Formal Methods during the Programming Phase

Richard F. Paige¹, Jim Woodcock¹, Phillip J. Brooke² and Ana Cavalcanti¹

¹ Department of Computer Science, University of York, UK.
{ paige, jim, alcc }@cs.york.ac.uk
² School of Computing, University of Teesside, UK. pjb@scm.tees.ac.uk

Abstract. Formal methods involves the application of sound mathematical specification and reasoning techniques to the development of software systems. They can be applied across all phases of the engineering lifecycle. This chapter discusses their specific application to the programming phase, which can entail enrichment of executable code with mathematical artefacts (to support and improve our capabilities for reasoning), and the derivation of executable code from mathematical specifications. We provide an overview of some of the important approaches that can be applied to this phase.

1 Introduction

1.1 The Engineering Lifecycle

Software engineering is the disciplined process of building and maintaining large-scale software systems. The software engineering technical process involves understanding customer requirements; analysing those requirements to identify inconsistencies, ambiguities, and priorities; designing a solution that satisfies the requirements (and clarifies why certain requirements may not have been met); implementing a solution in a form that can be accepted, processed, and executed by a computer; validating and verifying that the solution is acceptable, correct, and coherent; deploying the implemented solution; and maintaining the solution over its lifetime. The software engineering management process involves monitoring and measuring progress, providing guidance to engineers and customers alike, and ensuring that the resultant product is delivered in a timely manner.

Thus, a typical engineering lifecycle involves several phases, including requirements engineering, design, implementation (or programming), verification and validation, deployment, and maintenance. Different engineering processes provide different principles and practices for supporting each of these phases.

1.2 Formal methods

Formal methods are mathematically based techniques that are applied in an engineering lifecycle. More precisely, a formal method consists of one or more formal specification languages and a corresponding process that describes –quite possibly in a lightweight and informal way– how the specification languages are to be applied, and where, in the engineering lifecycle.
A formal specification language has a mathematical description of the language’s syntax, for example using a context-free grammar or EBNF, and a mathematical definition of the language’s semantics. Popular approaches for defining a language’s semantics include denotational semantics, axiomatic semantics and operational semantics. Well-known formal specification languages include Z [52], B [1], CSP [23], VDM [27], Alloy [26], SPARK Ada [5], and PROMELA [25], amongst many others.

Formal methods vary widely in terms of the level of rigour, structure, and formality in their process. Some methods simply provide a recommended style of specification and some basic practices that suggest how specifications should be constructed. A good example of this is with the various house styles for the Z specification language. Other methods provide a disciplined, rigorous process that defines well-formed specifications (e.g., those that are satisfiable), as well as steps or strategies that can be used to transform specifications created at one phase of the engineering lifecycle into specifications that are suitable for use in successor phases.

1.3 Applications of formal methods in programming

Formal methods can be applied in any phase of the engineering lifecycle, including for capturing and reasoning about requirements, designs, implementations, and deployed systems.

There is widespread interest in their application at the implementation – or programming – phase, though some researchers and practitioners take the view that there may be more benefits to applying formal methods earlier in the development phase, e.g., to detect requirements omissions and errors, or to help ascertain the implementability of a design. The wide interest in applying formal methods at the programming phase may be because of the substantial emphasis on this phase in many industrial software development projects: very often, new projects build on existing software, and the ability of our engineering methods and techniques to accommodate and interface with existing programs is therefore critical.

What does it mean to apply formal methods at the programming stage of the engineering lifecycle? There are many possible interpretations of this. Some of the most widespread interpretations are as follows.

1. To use formal methods to calculate or develop programs from designs or requirements;
2. To use formal methods to improve the quality of existing code, through annotation of code with mathematical constructs (e.g., invariants), or derivation of specifications from programs;
3. To use formal methods to help derive downstream engineering artefacts from programs, in order to simplify or improve the successor engineering phases like deployment, validation, and verification.

1.4 Approaches: code-based and specification-based

In this section, we will give examples of techniques that support these different kinds of activities. There are two basic approaches to applying formal methods in the pro-
gramming phase. These can be broadly described as approaches that are code-based and approaches that are specification-based.

