A case study of specification and verification using JML in an avionics application

Article · January 2006

DOI: 10.1145/1167999.1168018

6 authors, including:

Peter Schmitt
Karlsruhe Institute of Technology
100 PUBLICATIONS **1,564** CITATIONS
SEE PROFILE

Isabel Tonin
aicas GmbH
10 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Eric Jenn
Thales Group
15 PUBLICATIONS **581** CITATIONS
SEE PROFILE

James Jeffers Hunt
aicas GmbH
30 PUBLICATIONS **507** CITATIONS
SEE PROFILE

A Case Study of Specification and Verification using JML in an Avionics Application

Peter Schmitt, Isabel Tonin, and Claus Wonnemann
Institute for Theoretical Computer Science
Universität Karlsruhe
Am Fasanengarten 5
D-76131, Karlsruhe, Germany
pschmitt|tonin |claus.wonnemann@ira.uka.de

Eric Jenn, Stéphane Leriche
Thales-Avionics
Avenue du Général Eisenhower
F-31036 Toulouse, France
eric.jenn |stephane.leriche@fr.thalesgroup.com

James J. Hunt
aicas GmbH
Haid-und-Neu-Straße 18
D-76139 Karlsruhe, Germany
jjh@aicas.com

ABSTRACT

The literature for deductive formal verification is quite rich; however, very few case studies have been done. The authors present a case study of using deductive formal verification of a navigation system from the avionics domain. Both writing the specifications and their verification with a runtime assertion checker and KeY, a tool using automatic theorem proving techniques for verifying Java programs, are covered.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Specifications; D.2.4 [**Software Engineering**]: Program Verification; J.2 [**Computer Applications**]: Physical sciences and engineering

General Terms

DESIGN, VERIFICATION

Keywords

Design by contract, class invariants, correctness proofs, assertion checkers, Java Modeling Language

1. INTRODUCTION

As computer systems pervade society ever more deeply, higher demands will be made on their design, maintenance, and certification as greater reliance is placed on the delivery of their services. This will be particulary the case for high integrity systems, where failure can cause loss of life, environmental harm, or significant financial loss. The only way to reach the desired level of reliability is to use better tools.

The 4th Workshop on Java Technologies for Real-time and Embedded Systems - JTRES '2006 Paris, France

Formal methods are a prime candidate for meeting these demands. The term formal methods is used to describe a wide variety of different techniques addressing different needs at different stages of system development. *Deductive Formal Verification*, as exemplified by the KeY tool, is particularly suited for ensuring the proper functionality of a program.

Recently there has been considerable interest in using Java for critical systems. This has coincided with the extension of the KeY tool to use the *Java Modeling Language*[10] (JML) to verify Java programs. This case study demontrates to what extend formal specification using JML and the KeY tool can be applied to a real world safety critical application.

2. BACKGROUND

Formal methods may be defined as a collection of notations and techniques for describing and analyzing systems. These methods are *formal* in the sense that they are based on a mathematical theory, e.g. logic, automata, or graph theory. In fields like Civil Engineering, it is well accepted practice that, before building a structure, e.g., a bridge, mathematical models of it are constructed and checked through calculation to ensure that the structure will not collapse. Formal methods' proponents hope that this will some day also be the case in Software Engineering, at least for high integrity applications.

The following quick review of formal methods will thus be biased towards object oriented and especially JML annotated Java programs, even though most of the material remains relevant in a more general setting or could be adapted to it. The discussion further restricts attention, in accordance with the case study, to functional properties of sequential programs. In applications with concurrency, JML specification and deductive formal verification should be complemented with methods analyzing the interaction between different threads. In many cases, the division into sequential and concurrent parts comes quite natural.

JML is based on a widely adopted approach to capturing formal requirements called *Design by Contract* (DBC) as popularized by Bertrand Meyer[12]. DBC uses the conceptual metaphor of a legal contract. The parties in the contract are a "client" that wants to use a software compo-

nent and a "supplier" that provides it. If the client satisfies its part of the contract, referred to as the *precondition*, then the supplier promises to keep his part, referred to as the *postcondition*. Furthermore, there are laws governing all contracts in a given domain, referred to as *invariants*.

DBC is a very general paradigm and there is no restriction on the language used to formulate invariants, preconditions, and postconditions. In fact, almost anything between natural language and Lisp code has been employed for this purpose. JML is a convenient notation for specifying Java programs because it was design specifically for that purpose. Invented by Gary Leavens, JML has now turned into a group effort supported by a growing number of researchers worldwide. JML invariants are attached to JAVA classes and JML contracts to JAVA methods. Preconditions and postconditions are expressed via *requires* and *ensures* clauses. All clauses are directly placed as specially tagged comments into the JAVA code. The syntax of JML expressions closely follows that of JAVA. In addition, JML offers the possibility to declare so called *frame conditions*, i.e., information on what expressions will remain unchanged by a method. In fact, it works the other way round: the *assignable* clause of JML lists all expressions to which a method may assign.

An alternative to JML would have been the Object Constraint Language (OCL), a part of the OMG standard Unified Modeling Language (UML). OCL version 2.0 became available in June 2005[14]. The book, "The Object Constraint Language: Getting Your Models Ready for MDA"[16] provides an accessible introduction to OCL. OCL expressions are attached to elements of the UML model. This higher level of abstraction may have its merits; however, the closeness of JML to JAVA syntax and semantics is easier for the developer to use. JML also provides *model* elements, i.e., variables, fields, and classes, that are not part of the JAVA program, but can be used for specification purposes. In many situations, this provides the same level of abstraction as OCL.

Simply just writting contracts for software can reduce errors. Having constracts can lead to better structured code and expose logical errors. Having contracts available in a well defined format, such as JML, also enables the use of formal analysis techniques.

