt of aicas for many discussions, making the Jamaica API implementation available, and for many insights concerning this implementation.

7. REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[6] http://www.cse.chalmers.se/~gabpag/eclipse

[2] B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In B. Meyer and Y. Gurevich, editors, *Proceedings, International Conference on Tests and Proofs (TAP), Zurich, Switzerland*, volume 4454 of *LNCS*. Springer, February 2007.

[3] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[4] B. Beckert and P. H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM)*, pages 91–99. IEEE Press, 2003.

[5] C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP'03)*, July 2003.

[6] A. D. Brucker and B. Wolff. Interactive testing with HOL-TestGen. In W. Grieskamp and C. Weise, editors, *Proc. Workshop on Formal Aspects of Testing, FATES*, volume 3997 of *LNCS*, pages 87–102. Springer, 2005.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. 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*, 7(3):212–232, June 2005.

[8] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. *SIGPLAN Notices*, 37(12):47–59, 2002.

[9] T. Clark and J. Warmer, editors. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.

[10] X. Deng, J. Lee, and Robby. Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. 21st IEEE/ASM Intl. Conference on Automated Software Engineering*, pages 157–166. IEEE Computer Society, 2006.

[11] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In B. Meyer and Y. Gurevich, editors, *Proc. Tests and Proofs (TAP), Zürich, Switzerland*, volume 4454 of *LNCS*. Springer, February 2007.

[12] EUROCAE. Software considerations in airborne systems and equipment certification, January 2012. Document ED-12C.

[13] C. Gladisch. Test data generation for programs with quantified first-order logic specifications. In A. Petrenko, A. da Silva Simão, and J. C. Maldonado, editors, *ICTSS*, volume 6435 of *LNCS*, pages 158–173. Springer, 2010.

[14] W. Grieskamp, N. Tillmann, and W. Schulte. XRT — exploring runtime for .NET architecture and applications. In B. Cook, S. Stoller, and W. Visser, editors, *Proc. Workshop on Software Model Checking (SoftMC 2005)*, volume 144(3) of *ENTCS*, pages 3–26, 2006.

[15] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutorial on modified condition/decision coverage. Nasa/tm-2001-210876, Hampton NASA Langley Research Center, 2001.

[16] C. A. R. Hoare. Proof of correctness of data representation. In F. L. Bauer and K. Samelson, editors, *Language Hierarchies and Interfaces*, pages 183–193, 1975.

[17] J. Huges. Quickcheck: An automatic testing tool for Haskell. http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html.

[18] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek. Cd_{X}: a family of real-time Java benchmarks. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2009*, pages 41–50. ACM, 2009.

[19] J. Kwon and A. J. Wellings. Memory management based on method invocation in rtsj. In *Proceedings, On the Move to Meaningful Internet Systems (OTM) 2004*, volume 3292 of *LNCS*, pages 333–345. Springer, 2004.

[20] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java Modeling Language. In *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, 1998.

[21] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.

[22] Microsoft Corporation. *Code Contracts User Manual*, 2012. http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf.

[23] The Objenesis library website. http://code.google.com/p/objenesis/.

[24] The QuickCheck for Java website. http://java.net/projects/quickcheck/pages/Home.

[25] P. H. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In B. Beckert and C. Marché, editors, *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *LNCS*, pages 138–152. Springer, 2011.

[26] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.

[27] The Simplify project website. http://kind.ucd.ie/products/opensource/Simplify/.

[28] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[29] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. *Software Engineering Notes*, 29(4):97–107, 2004.

[30] A. J. Wellings and M. Schoeberl. User-defined clocks in the real-time specification for Java. In A. J. Wellings and A. P. Ravn, editors, *The 9th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '11*, pages 74–81. ACM, 2011.

[31] D. M. Zimmerman and R. Nagmoti. JMLUnit: The next generation. In *Revised Selected Papers, International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *LNCS*, pages 183–197. Springer, 2011.