Specification-based approaches – offered by formal methods like B, Perfect Developer [15], CSP, and Circus (see Section 2) – aim at supporting the style of development described in point 1, earlier. Specification-based approaches start from abstract, mathematical specifications and aim to develop code from these specifications. A variety of approaches can be used for this, including: positing an implementation, then proving that the implementation satisfies the initial specification; and generating (sometimes called calculating) an implementation from a specification. Such approaches are often based on a rigorous mathematical definition of a refinement relation between specifications and programs. That is, there is a mathematical underpinning that allows engineers to demonstrate precisely that the calculated or posited programs satisfy the original specifications of designs or requirements. The precise definition of satisfaction, and hence refinement, depends on the underlying formal semantics of languages used to express requirements, designs, and programs. An important side-effect of applying refinement in a calculational manner is that a mathematical proof of satisfaction is produced. We shall explore specific examples of specification-based approaches in Section 2.

Program-based approaches – offered through formal methods like Spec# [34], Daikon [17], Verisoft, Eiffel [36] and JML [10] (see Section 3) – support the style of development described in point 2, earlier. Program-based approaches assume that a program, in some form, already exists and can be enriched with additional information to help demonstrate that the program satisfies desirable properties.

Conceptually, specification-based and program-based approaches applied in the programming stage are very similar in terms of their mathematical bases and the results that are produced: (i) a program that satisfies a specification; and (ii) a mathematical argument that the relationship between program and specification is sound. Where they differ is in terms of how they are implemented and applied, i.e., how they are specifically applied in an engineering lifecycle, and the tools that exist for their support. We discuss this in more detail in the sequel.

2 Specification-based approaches

2.1 Refinement calculi

Programs do not need to behave exactly as indicated in a specification. It is often the case that a specification is abstract, in the sense that it captures the requirements of a system, and leaves open various design decisions; these are firmed up in a program. What we require in this case is that the program is a refinement of the specification and behaves better from the point of view of the user. The program may apply in more situations, and may satisfy a richer set of properties.

In a refinement calculus, this scenario is formalised using a rich language in which abstract specifications, designs and programs can be written, and are all regarded as programs [3, 38, 39]. In the context of such a language, refinement is defined as an ordering relation on programs: $p_1 \subseteq p_2$ holds when the program $p_2$ is at least as good as the program $p_1$ in the sense that it will meet every purpose and satisfy every specification satisfied by $p_1$. 
A refinement relation is a partial order: it is reflexive, anti-symmetric, and transitive. Transitivity justifies stepwise refinement, and antisymmetry reduces proofs of equivalence to proofs of mutual refinement. Furthermore, to allow independent refinement of subcomponents of programs, the language operators should preferably be monotonic with respect to $\sqsubseteq$. For example, if $p_1 \sqsubseteq p_2$, we are able to conclude that $p_1; p_3 \sqsubseteq p_2; p_3$. For constructs of concurrent and object-oriented languages, such properties may require extra restrictions due to issues of visibility of and conflicting access to information.

The refinement relation can be defined in terms of a semantic model of the language; widely-used models are denotational in style, and based on weakest preconditions [16] or relations on states [4]. In an algebraic style of reasoning, the properties of the specification and programming constructs are captured by equations and inequations (laws) that directly relate these constructs. Reasoning, therefore, is entirely conducted at the programming level. This has been an appealing basis for specialised techniques and tools for refinement.

A comprehensive set of laws for imperative programming can be found in [24]. Several other paradigms are now equipped with such sets of laws [6, 43]; their application to support program transformation vary from refactoring to compilation, for example [14, 44].

In the context of a refinement calculus for program derivation from specifications, the algebraic approach is pursued in [37]. A distinguishing feature of Morgan’s calculus is the specification statement $w : [pre, post]$, which describes a program that, when executed in a state satisfying the precondition $pre$, terminates in a state satisfying the postcondition $post$, possibly modifying the values of variables in the list (frame) $w$. Morgan’s calculus provides laws that allow specification statements to be refined into executable code.

The main challenge is the calculation of loops. For that, two important pieces of information need to be provided: the loop invariant and variant. A loop invariant is a property that characterises the functionality of the loop: if it holds at the beginning of loop, it is preserved by each iteration, and holds at the end (in conjunction with the termination condition). As an example, we consider the program below, which calculates the sum of the elements in an array $a$ of $N$ positions indexed from 0 to $N - 1$.

```
1 sum, i := 0, i;
while (i < N) do
   sum := sum + a[i];
   i := i + 1
od
```

The invariant of this loop is $\text{sum} = \sum_{n=0}^{i} a[i]$. It is established by the assignment before the loop, and when conjoined with the termination condition $i = N$, it establishes that the variables $\text{sum}$ contains the result of the sum of all elements of $a$ at the end of the loop.