The simplest technique is *runtime checking*. JML expressions are transformed into JAVA code that throws an exception when an invariant is violated, a method is called in a state where its precondition is not satisfied, or returning from a method when its postcondition is not true. Runtime checking is a form of testing. It does not provide a comprehensive check of a system; it is focused on sampling executions, according to some coverage criteria. Runtime checking can be classified as a *dynamic* technique in the same league with debugging or profiling. Dynamic techniques are used to analyze running programs and require instrumentation.

Static techniques, on the other hand, analyze code without running it. A simple example is checking for type correctness as performed automatically by a compiler. The verification of more complex system properties needs greater abstraction such as symbolic execution. Instead of executing a particular path through a method generating known values of variables, symbolic execution examines all paths with sets of possible values. Also the effect of control flow constructs, like loops, are covered at a symbolic level abstracting away from the number of loop iterations. Static program analysis techniques differ in their degree of abstraction, striking a balance between the precision of the results and the effort needed to obtain them. A very popular tool, that also works with JML assertions is the Java Extended Static Checker, ESC/Java[7]. As with most static analysis tools, ESC/Java may produce false warnings; however it may also produce false negatives. This is the price to be paid for increasing efficiency. Despite this, ESC/Java is a powerful and useful tool for finding many classes of errors; however, it can not give the level of guarentees required by safety critical applications.

Another approach to static analysis is *Deductive Program Verification* whose symbolic execution faithfully reflects the semantics of the programming language. For example, suppose one wants to verify that after execution of a loop the property P holds true. Given that the JML annotations specify a loop invariant I, the original proof obligation can be fulfilled by verifying that I is true on entering the loop, that I is again true after each execution of the loop body, and that on termination of the loop, I implies P. An integer expression t can be provided via JML's *decreases* clause to simplify the proof. Thereby, proving termination of a loop is reduced to verify that t only takes positive values and strictly decreases when symbolically executing the loop body. Tools for deductive program verification differ on the internal presentation of the proof obligations, on the coverage of the target programing language and, on the user interface. The precision of the results obtained via deductive verification has to be paid for by providing additional annotations and by increased verification effort.

Dynamic and static analysis of the contract are in general complementary. Dynamic tests have the advantage of providing tangible, auditable, evidence of the system behavior. Verifying it statically through the Deductive Program Verification technique, in turn, has the advantage of exploring all possible program executions, which is practically infeasible through testing. Additionally, negative properties, such as that the program is free of some anomalous behavior, are particularly difficult to test and are better verified with static analysis.

3. GOALS

Thales Avionics is investigating the potentials of JAVA as a technology for the implementation of safety critical realtime systems in avionics which must comply with official certification standards. The use of formal methods is considered as a means to improve the software integrity and to make the development process more efficient. These techniques are not in general usage in the development process of avionics software today, however, Design by Contract and formal methods are repeatedly recommended in the handbook of the *Object-Oriented Technologies in Aviation (OOTiA)* program[15]. This document is an early step towards the next release of the official requirements specification, RTCA/DO-178C, for airborne software products.

The general goal of this case study was to investigate the feasibility and benefits of formal software development techniques in an industrial context. The participants assessed the difficulties of writting specifications and verifying code. They also examined the contribution these techiques made to improving software development in the avionics domain.

The target software for the case study is an implementation of a flight management system in Java, the Java Flight

Manager, JFM. It was developed by Thales Avionics primarily as a platform supporting rapid prototyping of new features. A further goal was to investigate the benefits and risks of using object-oriented technologies, and particularly the JAVA programming language. The JFM provides the main features of a flight management system; but, being an experimental prototype, it does not reach the standards of a production system in terms of functional range, compliance to applicable norms, qualification, validation, and documentation. Nonetheless, it provides a working example of an industrially sized software application.

Normally, one would not want to specify a system after it has been implemented, since specification is regarded to be a part of the design phase. This is an imperfection the authors had to accept; however, the study is still useful. Considering the extend of change the introduction of software development using DBC would constitute, it was certainly worth the effort of performing the study so that the costs and benifits of DBC with Java could be better understood. Additionally, it has made implicit assumptions about the implementation of JFM explicit providing a better basis for future reimplementation.

Due to time restrictions, the goal was not to specify every single class. The idea was to cover all the major phenomena occurring in the application to ensure that the assessment would be realistic. Emphasis was put on invariants, as these capture the main characteristics of a class or an interface. Essential method contracts were written as well.

The case study also included testing via runtime checking that the implementation indeed complies with the formal specifications. For the test runs, a sample set of flight routes was computed by the JFM with assertion checking turned on. When an exception was thrown due to violated assertions, the developers determined whether these originate from an erroneous implementation and how such problems could be fixed.

The University of Karlsruhe spend considerable effort in developing the KeY system to improve the usability of deductive verification for industrial applications. An additional aim of the case study was to assess the progress made towards this goal by formally proving the correctness of the JFM implementation with respect to the JML contracts. The team expected to gain insights into the effort required to verify production code.

4. APPLICATION

A flight management system assists the pilot in navigating and managing an aircraft. Among other things, it provides services to construct and maintain a trajectory defining the path between given departure and arrival positions. It is part of the on board navigational equipment operated by the cockpit personnel. Given its importance for flight safety, the system belongs to the second highest category (out of five) of safety critical airborne software according to official certification standards. The portion of the Java Flight Manager (JFM) treated in this case study is the lateral module, which is responsible for the construction of a trajectory in the horizontal plane. For simplification purposes, altitudes or outer influences such as wind were not taken into account.

