Agile Specifications

Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson

MIT Computer Science and Artificial Intelligence Laboratory

{drayside, aleks, kuat, gdennis, dnj}@csail.mit.edu

Abstract

Traditional formal methods and modern agile methods are separated more by limitations of current technology than by fundamental intellectual differences. A *mixed interpreter* that executes mixed programs, comprising both declarative specification statements and regular imperative statements, might bridge the gap. This paper explores how such an interpreter might be used, showing by example how it might support a variety of development activities.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Design, Languages

Keywords formal methods, agile methods, specification statement, refinement calculus, test-driven development

1. Introduction

Our departure from tradition is a small one: we simply banish the distinction between specifications, subspecifications (super-programs?), and programs. To us, they are all *programs; what we give up is that all programs are directly executable. What we gain instead is a more uniform approach in which programs play a role at every level.* — Morgan [39]

Modern agile methods and traditional formal methods are often perceived as being diametrically opposed. We argue that with a touch of new technology, however, they can be quite compatible, and used in combination to good effect.

Following Carroll Morgan [39], we say that a *mixed program* is one written using a mixture of regular imperative statements and declarative *specification statements* [38]. We understand the imperative statements to be written in a conventional object-oriented language (*e.g.*, Java), and the specification statements to be written in a first-order logic specification language (*e.g.*, JML [32] or JFSL [12, 49]).

A *mixed interpreter* is an interpreter that can execute such programs. Hoare [26] and Hayes and Jones [23] argued years ago that executing declarative specifications was infeasible. But advances in the past twenty years suggest that mixed interpreters may in fact be possible. Preliminary steps have already been taken by Wahls *et alia* [31, 47], and we have also developed a prototype mixed interpreter ourselves (to be reported in a future paper).

In this paper we assume the existence of a mixed interpreter and explore how this technology might change our views of software development methodology. We argue that the divide between formal methods and agile methods is due more to limitations of current technology than to fundamentally irreconcilable intellectual positions. Through a series of examples we try to illustrate how a mixed interpreter might create a smooth continuum between the formal and the agile.

1.1 Background

This paper attempts to connect two large and often culturally disjoint schools of the software development methodology literature: formal methods and agile approaches.

Formal methods tend to emphasize specifications and verification, whereas agile methods emphasize tests and rapid prototyping. A characteristic example of traditional formal methods is Dijkstra's idea that programs and their proofs should be constructed in tandem [13] by a process of stepwise refinement [14, 48]. This approach was developed into the *refinement calculus* by Back [2], Hehner [24], Morris [40], Morgan [38, 39], and others.

On the agile side, Kent Beck's test-driven development [3] begins with concrete inputs and gradually evolves a program that computes over a range of similar inputs. Figure 2 lists some of the contrasting terms that are commonly associated with each side.

There have been a few hints of potential common ground between formal methods and agile approaches in the literature. The idea of *lightweight formal methods* [27] germinated around the same time as agile methods and similarly advocated a pragmatic focus on partiality and tool support, but with an emphasis on applying tools to designs and specifications as well as code. More recently, Amey and Chapman

Figure 1 Integer square root case study from Morgan [39]

(a) Purely declarative program:

```
$r : [r^2 \le s < (r+1)^2]$
```

(b) Mixed program:

```
1:  $r \leftarrow 0$
2:  $q \leftarrow s + 1$
3:  $I \triangleq r^2 \le s < q^2$      // loop invariant defn
4:  while $r + 1 \ne q$ do
5:      $p : [r + 1 < q,\ I,\ r < p < q]$
6:      if $s < p^2$ then
7:          $q : [s < p^2 \wedge p < q,\ I,\ q < q_0]$
8:      else
9:          $r : [s \ge p^2 \wedge r < p,\ I,\ r_0 < r]$
10:     end if
11: end while
```

(c) Purely imperative program:

```
1:  $r \leftarrow 0$
2:  $q \leftarrow s + 1$
3:
4:  while $r + 1 \ne q$ do
5:      $p \leftarrow (r + q)/2$
6:      if $s < p^2$ then
7:          $q \leftarrow p$
8:      else
9:          $r \leftarrow p$
10:     end if
11: end while
```

The purely declarative program in *(a)* is refined to the purely imperative program in *(c)*. The refinement process has many steps. One mixed program produced during this process is shown in *(b)*. The parts of the purely imperative program in *(c)* that have not changed from the mixed program in *(b)* are faded to emphasize the lines (5, 7, and 9) that have changed.