A loop variant establishes an upper bound on the number of iterations. It is an integer expression, whose value is decreased after each iteration, but never below 0. For our example, the variant is $N - i$. In Morgan’s calculus, using an invariant and a variant, we can calculate a loop from a specification statement. Similar approaches are seen in other languages, e.g., Eiffel.
As well as (algorithmic) refinement, program development typically also requires change of data representation. The data structures used in specifications are typically abstract and, often, not provided in traditional programming languages. During design, such data types need to be given more concrete representations.

A common technique to establish data refinement is called simulation. It is based on the definition of the relation between the abstract and the concrete data types. For example, if the abstract data type is a set \( s \), and the concrete representation is an array \( a \) that contains the elements of the set, the relation between them can be defined as \( s = \text{ran} \ a \), where the function \( \text{ran} \) gives the set of elements of the given array. Such relations are called coupling invariants. In many cases of practical interest, like in our example, these relations are functional, and are called abstraction functions.

In Morgan’s calculus, data refinement is formulated for programming modules. A module includes state variables, a state initialisation, and procedures which act on the module state. Data refinement allows us to change the variables in the module, and modify the procedures accordingly. In general, data refinement can be considered for any programming structure that provides hiding of data representation. Even variable blocks, which declare and define the scope of variables, can be changed to declare a different set of more concrete variables using data refinement.

**Refinement in Z**  The style of Morgan’s calculus has been adopted for development of programs based on more sophisticated specification languages: the Z notation is an example. The Z language [47] was developed in the early 80’s as a state-based specification notation. Operations in Z are specified as relations; the style of definition is similar to that used in specification statements, but does not give a separate account of the frame and of the precondition, which can be calculated.

A graphical notation called schema groups the declarations of the affected state variables, inputs, outputs, and the predicate that describes the operation. In addition, the schema calculus provides a number of structuring facilities that allows the construction of operation specifications by a combination of simpler operations. They can be disjoined, conjoined, included in the definition of another, used in sequence, and so on.

An adaptation of Morgan’s refinement calculus to Z [11] allows specifications written in the richer notation of Z to be refined to programs in a calculational style. In the Z refinement calculus, operation specifications can be refined to specification statements or to more structured designs: sequences, conditionals, procedure calls, and so on. Afterwards, laws very similar to those of Morgan’s calculus, but adapted to the Z mathematical notation, can be applied to calculate programs.

The Z data refinement technique is not calculational, though. ... Jim wanted to say something here?

**ClawZ** Industrial application of the Z refinement technique has been successfully achieved in the verification of control systems using a toolset called ClawZ [2]. In this technique, a system is specified using a control law diagram, which is a notation widely adopted by engineers. In particular, ClawZ is based on the discrete-time diagrams developed using the Simulink tool associated with MATLAB [35].
ClawZ includes a tool called Z producer which provides a Z account of the functional semantics of (one cycle) of the control law diagram. The objective is to use the Z specification to prove that procedures of a SPARK Ada program correctly implement the functionality of groups of diagram blocks.

Other tools in ClawZ, using information extracted from the Ada program and from the diagram, construct refinement conjectures that establish that the procedures refine the specification obtained from the diagram. Proof of the refinement conjectures is conducted using a Z theorem prover called ProofPowerZ [29].

Tailored proof tactics afford automatic proof of up to 95% of the verification conditions that arise from the refinement conjectures. With further use and further tailoring of the tactic, better and better levels of automation are possible.

We observe that, even though the ClawZ technique is based on the Z theory of refinement and on a refinement calculus, it is actually a verification technique. The Z specification and the Ada program are created before the verification, which consists of a posteriori proof of correctness, rather than the calculation of a correct procedure.

**CSP**

Further challenges are present when we consider the development of concurrent programs. A notation designed for specification and refinement of concurrent reactive programs is the process algebra CSP [42]. The programming languages occam [28] and Handel-C [48] are both based on this paradigm, which frees the programmers from worrying about monitors, semaphores, and shared variables. More recently, there have been efforts to extend Java with the programming facilities of occam and CSP [22, 51, ?].

In the CSP paradigm of specification and design, programs are formed by components, which we call processes, and interact with each other and an external environment. When developing a process, we are not only interested in the inputs and outputs, but also in each of the interactions in which the process may engage. Inputs and outputs are forms of interaction in this context.