The basic input for the JFM is a route defined in terms of smaller sections, referred to as legs. In order to define a route with a comparably small number of named geographical positions ("fixes"), a leg is characterized using the concepts of *paths* and *terminators*. Roughly speaking, a *path* defines a segment of the flight in a certain direction which is terminated by a condition defined in the *terminator*. For instance, a leg might be defined by a path which bears a certain course or a fixed destination position; the terminating condition could be the arrival at an intersection point with a specified bearing to a navigational beacon or another fixed geographical location. The majority of legs does not contain a fixed start position, as this corresponds to the end position of the preceding leg and is determined during the successive construction of the flight plan.

The concept of *procedures* is used to subdivide a route into several distinct phases above the leg level. A typical route groups the legs into procedures according to phases of the flight, either the *departure*, the *enroute*, the *arrival* or the *approach* phase. JFM's internal representation of a route is a hierarchically organized tree, whose inner nodes are objects representing the different procedure types and whose leaves are objects modeling legs. A depth first search collecting the tree's leaves delivers a sequence of leg objects corresponding to the actual route.

The task of a flight management system is to transform such a schematically defined route—its sequence of legs—into a sufficiently detailed trajectory which can be used directly for guiding an aircraft. During the construction of a trajectory, JFM takes into account further constraints, such as efficiency and fuel consumption, the laws of physics, e.g., when computing an aircraft's turn radius at a given speed, and passengers' comfort.

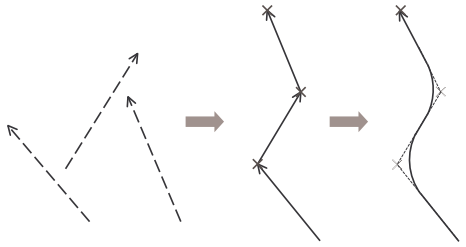

Figure 1: An outline of a trajectory's refinement stages.

JFM uses three different representations of a route successively in the construction of a trajectory, each being a refinement of the preceding. Initially, the route is specified as a sequence of legs; each of these legs is defined by a path and a terminator. In a first step, the JFM calculates all transition points between subsequent legs, i.e. the conditions formulated in the paths and terminators are successively instantiated to specific geographical positions marking the start and end points of the legs. These intersection points are referred to as *pivots* and represent a route made up of straight lines connected by these points.

Although a sequence of pivots already defines a fixed route, it is insufficient for use in aircraft navigation, since it does not account for its maneuverability. Thus, the final representation of the route—the trajectory—is built by reshaping the sequence of straight lines in such a way that it can be directly used for the guidance of an airplane. Roughly speaking, sharp edges are smoothed in keeping with an aircraft's agility. Fig.1 outlines the trajectory computation process

with respect to these three refinement stages.

While the calculation of the pivot points can be implemented straightforwardly, the construction of a trajectory between two pivot points is more complex. This second refinement step is implemented as an attempt to subsequently apply different strategies, until one of them succeeds in delivering a trajectory which meets all requirements. The order of strategies is fixed, beginning with the simplest one (in terms of the length of the path and the number of turns), up to a strategy that constructs a valid trajectory in any circumstance. The construction of the trajectory by means of different strategies is done on a per leg basis, i.e., a strategy attempts to construct a trajectory for the path between two pivot points. More advanced strategies might skip intermediate pivots and try to construct the trajectory between pivots which do not immediately succeed each other.

The JFM's functionality to construct a lateral flight plan is implemented in the `lateral` package, which consists of some seventy classes. These implement the different route representations and algorithms to construct the horizontal part of a trajectory from a given sequence of legs. The driving class for the representation of a lateral flight plan is called `LateralProfile`. Provided with a route object as obtained from a navigational database, its constructor triggers the construction of the trajectory according to the process outlined above.

5. TOOLS

For the JML part of the case study, the *Common JML Tools* distribution from Iowa State University was used. This includes a syntax checker (`jml`), a JML-enabled JAVA compiler to be used for runtime assertion checking (`jmlc`), and a tool to derive an HTML documentation of the software system and its specification from the source code (`jmldoc`).

Currently, `jmlc`[4] is the only implementation of a JML-enabled JAVA compiler, which augments the compilation of a JML-annotated JAVA program by inserting additional instructions for runtime checking. A JAVA program translated with `jmlc` can be run with a standard JAVA virtual machine using the JML class library. The compiled byte code transparently checks JML annotations at runtime, i.e. as long as no assertion is violated, the program behaves as if it was translated with a standard JAVA compiler.

Once a formal JML specification for a software system exists, there is no additional effort necessary to apply runtime assertion checking. Since the `jmlc` tool accepts all legal JML and JAVA code, no modification of the existing code base is necessary.

Although `jmlc` behaves as a standard JAVA compiler in the absence of JML annotations, its runtime performance and that of the generated programs will usually limit its application to debugging and testing purposes. Table 1 provides comparisions of the runtime performance of the JML annotated JFM for the computation of different routes, compiled with `jmlc` and `javac`, respectively. The execution time of code generated by `jmlc` is at least an order of magnitude greater than that produced by `javac`—a hugh ratio, even taking into account that `jmlc` uses the JML annotations to generate additional code over that of `javac`.

Although every legal JML code is accepted by `jmlc`, not every JML statement can be transformed into executable code, i.e. some assertion violations might remain undetected. This shortcoming applies particularly to quantified

# LEGS	RUNTIME `JMLC` (MS)	RUNTIME `JAVAC` (MS)
5	4493	227
7	6751	265
18	46412	285
24	122126	316
32	216660	397
33	309465	370
34	362443	402
37	391441	413

Table 1: Some example runtime figures for route computations in the JFM, compiled with `jmlc` and `javac`, respectively.

expressions, frame conditions, and the enforcement of invariants. Running code for these kinds of constructs would be prohibitively expensive.