[1] argued that programming with a sufficiently powerful static verifier is similar to pair programming. A panel discussion at a National Academies workshop including Kent Beck, Matthias Felleisen, and Anthony Hall came to the surprising conclusion that formal methods and agile approaches actually had much in common, primarily in being driven by perceived risk [43].

Figure 2 Characterizations of formal and agile methods

Formal	Agile
verification	validation
correctness	pleasantness
refinement	refactoring
abstract	concrete
general	particular
proofs	tests
upfront design	design evolves with code
analysis-paralysis	cowboy-coding
programmer	team
Dijkstra [13, 14]	Beck [3, 4]

2. Procedures

In this section we follow the first case study in Morgan's book [39]: a procedure that computes the non-negative integer square root of its input. This example has also been used by Dijkstra, Amey and Chapman [1], and others.

2.1 Programming from Specifications

If you don't drive development with tests, what do you drive it with? Speculation? Specifications?
— Beck [3]

Following Morgan's development, we start with a specification and systematically refine it to a mixed program, and eventually to a purely imperative program. A mixed interpreter supports this formal methodology by enabling the programmer to execute the program at any step in the process. The initial specification is:

$$\textbf{var}\ r, s : \mathbb{N} \tag{1}$$

$$r := \lfloor \sqrt{s} \rfloor \tag{2}$$

where $\sqrt{}$ takes the non-negative square root of its argument. The first step of the refinement is to remove the 'exotic' mathematical operators and replace them with non-deterministic assignment. In Morgan's syntax we write [39]:

$$r : [r^2 \le s < (r+1)^2] \tag{3}$$

which assigns to r an arbitrary value satisfying the formula on the right-hand side of the colon. This specification is now suitable for refinement. After a few refinement steps, Morgan [39] produces the mixed program shown in Figure 1b. Figure 1c shows the purely imperative program that is the final result of the refinement process [39].

Line 3 of Morgan's mixed program defines the loop invariant I. Lines 5, 7, and 9 of Morgan's mixed program are specification statements of the form $v : [pre, inv, post]$, where v is the variable being constrained, pre is the precondition, inv is the invariant, and $post$ is the postcondition [39]. These lines are further refined to simple assignments in the purely imperative program listed in Figure 1c.

2.2 Validation

Beware of bugs in the above code; I have only proved it correct, not tried it. — Knuth [30]

Using tests for validation has been independently advocated in the requirements engineering community (*e.g.*, [21]), the testing community (*e.g.*, [22]), and the agile methods community (*e.g.*, [36, 41, 45]). Validation has also historically been the main argument for the execution (and animation) of specifications (*e.g.*, [20]).

With our prototype mixed interpreter we can execute the specification for integer square root given in Formula 3. Suppose we want to compute the integer square root of 10, which we expect to be 3. With our mixed interpreter we execute Formula 3 with the input 10 and get the surprising result 27. Trying the same execution again we get other surprising results: -12, 42, -55, *etc.*. We eventually get an execution that returns the expected result of 3. What's going on?

A closer look at Formula 1 reveals that Morgan defines r and s over the natural numbers ($\mathbb{N}$), which range from 0 to Ω. Our program is written with ints. In Java, ints are signed 32-bit values ranging from -2^{31} to $2^{31} - 1$. For our mixed interpreter ints are signed 8-bit values, ranging from -128 to 127. These two different sets of ints differ from the natural numbers in the same ways: they have a finite upper bound, and their lower bound is below zero. It turns out that these properties are important for this program.

In the world of signed 8-bit ints, $27^2 = -39$ and $(27 + 1)^2 = 16$. -39 is indeed less than 10 and 16 is greater than 10, so our specification is satisfied: 27 is an integer square root of 10 according to our definition.

In fact, it turns out that there are sixty numbers between -128 and 127 that satisfy Formula 3. Similarly, with 32-bit ints there are 1,073,741,820 solutions that meet our specification as an integer square root of ten, which again is about 25% of the possible values.