An interaction is characterised in CSP as an event. In the description of a process, a first element of interest is the set of events in which it can participate; the definition of an event simply gives it a name.

Refinement is based on the possible interactions of the processes; there are three notions of refinement for CSP. The simplest notion, called traces refinement, states that the interactions of the implementation process have to be interactions that could be performed by the specification process. This is useful to reason about safety of implementations.

A further concern is related to the events in which a process may refuse to engage. Failures refinement requires that, in addition to preserving the history of interactions of the specification, the program does not deadlock more than the specification. This notion of refinement can be used to reason about liveness.

Finally, the most strict refinement relation also takes into account the possibility of divergence. In this case, refinement reduces the number of situations that lead to refinement.

Tools for CSP include an animator called ProBE and a model checker called FDR. As opposed to traditional model checking techniques based on labelled transition sys-
tems and temporal logic, FDR is a refinement checker. Both the system under analysis and the property of interested are specified in CSP, and FDR automatically verifies that the process that specifies the property is refined by the CSP model of the system. Any of the refinement relations can be checked.

Proof of refinement using FDR is fully automatic, but it is restricted to finite processes, involving very restricted data types. Abstraction techniques have been employed to allow analysis of a wider class of systems and properties. They allow the analysis of simpler processes to justify conclusions about data-rich processes when their behaviour is data independent [31]. Lazy, or just-in-time, evaluation can enable (incomplete) analysis of some larger processes [9].

Circus A refinement calculus is also available for a language called Circus [13], which combines Z and CSP to support the development of state-rich reactive systems. There has been a trend to combine and integrate notations, techniques and tools to achieve a wider coverage in support for analysis and verification of systems. Circus is one of many combinations of a process algebra with a state-based notation [18, 49]. It distinguishes itself, though, in that it is a language for refinement.

As such, Circus is a language that encompasses the specification constructs of Z to specify state, and CSP to specify reaction, but also includes programming constructs like assignments, conditionals, and loops for writing programs. Most importantly, Circus does not follow the communicating data type paradigm adopted by many integrated languages that are geared towards analysis and validation and system properties. Typically, a system is specified in Circus by a process like in CSP, but with a state, which is specified in Z, along with its associated operations; the behaviour of the process is defined by a main action, which is described using a combination of CSP constructs and Z operations.

In the communicating data types integration philosophy, the events of the description of reactive behaviour are identified with operations of a data type, so that processes control the use of the data types. This is a very elegant paradigm of specification, but does not correspond to the way in which programs are written. In Circus, the data operations and communication constructs are freely mixed. This means that Circus needed an entirely new semantic model, and reuse of Z and CSP tools is not immediate.

For refinement to code, the Circus technique extends that of Z. It is based on an iterative process which starts from a centralised process that specifies the state and the reactive behaviour of the system without the use of parallelism. During refinement, the parallel structure of the program is introduced gradually. In each iteration of the refinement process, a data refinement introduces a concrete representation for the state of the process. Afterwards, the main action is rewritten using algebraic laws of refinement so that it exhibits the parallelism embedded in the required design. Finally, the process is split into parallel processes as indicated by its main action.

Additionally, Circus is being used to extend the ClawZ technique for verification of control systems [12]. Due to its nature as a combined language, it is possible to use Circus to verify not only the correctness of the Ada procedures, but also their combined use in the program, and the scheduling policy.
2.2 Generative approaches

One of the apparent advantages of annotation-based approaches to formal methods is in avoiding a linguistic gap, by allowing programmers to use the languages that they are experienced with, and by avoiding introducing very new and different languages to the programmers. However, reasoning about realistic programming languages—particularly those that provide powerful and complex constructs, like inheritance, value and reference types (and thus aliasing), and polymorphism—is difficult, and pushes automated reasoning tools to the limit. Other approaches manage these difficulties by reducing the scope of problems that can be handled (e.g., by focusing on reasoning about specific properties, such as non-null pointer dereferencing), by restricting the parts of the language that can be subjected to automated reasoning, and by requiring user interaction to discharge verification conditions.

An alternative style of development, which avoids the problems of dealing with real programming languages, is offered by the so-called generative approaches to program development. These approaches focus on automatically generating executable code, in a real programming language, from an abstract formal specification. In doing so, the approaches introduce a new language which exhibits exactly and only the constructs and logic that can be supported and reasoned about by corresponding reasoning tools. Systems are typically developed step-wise, with more abstract descriptions supplemented with more implementation details and proved correct at each step.