The current implementation of `jmlc` can transform quantified expressions into executable code, when the range of the quantifier can be restricted to a finite set. *Assignable clauses* are ignored by `jmlc`. However, tools which are capable of statically validating them (at least partially) exist, e.g., the CHASE static checker for JML's assignable clause[3]. For practical reasons, `jmlc` does not fully adhere to the official semantics, the *visible state* semantics, ofJML invariants. It behaves closer to the *observed state* semantics. This point will be revisited in Sect.6.

The KeY tool[1], a verification tool jointly developed at Karlsruhe University (Germany), Chalmers University (Gothenburg, Sweden) and the University of Koblenz (Germany) was used for deductive program verification. The main component of the KeY system is the KeY prover, a semi-automated prover with a comfortable graphical user interface. It offers two front ends, one for *Object Constraint Language* (OCL) and one for JML specifications. OCL or JML specifications are automatically translated into proof obligations in JavaDL. The internal logic of the KeY prover is an instance of dynamic logic called Java Dynamic Logic (JavaDL)[2]. The JavaDL calculus covers 100% of Java Card, version 2.1 and additionally supports some Java SE features such as multi-dimensional arrays.

Verification with KeY proceeds by symbolic execution of the Java program being analyzed. This process is closely related to a programmer's understanding of the language. Since a stage in the proof corresponds to a stage during the execution of the program, the process lends itself to interactive verification. Special care has been taken to make user interaction, when it is necessary, as easy as possible. User support includes flexible selection of proof strategies, the possibility to save and reload (partial) proofs, to rerun proofs on changed proof obligations, built-in decision procedures for integers, drag-and-drop interaction, and a clearly laid-out display of the proof.

6. SPECIFICATION AND TESTING

The formal specification of the JFM roughly followed the construction process of lateral flight plans as described above, beginning at the central class `LateralProfile`, which represents the lateral part of the trajectory. The main approach was to define an abstract model for a class, which reflects

the basic entities needed for its characterization. Subsequently, invariants relating the fields of the abstract model and method contracts were developed. About forty classes of the JFM's lateral module have (at least partially) been specified and tested using runtime assertion checking in a period of about three months. In this section we will address several interesting issues that came up in writing the specification. Additional details can be found in "Formal Specification and Verification of Avionics Software"[17].

6.1 JML Model Library

The specification of the JFM followed a model based approach, using JML's built-in support for the creation of abstract models through model variables[5]. Although there is a class library for this purpose delivered with the JML distribution, it proved to be too cumbersome to use. Instead the team employed built-in types and the data structures from the standard JAVA class library. These were adequate for the study. The level of abstraction was high enough to let the implementor freely choose the data organization required. The types from the JML model library often make a specification awkward and non-intuitive, as common mathematical concepts and relations have to be expressed in a programming-style notation using classes, interfaces, and method calls. In this respect, they offer no advantage over standard JAVA classes, with which the programmers and designers are more familiar. Furthermore, a specification using the JML modeling classes tends to be difficult for formal verification, since this library forms a self-contained set of types, whose specifications heavily depend on each other and easily make a proof obligation more complicated than necessary.

6.2 Liskov's Substitutability Principle

The Liskov's Substitutability Principle, (LSP), has been proposed[11] to support modular reasoning about programs featuring subtyping and dynamic dispatch. Let m_A be a method in a class A that is overridden by the method m_B in the subclass B of A. LSP requires that the precondition of m_A is at least as strong as the precondition of m_B and that the postcondition of m_B is, in turn, at least as strong as the postcondition of m_A. The OOTiA handbook names LSP explicitly as a desired feature[15, §3.3.6.3] The JML semantics of contract inheritance also complies with it[9, §1.4][6].

Strict enforcement of the LSP, however, sometimes gets in the way of easy use of inheritance. In the JFM case study, the class representing a node in a tree structure, `Tree`, was written as a subclass of the JAVA library class `ArrayList`. This provides a convenient way to maintain the children of a tree node by reusing the fields and methods of `ArrayList`. The precondition for the `add`-method from `ArrayList` requires the object to be added to be of type `Object`, while the precondition for the `add`-method in the subclass `Tree` requires it to be an instance of `TreePart`—A clear violation of LSP. In this case, encapsulating the `ArrayList` in the `Tree` is a better solution. Encapsulation also avoids exposing unneeded methods.

Problems like these can occur in many cases where general types are used. but care should be exercised in resolving them. Some suggest relaxing the precondition of the overridden method and to make it throw an `IllegalArgumentException` when the parameter is not of the restricted type. Though this solution may apprear to abide by LSP, the subclass can not be used in place of its parent. Another suggested resolution is the use of *abstract preconditions*[13, §16.1]. In the above example, this would amount to introduce a Boolean field, call it `paramOK`, that as a field in `ArrayList` would be true of any object and as field in `List` would be true for all objects of type `TreePart`. This seems to be a powerful feature, as it has the capability of arbitrarily avoiding the inheritance restrictions imposed by LSP; however, using parameterized types from Java 5 generics is a sounder solution.

In general, care should be taken to distinguish subtyping from reuseing implementation. The first is a valid used of subclassing. The second should be done via another mechanism such as delegating to an encapsulated object or using parameterized types.

6.3 Pure Methods

Besides fields, JML also allows the use of methods from the surrounding JAVA code in annotations. Since evaluation of annotations should always be free of side effects this is only allowed for methods that have been declared *pure*. A pure method must terminate, may only assign to locations which were not allocated when the method started execution, or are local to the method. This definition of purity turned out to be overly restrictive in the course of the JFM specification, since it also prevents "benevolent" side effects, i.e., side effects that somehow change the state of the program, but not its functional behavior.