We need to re-write Formula 3 to get the result we want given the machine we have. First, we need to explicitly state that we're looking for a non-negative root (which is explicitly stated in Morgan's text, and implicitly in his definition of $r \in \mathbb{N}$). Then we re-write the previous two clauses in terms of division instead of multiplication to avoid overflow. So Formula 3 becomes Formula 4 (for simplicity of exposition we exclude the case where s and r are 0):

$$r : [s > 0,\ r > 0 \wedge r \leq \frac{s}{r} \wedge \frac{s}{r+1} < r + 1] \tag{4}$$

When we execute this revised specification with our prototype mixed interpreter we get the expected result of 3 (as the non-negative integer root of 10) on the first try. Further investigation confirms that 3 is now the only valid answer.

A student [8] of Michael Jackson [28] might diagnose the surprises we experienced in attempting to copy the pseudocode out of *Programming from Specifications* [39] and into a real computer as a failure to distinguish between *requirements* and *specifications*. Requirements are all about – and only about – the problem domain. Programs are all about – and only about – the machine. Specifications stand in between requirements and programs, speaking of shared phenomena.

Formulas 1, 2, and 3 are really requirements: they speak about the domain of mathematics. Formula 4 is a specification: it describes the problem domain solution in terms of the machine. We were seduced into thinking of Formulas 1, 2, and 3 as specifications because we like to think of programming as a mathematical activity. This illusion disappeared as soon as we tried to execute these 'specifications'.

Underestimating the significance of arithmetic overflow is not a mistake confined to novices. Joshua Bloch [7] reports making exactly this mistake when he copied (a mathematically proven) binary search algorithm out of Jon Bentley's *Programming Pearls* [5]. When computing the mid-point of high and low, Bloch's implementation (which was shipped with the standard Java libraries for almost a decade) would potentially overflow if the size of the array was greater than about a billion elements.

Similarly, Rod Chapman [personal communication] relays that Praxis High Integrity Systems uses this integer square root example in their SPARK/Ada training courses. The trainees, who are usually experienced programmers, are given the specification and asked to implement it – and to verify their implementation with the SPARK Examiner tool. The SPARK Examiner statically verifies that the code meets its specification and will not throw any runtime exceptions (integer overflow causes a runtime exception in Ada). Only one 'student' has ever written a verifiably correct implementation on the first try: Professor Robert Dewar.

More quantitatively, Christey and Martin [10] report that integer overflows are an increasing source of discovered security vulnerabilities, especially in operating systems code. These kinds of errors are made by competent professional programmers every day.

Testing serves as an important tool for validating specifications (declarative programs) and for verifying imperative programs. Just as fully imperative programs often compute correct results for common inputs and incorrect results for uncommon inputs, fully declarative programs often allow undesirable results for common inputs (*i.e.*, are under constrained). Testing our specification by executing it with a mixed interpreter was essential to validating that the specification accurately captured our requirements.

2.3 Test-Driven Development

What of testing and debugging? They are still necessary. ... Those were the only errors, and 'it ran third time.' But the point had been made: mathematical rigour cannot eliminate mistakes entirely.

— Morgan [39]

Test-Driven Development is a well-known agile methodology advocated by Kent Beck [3]. The motto of test-driven development is *red/green/refactor*. First the programmer writes a test for functionality that does not yet exist, and confirms that the test fails. This failing test is indicated by a red light in the test harness. Next the programmer writes

Figure 3 Test-Driven Development applied to the integer square root specification. Each row is a new step in the development. Test cases are in columns. At each step either a new test is added or the program is modified. Test results can be red (R), green (G), or yellow (Y). Red is fail. Green is pass. Yellow means the result is a super-set of the desired result.