Two well-known examples of generative approaches to program design are provided by Perfect Developer and the B Method. We describe each briefly here.

**Perfect Developer**  Perfect Developer [15] is a pure object-oriented method for program design. It is a generative approach, and introduces its own formal specification language (which has similarities with JML, Eiffel, Z, and SPARK Ada). The specification language can be used to automatically generate C++ and Java programs. Automatic verification is supported on the formal specification language. In particular, verification is supported through generation of verification conditions, in order to capture a collection of well-formedness properties, such as: variables are initialised prior to first use; type rules are obeyed; loops terminate; and postconditions are satisfied. A theorem prover is included with Perfect Developer to discharge these verification conditions. Of particular note is that the prover is fully automatic, and that it is based on a multi-sorted logic (as opposed to two-valued boolean logic which is used in many other provers).

A novelty with Perfect Developer, and with the B Method discussed in the next section, is that it supports automated refinement. In the context of Perfect Developer, refinement involves both automatic generation of executable code in a programming language (e.g., in C++) and transformation to an internal language, which is a subset of the standard Perfect Developer language. This transformation eliminates constructs that are difficult to map directly to programming languages like Java and C++ (such as quantifiers), while introducing concrete constructs like assignment statements. The internal language can then be used as the source for automatic generation of executable code.

An example of a Perfect Developer specification of an operation is in Listing 1.1. The *schema* keyword introduces a state-change operation of a class. The postcondi-
tion of the schema indicates both a frame (i.e., that balance can be changed), and the postcondition expression itself.

Listing 1.1. Perfect Developer example

```plaintext
1 schema withdraw(amount: nat)
  pre amount >= 0
  post change balance
    satisfy balance' = balance-amount
```

An important novelty of Perfect Developer is that its specifications (such as withdraw) can be extended with refinements and implementations. Perfect Developer will then attempt to generate verification conditions and automatically prove that the implementation satisfies the specification (though some verification conditions may not be provable automatically). Thus, the specification in Listing 1.1 could be extended as follows.

Listing 1.2. Extended Perfect Developer specification

```plaintext
1 schema withdraw(amount: nat)
  pre amount >= 0
  post change balance
    satisfy balance' = balance-amount
5 via
    balance! = balance-amount
end
```

The specifications in Listings 1.1 and 1.2 are written in Perfect’s own specification language; automatic code generators for C++ and Ada are available in order to produce a working implementation.

Perfect Developer is unique amongst many formal methods in that it provides fully automated verification facilities. However, the requirement to learn and apply a new language may (as in other formal methods) make it less attractive to developers.

The B-method The B Method, due to Abrial, is a formal program design method that focuses on refinement of specifications to code [1,45]. It is based on the theory of generalised substitutions, and introduces the Abstract Machine Notation (AMN) for writing modular specifications. The engineering process involves constructing machines, refining machines, proving that refinements satisfy desirable properties, and automatically generating code (e.g., in C) from refined machines. The B Method is supported by a number of powerful tools that aim to automate the refinement process, e.g., the B-Toolkit, Atelier B, Rodin, ProB, and B4Free. B is one of the more active formal methods, and has been linked with other formal methods and software engineering methodologies. The Rodin project, in particular, looked at defining links between the Unified Modelling Language (UML) and B, and provided open-source modelling and proof support as part of the Eclipse framework.

An example machine is shown in Listing 1.3.

Listing 1.3. Example B machine

```plaintext
1 schema withdraw(amount: nat)
  pre amount >= 0
  post change balance
    satisfy balance' = balance-amount
5 via
    balance! = balance-amount
end
```
MACHINE BankAccount
VARIABLES balance
INVARIANT balance ∈ N
INITIALISATION balance := 0
OPERATIONS
  \( d \leftarrow \text{deposit}(d) \) \( \triangleleft \)
  PRE \( d \in \mathbb{N}_1 \)
  THEN balance := balance + d
  END
  \( w \leftarrow \text{withdraw}(w) \) \( \triangleleft \)
  PRE \( w \in \mathbb{N}_1 \land balance \geq w \)
  THEN balance := balance - w
  END
END

It gives a single variable, the balance, and two operations, deposit and withdraw. Each has preconditions, introduced by PRE and a body which describes what the modifications specified by the operation. Nondeterminism can be explicitly introduced into specifications via operators such as ANY, CHOICE, and SELECT.