The method `getObject` from JFM's database module returns a database entry, which is referenced by a `RecordLocator` instance, which is passed as an argument. It first attempts to retrieve the desired object from a local cache; in case of a miss, the database entry is read from the disk, which is obviously a more costly operation. The disk reader's `readRecord` method stores the retrieved object in the cache.

```
private static DbObject getObject(RecordLocator rl)
{
  DbObject o = (DbObject)cache.get(rl);
  if (o == null)
    {
      reader.readRecord(rl);
      o = (DbObject)cache.get(rl);
    }
  return o;
}
```

Although the `getObject` method seems intuitively pure (it just returns an object), it violates its definition by assigning to an already existing cache object.

All methods involving I/O are considered to have side effects as well and thus cannot be declared pure. In the JML case study, this particular restriction prevented a precise specification of the constraints on the lateral module's initial input: a sequence of legs stored in a navigational database.

6.4 Invariants

Invariants can uncover hidden errors. For example, a bidirectionally linked tree was used to represent a route. The following invariants were specified:

- for all child objects, the `parent` reference must point to `this`, and

- the object indicated by `parent` (if not `null`) must have a child reference to `this`.

Testing this invariant revealed that the implementation of `Tree`'s `clone` method returns a copy with a all parent fields set to null, and therefore a tree structure that does not comply with these invariants. This is a potentially dangerous behavior, as other modules might implicitly rely on a tree being "correctly" linked. There is indeed an example from the JFM where this assumption has obviously been made. The class `RouteTransformer` defines static methods for the manipulation of tree structures, such as the `insert` method to add additional legs to an existing route:

```
public static void insert(Route route, int from,
                          int to, List fixes)
{
  ...
  int index = fromLeg.getParent().indexOf(fromLeg);
  fromLeg.getParent().add(index + 1, proc);
}
```

A new `Procedure` instance is inserted to the right place by traversing the tree "bottom up", using the `getParent` method that returns the parent field, which is likely to have the value `null` if the tree was cloned. Although this is a rather obvious defect which would have been detected by thorough tests, it could have been prevented by a clear statement of `Tree`'s properties and a consequent check of its methods' compliance with the invariants.

6.5 Code Changes

The study revealed places where the code could be improved. The implementation of the JFM was modified slightly in some areas to ease specifications and to make the design more consistent. The changes were minor and do not affect the overall architecture.

Methods were modified which inherit from pure methods but did themselves have side effects. The `Tree` class, e.g., features an inner class `PartsIterator` which implements the `ListIterator` interface to traverse the tree. The `hasNext` method is declared pure in the specification of the `ListIterator` interface which is part of the JML distribution. Yet, the implementation of the `hasNext` method did assign to non-local locations. This in itself violates the behavioral subtyping principle, but, also had further consequences. The implementation of the method `next` relies on `hasNext` to set the "cursor position" of the iterator to the appropriate position. In its specification, one needs to refer to `hasNext`, but this prevents the code from being compiled with the JML tools if `hasNext` is not pure. This problem could be easily resolved by making `hasNext` side-effect free and by introducing a new method `goToNext` which imitates the original behavior of `hasNext`.

Placing much of an object's functionality in the constructor enables stronger invariants and less complicated reasoning about instances. The properties expressed by a class's invariants can be concluded from the mere existence of an instance in all visible states. Therefore, some constructors were changed in this manner, when this was reasonable from the characteristics of the class and implementable without major impact on the JFM's overall design.

An example is the constructor of the `Pivot` class. In the original implementation, it merely assigned its argument values to local references. Thus the invariant for an existing `Pivot` object could merely described it as a container for other objects. Whenever a `Pivot` object was created in the JFM, however, a subsequent method call triggered the computation of the intersection between two legs. This method was inlined into the constructor, enabling invariants concerning the intersection point thus better reflecting the characteristics of a pivot.

A redesign of the route representation as a tree structure to avoid the problem with LSP mentioned above was suggested. This would have led to a bigger effort since several modules of the JFM rely on properties of the current design. A decision was thus postponed, since also the details of a new design had still to be worked out.

7. VERIFICATION AND REFINEMENT

Once the specification was in place and some testing had been done, some parts of the code were formally verified for correctness with respect to the specifications using the KeY tool. A method from the avionics application was chosen based on its relevant characteristics for this study. The work performed until its verification is related here.

The method chosen was taken from the trajectory construction part of the JFM called *mergeProcedures*. Generally speaking, during the construction process of a trajectory, the JFM uses three different representations of a route, where each is a refinement of the preceding (see Figure 1). Initially, there is a route defined by several procedures (sequence of legs) obtained from a data base. Then, with the *mergeProcedures* method, the JFM calculates intersection point between subsequent legs (*pivots*), and merges the procedures based on them.

The first step for verifying a method is to analyze it carefully in order to write a clear and correct contract for it. The analysis made for the *mergeProcedures* method is shown in the Figure 2. The top of the figure shows the general view of the method: it receives two procedures and merges them producing a result procedure containing some legs from the first and some legs from the second procedure. The way the procedures are merged is shown at the bottom of the figure.

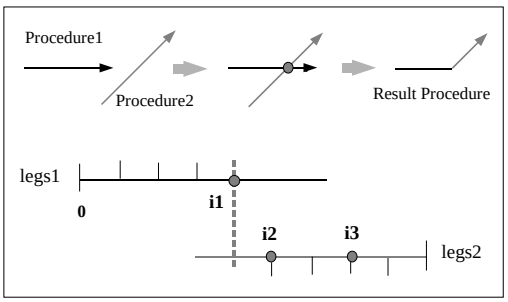

Figure 2: The mergeProcedures method analysis.

legs1 and *legs2* represent the legs from the procedures 1 and 2 respectively (in this implementation *legs1* and *legs2* are arrays).

The result procedure is built with legs from *legs1* (from the beginning until the point *i1*) and from *legs2* (from the point *i2* until the end).