Step	**Program**	$\lfloor\sqrt{9}\rfloor = 3$	$\lfloor\sqrt{16}\rfloor = 4$	$\lfloor\sqrt{10}\rfloor = 3$	$\lfloor\sqrt{100}\rfloor = 10$	$\lfloor\sqrt{120}\rfloor = 10$
1	$r = 0$					
2	"	R {0}				
3	$r = \mathbf{3}$	G				
4	"	G	R {3}			
5	$r^2 = s$	Y {-125, -3, **3**, 125}	Y {..., -4, **4**, 28, ...}			
6	$r^2 = s \wedge \mathbf{r \geq 0}$	Y {**3**, 125}	Y {**4**, 28, ...}			
7	$\mathbf{r} = \frac{\mathbf{s}}{\mathbf{r}} \wedge r > 0$	G	G			
8	"	G	G	G		
9	"	G	G	G	G	
10	"	G	G	G	G	R ∅
11	$r \leq \frac{s}{r} \wedge r > 0$	Y {1, 2, **3**}	Y {1, 2, 3, **4**}	Y {1, 2, **3**}	Y {1, 2, ..., **10**}	Y {1, 2, ..., **10**}
12	$r \leq \frac{s}{r} \wedge r \geq \frac{s}{r} \mathbf{-1} \wedge r > 0$	G	G	G	G	R ∅
13	$r \leq \frac{s}{r} \wedge r \geq \frac{s}{\mathbf{r+1}} - 1 \wedge r > 0$	Y {2, **3**}	Y {3, **4**}	Y {2, **3**}	Y {9, **10**}	G
14	$r \leq \frac{s}{r} \wedge r \mathbf{>} \frac{s}{r+1} - 1 \wedge r > 0$	G	G	G	G	G

the simplest possible code to make the test pass – perhaps as simple as `return 3`. This passing test is indicated by a green light in the test harness. The programmer then adds new test cases and refactors the implementation to be a more general program that works over a broader range of inputs.

With a mixed interpreter the test-driven methodology can also be applied to specifications. Making specifications interactive in this way potentially offers usability benefits. One way to develop a specification is to have a brilliant flash of insight and, with a single stroke of the pen, capture that insight in a mathematical formula. Another possibility is to start with concrete input/output pairs, expressed as test cases, and build up the general formula through interaction with a mixed interpreter.

Figure 3 walks through applying test-driven development to the integer square root example. Each step in the development is a new row in the table, and each step either adds a new test case or modifies the program. The results for each test are either red (R), green (G), or yellow (Y). Red and green indicate failing and passing tests, respectively, as per normal. Yellow indicates that the expected answer is among the answers, but is not the only answer (*i.e.*, the program is under-constrained). In red and yellow cases a sample of the results returned by the program is also displayed. Step-by-step:

1. We start with the program $r = 0$ and no test cases.
2. We add the test case $\lfloor\sqrt{9}\rfloor = 3$, which fails. Keep it simple to start with: only perfect squares.
3. Revise the program to $r = 3$. The single test case passes.
4. Add a new test case, $\lfloor\sqrt{16}\rfloor = 4$, which is also a perfect square. This test case fails.
5. Revise the program to $r^2 = s$. Both test cases return yellow, due to negative roots and overflow.
6. Add $r \geq 0$ to the program to eliminate the negative roots.
7. Divide both sides of $r^2 = s$ by r to avoid overflow. Since we're dividing by r let's also now ensure that $r > 0$. Both perfect-square tests pass.
8. Add a new test case, $\lfloor\sqrt{10}\rfloor = 3$, our first test case that's not a perfect square. Surprisingly, it passes: $\frac{10}{3} = 3$ in integer division.
9. Add a new test case with some larger numbers: $\lfloor\sqrt{100}\rfloor = 10$. This also passes; it's a perfect square.
10. Add a new test case with a larger number that isn't a perfect square: $\lfloor\sqrt{120}\rfloor = 10$. This is a boundary case just under $\lfloor\sqrt{121}\rfloor = 11$. It fails: there is no integer r such that $r = \frac{s}{r}$. For example, $\frac{120}{10} = 12$, and $\frac{120}{11} = 10$.
11. Relax the program from an equality to an inequality. Now all tests are yellow. Notice that the desired result is always the upper bound of the result set. We need a stronger lower bound for r to eliminate these undesired values.
12. Add $r \geq \frac{s}{r} - 1$ to the program. It's one less than our upper-bound, so maybe it will work. All tests are green except the last one, which again has no result. Why doesn't our desired answer 10 work? $\frac{120}{10} - 1 = 11$, which is greater than 10. How can we change the left hand side so that it is a lower bound on 10? Let's examine the concrete values. Possibilities include: $\frac{120}{10} - 2 = 10$ or $\frac{120}{10+1} - 1 = 9$. The latter seems like a better guess: sticking 1's into a formula is more likely to work out than 2's.
13. Change the lower bound to $r \geq \frac{s}{r+1} - 1$. Most tests are yellow: this lower bound is a bit too low.
14. Change '≥' to '>' on the lower bound clause to tighten it up a bit. All tests now pass.