Proof obligations are imposed in a number of circumstances. A machine’s initialisation must satisfy the invariant; similarly, an operation must (when its precondition is satisfied) preserve the invariant. Large specifications can be structured (e.g., with INCLUDES. This enables modularity and some degree of abstraction. Machines can be parameterised, then included (with these parameters instantiated) allowing for components to be constructed and re-used.

B supports step-wise refinement, starting from an abstract specification and gradually producing a final implementation. The intermediate steps are refinement machines, written in the same notation. Each refinement provides more detail of how an operation is implemented or data is represented. Each step is associated with further proof obligations (e.g., each transition in a refined machine is matched by a relevant transition in the abstract machine).

There are similarities to other approaches already seen. Morgan’s calculus describes loops with the loop invariant and variant. A loop in a B machine is written in the form

Listing 1.4. Loops in B

<table>
<thead>
<tr>
<th>1</th>
<th>WHILE P</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DO S</td>
</tr>
<tr>
<td>3</td>
<td>INVARIANT I</td>
</tr>
<tr>
<td>4</td>
<td>VARIANT V</td>
</tr>
<tr>
<td>5</td>
<td>END</td>
</tr>
</tbody>
</table>

capturing the same ideas.

Finally, B implementation machines are a specialised form of refinement machine intended for the production of executable code. Implementation machines have a restricted form of AMN, e.g., nondeterministic constructs are disallowed. The code generator can then produce code suitable for a conventional compiler.
3 Program-based approaches

Program-based approaches operate on existing programs, and aim to support analysing and reasoning about them. One of the attractions of this approach to formal methods is that it is applicable to legacy programs (even those for which source code does not exist, as we see shortly). Another potential benefit of this kind of approach is that it can be applied, in some cases, to specific parts of a program, instead of the program as a whole. This generally requires programs to be broken up into parts (e.g., classes or components) with well-defined interfaces and the ability to to hide implementation details behind interfaces.

Program-based approaches can be broadly categorised into several groups. We identify the following.

– **Annotation-based approaches**, wherein a program is annotated with additional information – such as pre- and postconditions – either manually or automatically. This information can then be used to help support formal analysis, reasoning, and verification. We explore several annotation-based approaches in Section 3.1.

– **Abstraction-based approaches**, where a formal specification is automatically or semi-automatically generated from a program (which may need to be written in a subset of a full programming language like C++). The abstraction can then be subjected to formal analyses, such as model checking or theorem proving. We consider examples of these approaches in Section 3.2.

– **Test-based approaches**, which attempt to generate test information from annotated programs. We discuss an example of test-based approaches in Section 3.3.

3.1 Annotation-based approaches

Professional programmers are normally very experienced with a selection of programming languages: they are familiar with the libraries provided with the languages, and understand the often-times complex semantics and emergent behaviour of the languages’ constructs. One criticism levelled against certain formal specification languages — such as Z or CSP — is that they require programmers to learn new, often quite different, languages and different ways of thinking about programming. On a practical level, using such formal languages can make it difficult for programmers to make use of the libraries that they are used to.

Annotation-based approaches attempt to surmount these difficulties by allowing programmers to work with languages they are used to — such as Java, C#, and Ada — while still providing the means to write and reason about precise specifications. Annotation-based approaches operate on the principle that an existing language, such as Java, is extended with a flexible annotation mechanism that allows properties to be attached to parts of programs. Annotations are often expressed as boolean expressions or predicates, but in general can be expressed in a rich language of their own, which includes the abilities to write different kinds of annotations, comments on annotations, and information to be used in a reasoning process.

We now briefly discuss a selection of different annotation-based languages that see widespread use, and then consider the different kinds of reasoning that these approaches support.
Languages that support annotations can be categorised in two groups: those that have been extended with annotations (e.g., via pre-processors, external tools and libraries), and those that have supported formal annotations as first-class constructs from initial language conception.

A large number of mainstream programming languages have been extended with annotations to support formal program development. Perhaps the most well known are Java, C#, and Ada but in general annotation support for most programming languages exists at different levels of sophistication. The extensions all generally have the following characteristics:

- support for specific kinds of annotations, most notably preconditions and postconditions of functions or methods, and invariants on modules, classes or objects.
- mechanisms for indicating whether annotations should be evaluated or not during execution.
- extensions to the boolean expression language subset of the programming language, to include more abstract, and typically declarative, styles of specification, e.g., iterative operations on sets, quantifiers, higher-order functions.