The calculation of the points *i1* and *i2* determines which legs will be part of the result procedure.

These points are calculated in the following steps.

- *i1* represents the last leg from *legs1* to be part of the result procedure. It is found by searching for an intersection point between *legs1* and *legs2*. Each leg from *legs1*, starting with the first, is tested to see whether it exists in *legs2* or not. If one exists, this leg becomes *i1*. If no intersection was found, the last leg of *legs1* becomes *i1*.
- *i2* represents the first leg from *legs2* to be part of the result procedure. Its calculation is performed within the calculation of point *i1*. When an intersection is found, the first leg of *legs2* after the intersection leg becomes *i2*. If no intersection is found, the first leg from *legs2* becomes *i2*.
- *i3* represents a limit for the search for intersection performed in *legs2*. It is the first occurrence of a leg of the type *runway*.
- The search for intersection in *legs2* is performed backwards (from the *i3* point to the beginning) in order to find the last intersection point before the runway.

The JML contract written for this method is shown in Figure 3 and can be read as follows.

- The points *i1*, *i2*, *i3*, are written as indexes of the array of legs (going from 0 to the length of the array minus 1).
- Lines 2 and 3 state the preconditions: both legs from the argument must be neither null nor empty.
- Line 4 declare that the only class fields assigned by this method are the i1 and i2 (auxiliary fields).
- Lines 5 to 9 gives the first post condition. It states the two mutually exclusive values for the fields *i2* and *i1*.
  - The field *i2* is the first leg of *legs2* (0) and the field *i1* is last leg of *legs1* (legs1.length - 1), indicating the case where no intersection was found.
  - The field *i2* is some leg after the first leg *legs2* (it can be after the existing legs, representing the case where an intersection is found with the last leg of *legs2*). The field *i1* is some leg between the beginning and the end of *legs1*. The *legs1* position indicated by *i1* is not null and this position is equal to the previous *legs2* position indicated by *i2*.
- Lines 10 and 11 give the second post condition: all legs from the beginning of the result procedure until the leg indicated by *i1* come from *legs1*, keeping the same index (order is respected).
- Lines 12, 13, and 14 give the second post condition: all legs of the result procedure, from the leg after the point indicated by *i1*, until the end, come from *legs2* (order is respected).
- Lines 15 and 16 give the last post condition: all legs from *legs2*, from the point *i2* to the end, are member of the result procedure, in the same order, starting in the result procedure at the leg after the point *i1*.

```
/*@ private normal_behavior
  @ requires legs1 != null && legs2 != null &&
  @  legs1.length > 0 && legs2.length > 0;
  @ assignable i1, i2;
  @ ensures (i1 == legs1.length - 1 && i2 == 0 ||
  @  (i2 >  0 && i2 <= legs2.length &&
  @   i1 >= 0 && i1 <  legs1.length &&
  @   legs1[i1] != null &&
  @   legs1[i1] == legs2[i2 - 1]))  &&
  @  (\forall int i; 0 <= i && i <= i1;
  @    \result[i] == legs1[i]))     &&
  @  (\forall int i;
  @     i > i1 && i < \result.length - 1;
  @    \result[i] == legs2[i + i2 - i1 - 1]) &&
  @  (\forall int i; i2 <= i && i < legs2.length;
  @    \result[i1 + 1 + (i - i2)] == legs2[i]);
  @*/
private
  Leg[] mergeProcedures(Leg[] legs1, Leg[] legs2)
{
```

Figure 3: JML contract for mergeProcedures.

The next step after having written the contract was to use the KeY tool to prove the correctness of the implementation according to the contract. The first attempt did not succeed at all. One reason was that writing such a contract was not so simple so its first versions were incorrect. Another reason was the method complexity. In fact, this method is a quite interesting choice for a case study in verification. Even though it is not so large, it contains five loops, two of them nested. It is well known that loops can make any verification at least challenging (when it does not make it impossible). The final reason is the proving process itself. Formally proving a method is an interactive process that can require successive refinement in both the code and the contract.

In order to reduce the complexity of the verification to be performed, the method was redesigned and divided in four parts: (i) seeking the index of the first runway (one loop on *legs2*), (ii) finding the intersection (two nested loops, to each element of *legs1*, iterate backwards on *legs2* trying to find the common leg), (iii) finish merging (one loop on *legs1* and one loop on *legs2* to copy the legs to the result procedure), and (iv) a new mergeProcedures redefined for calling the other methods. The new *mergePrcedures* kept the same contract as the old one and the other new methods had to have new JML contracts written for them. The new *mergePrcedures* code and contracts are shown in Figures 4, 5, 6, and 7. Full examples are provided in a related paper on bounds checking at this workshop[8].

```
private
  Leg[] mergeProcedures(Leg[] legs1, Leg[] legs2)
{
   i1 = legs1.length - 1;
   i2 = 0;
   int i3 = indexFirstRunway(legs2);
   if (i3 > 0)
      findIntersection(legs1, legs2, i3);
   return finishMerging(legs1, legs2);
}
```

Figure 4: The new mergeProcedures method.

The next step after dividing the original code and writing the contract for each of these new pieces of code was to prove each one separatedly. The verification of each piece of code was completed successfully. The difficulties faced were mostly due to the loops. In KeY, a way of verifying a loop is writing a kind of contract comprising three items, see also the explanations in Sect.2.

7.1 Loop invariant

A loop invariant is any relation (or formula) that remains true during the whole loop execution (including immediately before starting the loop and immediately after finishing the loop). Thus many formula would be possible candidates for a loop invariant. However, in practice, one has to chose the invariant that leads to the postcondition to be proven. One should keep in mind that the only information that the KeY tool has after symbolically executing the loop is the information contained in the loop invariant. Consequently, the loop invariant has to be as strong as the method postcondition requires.