The astute reader will see that the specification developed here is the same as that in Formula 4 (notwithstanding the $s > 0$ conjunct). Although both methods arrived at the same result, they did so in different manners. With Beck's agile method we dealt with negative values and overflow early in the development, whereas with Morgan's formal method these adaptations to the actual machine were the last step. In the agile approach we adapted to the environment first, and generalized later. In the formal approach, we generalized first and adapted to the environment later.

3. Data Types

Specifications of data types include not only the behavior of the type's operations, but also *representation invariants* [25] and *abstraction functions* [25]. (Jones [29] provides an interesting discussion of the early work on abstraction functions.) Representation (or class) invariants define the set of valid concrete (runtime) values for the structure. An abstraction function maps concrete values to their abstract counterparts.

A mixed interpreter might make agile practical use of both representation invariants and abstraction functions.

3.1 Test-Input Generation

A number of recent tools, such as TestEra [35] and Korat [9], have used representation invariants as a basis to generate test inputs. The goal is to generate all non-isomorphic test inputs that satisfy the representation invariant specification.

A mixed interpreter naturally subsumes the functionality of these task-specific tools. The advantages to this more general approach are (1) the programmer uses the same specification language for class invariants as for procedures; (2) the ease with which various specifications can be composed in order to generate test-inputs with a particular focus (composition of logical specifications is simply conjunction); (3) the other tasks for which these same specifications can also be used, as described in the rest of this paper.

Automatic generation of test inputs from specifications has already been established as an area that blurs the boundaries between the concrete, test-centred world of agile methods and the abstract world of formal methods. Mixed interpreters make this part of a more general and uniform approach to programming.

3.2 Data Structure Repair

Sometimes data structures get into a bad state. At this point options include: abrupt termination, try to keep computing with invariants violated, or try to repair the data structures before continuing. This last option, data structure repair, has received some attention in recent years (*e.g.*, [11, 18]), and is usually based on the representation invariants.

Given a logical specification of the class invariant, a mixed interpreter can easily be used to (a) check that the current state complies with the invariant, and (b) to search for a state that does comply with the invariant.

Performing data structure repair with a mixed interpreter might have some advantages and disadvantages as compared with previous specialized approaches to repair. The main disadvantage is that, without some extra customization for this task, the mixed interpreter isn't going to be constrained to find a valid state that is similar to the broken state. In the extreme, consider that an empty list is a valid list: one valid (although not preferable) repair strategy for a list data structure is just to delete all elements of a broken list.

An advantage that a mixed interpreter might have over a customized solution is the ease with which various specifications can be combined. For example, class invariants are usually checked at the end of public methods. If, at the end of the execution of a public method, the mixed interpreter finds that the class invariant of the receiver no longer holds, it can search for a state that respects both the invariant and the postcondition while mutating only the receiver. Such flexibility might be harder to achieve with a customized solution.

3.3 Object Contract Compliance

In many object-oriented languages, including Java and C#, every object is supposed to provide implementations of the so-called 'object contract' methods `equals` and `hashCode`. Correct implementation of these methods is notoriously tricky, tedious, and error-prone [6, 33, 42, 44, 46].

However, correct implementations of these object-contract methods can be mechanically derived from programmer-provided abstraction functions [44]. A mixed interpreter enables these abstraction functions to be written in the same logic as the other specifications, and therefore also used as part of the other software engineering activities described in this paper. (Previous work by Rayside et al. [44] required non-trivial abstraction functions for this purpose to be written in imperative code.)

While there are some procedures for which it may be easier to write code than a specification, for abstraction functions it is almost always easier to write the specification.

4. Putting it all together: Mock Objects

Mock objects are a specific form of rapid prototyping advocated by the agile methods community [3, 19, 34, 37]. Mock objects, as objects, comprise both procedures and data.

The motivation for mock objects is to facilitate testing of other code that uses complex infrastructure: mock implementations of the infrastructure are developed to enable testing the other code. For example, one might use an alternative in-memory database for testing an order processing system rather than using the real on-disk database.

Suppose we are developing an email client, and are about to write `EmailMessage.bind(AddressBook)`, which attempts to look up a person's email address in the address book. Following an agile test-driven approach [3], we first write two unit tests for `bind`, shown in Listing 1.