The Eiffel programming language [36] is probably the first modern language to fully integrate mathematical annotations from initial language conception. Eiffel supports preconditions, postconditions, class invariants, and general assertions within routines, as well as mechanisms for turning assertion checking on and off at run-time. Annotations in Eiffel are predominantly used as a run-time reasoning mechanism, as part of a rigorous testing process, but work is ongoing [40, 46] on reasoning theories and tools for Eiffel programs.

We now briefly discuss several alternative approaches that add annotations to existing programming languages.

**Java and JML.** The Java Modelling Language (JML) [10] is an extension of Java. It allows specification of behaviour of Java classes and methods, and is derived from the Eiffel programming language and the refinement calculi of Back, Morgan [37], and Hehner [20]. JML annotations are a superset of those in Java, building on Java’s expression language to include abstract specification facilities like model variables, frame conditions, and others. JML is also supported by a selection of powerful tools. Beyond a syntax checker, there is also a run-time assertion checker, and a selection of static checkers like ESC/Java2 [30] (which supports fully automated reasoning about a selection of properties), the LOOP tool (which automatically generates PVS theories from JML specifications), and the JACK tool (which supports a subset of JML and Java).

A snapshot of a JML specification is shown in Listing 1.5.

```
/* @ requires amount >= 0;
   @ assignable balance;
   @ ensures balance == \old(balance) - amount
   @ & & \result == balance;
 */
```

**Listing 1.5.** Example JML specification
```java
int withdraw(int amount) {
    if (amount <= balance) {
        balance -= amount;
        return balance;
    } else {
        throw new BankException("overdrawn by "+ amount);
    }
}
```

The keywords requires and ensures indicate pre- and postconditions, respectively, whereas the assignable keyword indicates that the method to which the annotations are attached can be assigned to directly.

When working with JML and Java, a programmer can make use of existing Java libraries, as well as a number of JML-specific libraries (e.g., for sets and other abstract data structures). Tools also exist to help the programmer in constructing annotations, e.g., the Daikon invariant detector (which will be discussed later).

**C# and Spec#** Spec# [34] is an annotation-based extension to the C# programming language, and is very similar in many respects to JML. It provides preconditions and postconditions, like JML, as well as class invariants and non-null types (i.e., variables that must always have an object assigned to them). Spec# is supported by several tools, including the Spec# compiler, integrated within Microsoft’s Visual Studio, and the Spec# static program verifier. The Spec# compiler serves a similar purpose to JML’s run-time assertion checker, as it compiles Spec# statements into run-time checks in pure C# code. The static program verifier for Spec# generates verification conditions from Spec# programs, which are then used internally to attempt to find errors in Spec# code, or to prove that a program satisfies its annotations.

Listing 1.6 shows the same example used in Listing 1.5, but expressed in Spec#’s syntax, which is very similar to that used in JML.

**Listing 1.6. Example Spec# specification**

```csharp
public static int withdraw(int amount) {
    requires amount >= 0;
    modifies balance;
    ensures balance == old(balance) - amount;
    ensures result == balance;
    {
        if (amount <= balance) {
            balance -= amount;
            return balance;
        } else {
            throw new BankException("overdrawn by "+ amount);
        }
    }
}
```

Spec# introduces the modifies syntax, which indicates the frame of the specification, i.e., the state that the specification (in this case, the method withdraw) can modify.
The JML and Spec# are collaborating to exchange ideas and techniques. Where do the two approaches differ, beyond the programming language that they extend? Spec# has been designed, in particular, to be easy to use to transition to C# programs, and within the Visual Studio toolset. A result of this is that the interface to Spec# attempts to appear much as a standard debugger would—when a proof obligation or conjecture fails, the corresponding fragment of Spec# is highlighted. In general, Spec#’s designers have made pragmatic design decisions—such as not checking frame conditions at runtime—to make integration with Visual Studio and transition to C# easier. As well, Spec# has paid a great deal of attention to correct reasoning about object invariants, and support a flexible notion of when an object invariant should hold (e.g., through expose constructs). Further, Spec# is supporting a notion of ownership and ownership transfer that should make it easier to support compositional reasoning. However, there is clear cross-pollination of ideas between the approaches.