7.2 Assignable clause

The assignable clause is the only loop clause adopted by KeY that is not yet in JML. A similar clause is already being used by other groups, so it is just a matter of time before this clause officially becomes part of JML. It decreases the complexity of the loop proof by declaring the variables that are modified by the loop body.

7.3 Decreases clause

A decrease clause is a formula that must decrease by at least one in each step of its loop and remains positive during execution of the loop.

```
/*@ private normal_behavior
  @ requires legs2 != null && legs2.length > 0;
  @ assignable \nothing;
  @ ensures \result >= 0 && \result < legs2.length;
  @*/
private static int indexFirstRunway(Leg[] legs2)
{
```

Figure 5: JML contract for indexFirstRunway.

Formalizing loops can be as difficult or more than formalizing the whole method and, as said before, interdependent. The specification for the first loop of the *finishMerging*, the one that adds the legs from *legs1* to the result procedure, is shown in Figure 8. In this specification one can see, for instance, that the third and fourth lines make the same statement as the seventh and eigth lines of the methods contract (Figure 7).

After many interactive steps tuning contracts and code, the three method were finally proved. The next step was to prove the new *mergeProcedures* method. The idea for this method was to use the already proven contracts from the other methods. Using contracts allows the tool to avoid symbolically executing a method, since it may using its contract instead. For that, when performing a proof using contracts, the user is required to show that the preconditions of the desired contracts are met in order to use its postcondition in the proof body.

Considering that each method was already proven, the proof of the new *mergeProcedures* was expected to be very

```
/*@ private normal_behavior
  @ requires legs1 != null && legs1.length > 0 &&
  @          legs2 != null && legs2.length > 0 &&
  @          i3 > 0        && i3 < legs2.length;
  @ assignable i1, i2;
  @ ensures i2 == 0 && i1 == legs1.length-1    ||
  @        (i2 >  0 && i2 <= i3 + 1            &&
  @         i1 >= 0 && i1 <  legs1.length      &&
  @         legs1[i1] != null                  &&
  @         legs1[i1] == legs2[i2-1]);
  @*/
private
  void findIntersection(Leg[] legs1, Leg[] legs2, int i3)
{
```

Figure 6: JML contract for findIntersection.

```
/*@ private normal_behavior
  @ requires legs1 != null && legs1.length > 0 &&
  @          legs2 != null && legs2.length > 0 &&
  @          i1 >= 0 && i1 < legs1.length      &&
  @          i2 >= 0 && i2 <= legs2.length;
  @ assignable \nothing;
  @ ensures (\forall int x; x >= 0 && x <= i1;
  @            \result[x] == legs1[x]) &&
  @  (\forall int y;
  @    y > i1 && y < \result.length - 1;
  @    \result[y] == legs2[y + i2 - i1 - 1]) &&
  @  (\forall int z; z >= i2 && z < legs2.length;
  @    \result[z - i2 + i1 + 1] == legs2[z]);
  @*/
private
  Leg[] finishMerging(Leg[] legs1, Leg[] legs2)
{
```

Figure 7: JML contract for finishMerging.

simple. However, the proof could not be performed. Looking closely revealed what had happened. The postcondition of the *finishMerging* contained the part
`\forall int x; x < i1; \result[x] == legs1[x]`
while the corresponding part for *mergeProcedures* read
`\forall int x; x =< i1; \result[x] == legs1[x]`.
Trying to prove the strengthened postcondition for *finishMerging* revealed that its implementation was in fact incorrect. Once the problem had been identified, it was corrected (code and contract) and the proof process was restarted. However this time, first the contracts of the called methods were proved to indeed suffice to guarantee the correctness of *mergeProcedures*. After that, the correctness proofs of the

```
/*@ loop_invariant
  @  i >= 0 && i <= i1 + 1 &&
  @  (\forall int x; x >= 0 && x < i;
  @           merge[x] == legs1[x]);
  @ assignable merge[0..i1], i;
  @ decreases (i1 - i + 1);
  @*/
  while (i <= i1)
  {
    merge[i] = legs1[i];
    i++;
   }
```

Figure 8: A loop specification for KeY.

Method	Nodes	Branches
mergeProcedures	1155	18
indexFirstRunaway	1513	26
findIntersection	3407	35
finishMerging	23306	109

Table 2: Statistics on KeY proofs.

called methods were performed independently.

The results of the proofs performed are shown in the Table 2. In this last attempt, knowing well the methods and contracts, the four proofs could be performed in about 6 hours. This is a very short time considering the time spent for proofs at the beginning of this case study. Three main reasons contributed reducing the time: (i) the tool was improved during the case study becoming able to handle more automatically, (ii) the experience gained helped knowing the "tricky" parts of the proofs, and (iii) the contracts were corrected during the many interactions. The time spent on writing the contracts (including the time of analyzing the application) was not measured, but it probably consumed a half of the whole time spent in the case study. It was also noticed that experience counts considerably for writing good and correct contracts, and for shortening the time required for writing them.

8. LESSONS LEARNED

Several lessons about the use of formal specification were learned from the JFM case.