Now we need an AddressBook object so that we can run our tests. But the developer working on the real AddressBook implementation hasn't finished it yet. However, following traditional formal methods, she has written a specification for it first (Listing 3). All we need to do then to get a working AddressBook object for our tests is to create a class MockAddressBook that implements AddressBook and specifies the initial conditions (*i.e.*, that the book is empty) on the constructor (Listing 4). Finally we implement EmailMessage.bind() (Listing 2) and run our tests.

With the help of a mixed interpreter we have a mock implementation at no additional cost to writing its specification. Moreover, the specification-based mock provides additional benefits: support for verification of the real implementation, test-input generation, data structure repair, object contract compliance, and so on.

Listing 1. Unit tests for EmailMessage.bind()

```
@Before public void setUp() {
    addressBook = new MockAddressBook();
    addressBook.setEmailAddress("Daniel", "dnj@mit.edu");
}
@Test public void testBindIfPresent() {
    EmailMessage m = new EmailMessage("Daniel");
    Assert.assertTrue(m.bind(addressBook));
    Assert.assertEquals("dnj@mit.edu", m.email);
}
@Test public void testBindIfAbsent() {
    EmailMessage m = new EmailMessage("Robert");
    Assert.assertFalse(m.bind(addressBook));
}
```

Listing 2. EmailMessage.bind() method

```
public boolean bind(AddressBook abook) {
  if (!abook.contains(this.name)) {
    return false;
  } else {
    this.email = abook.getEmailAddress(this.name);
    return true;
  }
}
```

Listing 3. AddressBook interface and specification

```
@SpecField("data : String -> String")
@Invariant("all x:String | lone this.data[x]")
public interface AddressBook {

    @Requires("name != null && email != null")
    @Ensures("this.data = @old(this.data) ++ name -> email")
    @Modifies("this.data")
    void setEmailAddress(String name, String email);

    @Ensures("return - null = this.data[name]")
    String getEmailAddress(String name);

    @Returns("some this.data[name]")
    boolean contains(String name);
}
```

Listing 4. MockAddressBook implementation

```
public class MockAddressBook implements AddressBook {
    @Ensures("no this.data")
    @Modifies("this.data")
    public MockAddressBook() { }
}
```

Terminology. The term *mock object* was introduced by Mackinnon et al. [34]. Since then the terminology in the agile-methods community has evolved [19, 37]. In the new terminology our MockAddressBook would be referred to as a *fake object* rather than a mock object. Fake objects are fully (or mostly) functional replacements, whereas mock objects are simply pre-programmed to respond to a specific sequence of method calls for a single unit test. In other words, fake objects are richer than mock objects and take more effort to implement [37]. A mixed interpreter provides fake objects for no additional effort over specifying them.

5. Performance of a Prototype

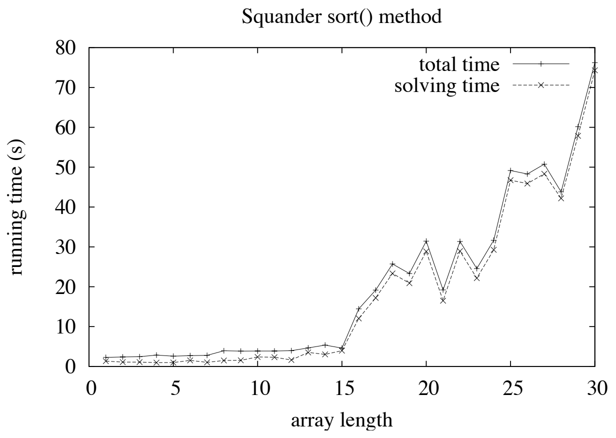
On reasonable assumptions, the whole universe will reach a uniform temperature around four degrees Kelvin long before any interesting calculation is complete. — Hoare [26]

Our prototype mixed interpreter can perform the basic computations described in this paper with small inputs: 8-bit integers and a dozen or so objects. Mixed programs that we have executed with our prototype include the integer square root programs above, the address book mock object program above, sorting a list of integers, and various manipulations of binary search trees.

Figure 4 shows a plot of execution time versus input size for sorting an array of integers by executing a declarative specification for a sort procedure. Lists of up to size 15 can be sorted in a couple of seconds on stock hardware.

These results show that interesting calculation can be completed before the heat-death of the universe. While mixed interpreters may not yet be ready for everyday use, they are clearly no longer beyond the bounds of our imagination.

Figure 4 Performance sorting an array of integers



6. Conclusion

Much work has been done since the crucial first step of considering both specifications and code to be programs. The effect has been to simplify, and make more regular, much of the detail of constructing programs: and there are significant implications for the practice of software engineering generally. — Morgan [39]

Until now, formal specifications have occupied a rather specialized niche in software engineering. While advocates of formal methods have argued, for several decades, that specifications should lie at the center of the programming process, practitioners have been reluctant to adopt them – often out of ignorance, but also from a realistic assessment of their costs and benefits. With programming languages offering better support for runtime assertions, and a more positive attitude to unit testing encouraged by the agile programming movement, the value of specifications is being reassessed. Synergies between tools allow multiple benefits to be obtained from a single specification; the same annotation used as an oracle for unit testing can be fed to a theorem prover.

This paper has argued for taking one step further along the road towards a full integration of specifications and code. In addition to using specifications for traditional purposes, we have proposed that they be executed just like code. From one perspective, this is nothing more than providing some mechanical support for the refinement calculus. From another perspective, however, this makes specifications more agile by shifting the emphasis from abstraction and proof to simulation and checking. Something has been gained and nothing has been lost: systematic, proof-oriented approaches are still perfectly compatible with this technology.

Specification statements [2, 38, 40] were developed as part of the refinement calculus to support a very systematic style of programming that proceeds in an orderly fashion from specification to code. One might imagine that, in a less systematic and more experimental setting, these notions would be less useful. But, on the contrary, it seems likely that the ability to write a mixed program will be particularly helpful when the programmer is less disciplined, since it makes it easier to maintain a complete version of the code (albeit with sections 'implemented' as specifications) and explore it as it evolves.

It has become clear that the writing of code in its narrowest sense is only a small part of software engineering, and even of programming. The best programmers are willing to invest in the surrounding infrastructure – test cases, stubs and oracles, runtime assertions, documentation – and are eager to integrate this infrastructure more closely with the code proper, and to find ways to develop the two in tandem. The ability to execute specifications might take us much closer towards this goal; at the very least, the range of benefits of specification would be greatly expanded.

Through all of this what we have endeavoured to show is that formal methods and agile methods are separated more by shortcomings of existing technology than by fundamental intellectual differences. One focuses on the abstract and on correctness and verification. The other focuses on the concrete and the convenient and on validation. Both emphasize co-development of the program and its evaluation. New technology to connect the concrete and the abstract can give the practicing programmer a more flexible and unified spectrum of approaches.

Acknowledgments

Jonathan Edwards [15–17] has been an important influence in thinking about connecting the concrete and the abstract.

This research was funded in part by the National Science Foundation under grant 0541183 (Deep and Scalable Analysis of Software).

References

[1] Peter Amey and Roderick Chapman. Static verification and extreme programming. In *SIGAda'03*, December 2003. URL `http://www.praxis-his.com/sparkada/publications_confs.asp`.

[2] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Report A–1978–4.

[3] Kent Beck. *Test-Driven Development*. Addison-Wesley, 2003.

[4] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[5] Jon Louis Bentley. *Programming Pearls*. ACM Press, 1986.

[6] Joshua Bloch. *Effective Java*. Addison-Wesley, 2001.

[7] Joshua Bloch. Nearly all binary searches and mergesorts are broken. Official Google Research Blog, June 2006. URL `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

[8] Joshua Bloch. Response to discussion of [7], June 2006. URL `http://lambda-the-ultimate.org/node/1549`.

[9] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. In Phyllis Frankl, editor, *Proc.ISSTA*, Rome, Italy, July 2002.

[10] Steve Christey and Robert A. Martin. Vulnerability type distributions in cve, May 2007. URL `http://cwe.mitre.org/documents/vuln-trends/index.html`. Version 1.1.

[11] Brian Demsky and Martin C. Rinard. Goal-directed reasoning for specification-based data structure repair. *TSE*, 32(12):931–951, December 2006.

[12] Greg Dennis. *A Relational Framework for Bounded Program Verification*. PhD thesis, MIT, 2009. Advised by Daniel Jackson.

[13] Edsgar W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3): 174–186, September 1968.