- The experience gained from specifying the JFM suggests that thinking about invariants brings a great deal in getting a precise understanding of the system's behavior. The definitions of invariants and the subsequent check for the implementation's compliance with them has revealed several errors in the course of this case study.
- Formal specifications convey an unambiguous description of the software's behavior and provide a clear guideline for the implementors and users of a software component. The aspect of clarification is particularly true for the development process of the specifications themselves; the task of formally expressing a module's properties requires thorough reflections on its characteristics, responsibilities, and relations to other parts of the software system, and arguably enforces a better design.
- Basic features of the Java Modeling Language, such as simple method contracts, are immediately accessible to programmers used to Java, although more complex constructs require some training to use them efficiently.
- Getting the specifications right is difficult—apparently obvious constraints might not hold in certain situations. This is even true if invariants are formulated in natural language.
- Strict enforcement of the LSP restricts the use of subclassing but may result in better code. Delegation to enclapsulated objects or parameterized types can replace inheritance in most of these instances. A categorical adherence to the LSP might result in some increase in code size, but the resulting design will be safer.
- JML's close resemblance to a programming language makes it easy to write extensive specifications that are too verbose, as the language lacks a symbolical notation and requires frequent type casts. This approach ensures in principal the executability of assertions, but was found to be a serious handicap for the usability of the JFM's specifications as a means of documentation, due to restricted readability and difficult maintainability.
- At some points, informal specifications seemed more suitable than formal ones; for instance, the property of sortedness is conveyed more easily by a few words than by an extensive clause. Textual specifications and other means of informal documentation were particularly found to excel in the description of a system's overall properties, i.e. in the expression of global, class spanning relations, however, the possibility of verification is lost.
- The accessibility to software engineering tools is a further benefit of a formal specification which comes at no additional cost. Runtime assertion checking was found particularly useful as an easy means of validation for the specifications themselves.

Regarding the verification of the mergeProcedures methods, the following conclusions can be drawn.

- The most important lesson learned is that deductive verification is by itself a complex and time demanding task to perform. Therefore one should do what is possible to make it simpler.
- Keep your code simple! Writing clear and simple code makes verification simpler.
- When possible, contracts and code should use similar structures/formulas. The task of showing that two differently worded constructs are equivalent increases the verification work to perform.
- Loops should be written to be as simple as possible but still have invariants as strong as required by the postcondition.
- Composition of methods (as the new *mergeProcedures* method presented in Section 7) seems to be a good solution for breaking down the complexity of a method. A composition of methods can use the contracts of the called method for its proof without unfolding their implementation code;
- Verification seems to be better done top down. First show that the contracts of called methods are sufficient to establish correctness of the composed method, then show that the implementation of the called methods satisfy their contracts;
- Library functions are typical methods used in composition. One should think about how to write general contracts for them. One way can be trying to have several different examples for testing the contract (or writing a contract based on examples). Another is trying to give as much information in the postcondition as possible and making the precondition as simple as possible, so the method is as applicable as possible.

- The use of *interfaces* and *superclasses* can also make verification simpler. Contracts can be written for its methods and these methods can be used when the interface/superclass is referenced (instead of evaluating each possible implementation of the method). However in order to keep correctness, each implementation of the method must be correct in accordance to this contract written in the interface/superclass.

9. CONCLUSION AND FUTURE WORK

The case study presented here was an important step toward the use of deductive formal verification in industry. Using a realistic example from an industrial context to gain insight into the problems of writing design by contract specifications and of deductively verifying these contracts provided useful insights in the current state and where improvement should be made. Both parts of the case study were done from a user's point of view by people who had experience with logic and theorem proving in general, but had no training with the particular languages and tools used.

One of the results to be stressed is, that it is not so easy to get contracts right. This has nothing to do with formalizing contracts in a particular formal language and applies as well to contracts in natural language. This clearly calls for increased efforts to develop a methodology for writing contract. One could think of complementing the emphasis on invariants, as was done in this case study, by an approach that concentrates on preconditions and postconditions. Stronger guidance for these might be obtained from a use case requirements analysis.

Regarding the results of the case study on deductive verification it is worth observing that it is possible with a considerable but not forbidding overhead. Design for verification is a crucial factor for the success of this method. Ways to improve deductive verification are enhanced automation of the proof process and increased integration with traditional program analysis methods.

Though there are still theoretical problems to solve, the bulk of the work needed to bring formal verification into industrial settings is engineering work. Improving the usability and applicability of the tools could broaden its applicability tremendously. Even though formal methods are not a panacea for all the ills of software development, one can surely benefit from their use.

10. ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support from the European Union through the IST framework as partners in the HIJA project. Many thanks also go to Richard Bubel from the KeY-team in Karlsruhe for his unrelenting, competent support and insights.

11. REFERENCES

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041, pages 6–24. Springer-Verlag, 2001.

[3] N. Catano and M. Huisman. Chase: A Static Checker for JML's Assignable Clause. Technical report, INRIA Sophia-Antipolis, France, 2003.

[4] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, April 2003.

[5] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model Variables: Cleanly Supporting Abstraction in Design By Contract. *Software – Practice and Experience*, 35(6):583–599, May 2005.

[6] K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping Through Specification Inheritance. Technical Report TR 95-20c, Department of Computer Science, Iowa State University, March 1997.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin*, pages 234–245. ACM Press, 2002.

[8] J. J. Hunt, I. Tonin, P. H. Schmitt, and F. Siebert. Provably correct loop bounds for realtime java programs. In *Proc. The 4th Workshop on Java Technologies for Realtime and Embedded Systems (JRTES'2006) Paris, France*, October 2006.

[9] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. Technical Report TR 03-04a, Department of Computer Science, Iowa State University, 2004.

[10] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, November 2004. Draft revision 1.98.

[11] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[12] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[13] B. Meyer. *Object-Oriented Software Construction, second edition*. Prentice Hall, 1997.

[14] Object Modeling Group. *Object Constraint Language Specification, version 2.0*, June 2005.

[15] Object Oriented Technology in Aviation Program. Handbook for Object-Oriented Technology in Aviation (OOTiA), January 2004. Available at `http://www.faa.gov//aircraft//air_cert/ /design_approvals//air_software//oot//`.

[16] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, August 2003.

[17] C. Wonnemann. Formal specification and verification of avionics software. Diplomarbeit, Department of Computer Science, Universität Karlsruhe, 2006.