[14] Edsgar W. Dijkstra. Notes on structured programming. In O.-J. Dahl, C.A.R. Hoare, and E.W. Dijkstra, editors, *Structured Programming*. Academic Press, New York, 1972.

[15] Jonathan Edwards. Example centric programming. In Doug Schmidt, editor, *Proc.19th OOPSLA*, October 2004.

[16] Jonathan Edwards. Subtext: Uncovering the simplicity of programming. In Richard P. Gabriel, editor, *Proc.20th OOPSLA*, October 2005. ISBN 1-59593-031-0.

[17] Jonathan Edwards. No ifs, ands, or buts: Uncovering the simplicity of conditionals. In *Proc.22nd OOPSLA*, pages 639–658, Montréal, Canada, October 2007.

[18] B. Elkarablieh, I. Garcia, Y. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In Alexander Egyed and Bernd Fischer, editors, *Proc.22nd ASE*, Atlanta, GA, November 2007.

[19] Martin Fowler. Mocks aren't stubs, January 2007. URL `http://martinfowler.com/articles/mocksArentStubs.html`.

[20] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, September 1992.

[21] Donald C. Gause and Gerald M. Weinberg. *Exploring Requirements*. Dorset House, 1989.

[22] Dorothy Graham. Requirements and testing: Seven missing-link myths. *IEEE Software*, 19(5):15–17, 2002.

[23] Ian Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989. ISSN 0268-6961.

[24] E. Hehner. Do considered od: a contribution to the programming calculus. *Acta Informatica*, 11:287–304, 1979.

[25] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, December 1972.

[26] C. A. R. Hoare. An overview of some formal methods for program design. *IEEE Computer*, 20(9):85–91, 1987.

[27] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, pages 21–22, April 1996.

[28] Michael Jackson. *Software Specifications and Requirements: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995. ISBN 0-201-87712-0.

[29] Clifford B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[30] Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Letter to Peter van Emde Boas, March 1977. URL `http://www-cs-faculty.stanford.edu/~knuth/faq.html`.

[31] Ben Krause and Tim Wahls. jmle: A tool for executing jml specifications via constraint programming. In L. Brim, editor, *Formal Methods for Industrial Critical Systems (FMICS'06)*, volume 4346 of *LNCS*, pages 293–296. Springer-Verlag, August 2006.

[32] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, April 2003. URL `http://www.jmlspecs.org`.

[33] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[34] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, 2000.

[35] Darko Marinov and Sarfraz Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. In *Proc.16th ASE*, pages 22–31, November 2001.

[36] Robert C. Martin and Grigori Melnik. Tests and Requirements, Requirements and Tests: A Möbius Strip. *IEEE Software*, 25(1):54–59, 2008.

[37] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[38] Carroll Morgan. The specification statement. *TOPLAS*, 10(3), 1988.

[39] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, Inc., 2nd edition, 1998. First edition 1990.

[40] J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9 (3), December 1987.

[41] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice-Hall, Inc., 2005.

[42] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, November 2008.

[43] Committee on Certifiably Dependable Software Systems, editor. *Summary of a Workshop on Software Certification and Dependability*. The National Academies Press, 2004. ISBN 978-0-309-09429-0. URL `http://books.nap.edu/catalog.php?record_id=11133`.

[44] Derek Rayside, Zev Benjamin, Rishabh Singh, Joseph P. Near, Aleksandar Milicevic, and Daniel Jackson. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In Joanne Atlee and Paola Inverardi, editors, *Proc.31st ICSE*, 2009.

[45] Filippo Ricca, Marco Torchiano, Massimiliano Di Penta, Mariano Ceccato, and Paolo Tonella. Using acceptance tests as a support for clarifying requirements: A series of experiments. *Information and Software Technology*, 51(2):270–283, 2009.

[46] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In Erik Ernst, editor, *Proc.21st ECOOP*, volume 4609 of *LNCS*, pages 54–78, Berlin, Germany, July 2007. Springer-Verlag.

[47] Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing formal specifications with concurrent constraint programming. *Automated Software Engineering Journal*, 7:315–343, 2000.

[48] Niklaus Wirth. Program development by stepwise refinement. *CACM*, 14(4):221–227, April 1971.

[49] Kuat Yessenov. A light-weight specification language for bounded program verification. Master's thesis, MIT, May 2009. Advised by Daniel Jackson.