Ada and SPARK Ada SPARK is an extension of a subset of the Ada language, originally designed to support the development of safety-critical and high-integrity systems. SPARK was designed to be logically sound (a requirement if it was to be used for building high-integrity systems), as well as to support automated verification. SPARK excludes a number of constructs of Ada that are known to be difficult to use for verification, including tasks, goto statements, and type aliases. It also mandates a specific set of annotations to be used for formal verification. These include annotations designed to make dependencies between variables (i.e., which define the relationship between imported variables in a module and variables exported from a module) explicit, which can make formal reasoning much easier to do.

SPARK also supports the notion of a proof context, which is annotation designed to support formal proof. Types of proof contexts in SPARK include preconditions and postconditions, assert statements (e.g., for loop invariants) and check statements (located between program statements).

SPARK programs are designed to be checked by tools. A SPARK program will be accepted by the SPARK Examiner toolset, which is used to generate verification conditions, check that the SPARK program is well formed (e.g., that is has all required annotations), and that the program conforms to its proof contexts. The SPARK Examiner can be used in concert with the SPADE proof checker to prove that a program satisfies its annotations.

A key difference between SPARK, JML, and Spec# is that the former has been designed for soundness, a requirement of the domain in which it is to be used. JML and Spec# have not been designed with this as a hard requirement.

An example of a SPARK program for the bank account example is shown in Listing 1.7

```
procedure withdraw(amount: in Integer);
  -- # global balance;
  -- # derives balance from amount, balance;
  -- # pre amount >= 0;
  -- # post balance = balance - amount;
```

Listing 1.7. SPARK Ada example
Other annotation-based approaches  Annotation extensions exist for many other programming languages, including C++, C, Python, Perl, Sather, Scheme, Ruby, and Common Lisp. The annotation approach to formal program design has also proved influential in design and analysis, e.g., through the addition of the Object Constraint Language (OCL) to the UML family of standards.

Reasoning about annotated programs  The Eiffel, JML, Spec#, and SPARK approaches to program development all come with supporting tools. Some of these tools are lightweight and designed to simply check syntax and type usage. Others aim at supporting reasoning about the annotated programs. Reasoning about annotated programs takes three general forms:

- run-time reasoning (e.g., via testing), in which annotations (such as preconditions and postconditions) are transformed automatically into executable statements in the annotated language (i.e., Java, C#, and SPARK Ada, respectively) or its underlying object language (e.g., Eiffel). The program, with executable annotations, is then executed and errors can then potentially be detected through failures or exceptions raised at run-time. This approach can normally be applied without the need for advanced reasoning technology like theorem provers.

- static reasoning, wherein theorem proving technology is used either behind-the-scenes (e.g., Spec#) or up-front (e.g., the LOOP tool for JML, SPADE tool for SPARK) to prove that the program satisfies properties captured in annotations. Efforts on Eiffel [46] are also proceeding in this direction. The advantage of static reasoning over run-time reasoning is that errors can be caught earlier, e.g., prior to deployment, and established once-and-for-all, rather than for specific values used for a particular program execution.

- to automatically generate tests from the programs, and to use these tests to exercise the programs and the annotations. This approach is taken with the AutoTest tool, and will be discussed in more detail later.

Large-scale applications of reasoning may combine all three approaches within a development process.

3.2 Abstraction-based approaches

Executable programs, in modern third-generation programming languages, are complicated and complex entities. Reasoning about programs written using the full suite of programming constructs —such as pointers, references, generics, and external libraries—can be very difficult, since not only is it necessary to reason about the program itself, but the context in which it is executing. This often entails enriching theories with information about external libraries and APIs, operating system details, and possibly even details of hardware.

For this reason, reasoning about programs written in full-featured programming languages is not often done directly. Instead, abstractions of programs are reasoned about. Abstractions of programs can be produced in many ways. But the fact that programs are written in a specific concrete syntax, which can be manipulated by tools, means that
it is often feasible to automatically generate abstractions of different kinds, in order to support reasoning.

We now discuss two successful approaches to abstraction: the Daikon invariant detector, and Verisoft.

**Invariant detection and Daikon** In programming, an invariant is a condition that evaluates to *true* at a specific point in the execution of a program. Invariants are used to document a program, to provide a basis for formal reasoning (e.g., to help prove that a program satisfies desirable conditions), and to constrain reuse and modification of a program (i.e., by capturing conditions that must be maintained, even if the program were to be changed).

Annotation-based approaches (Section 3.1) can be used to express invariants and encode them in a program. In languages like Eiffel and JML, invariants are usually attached to classes or objects, and are evaluated at run-time. The use of invariants in annotation-based approaches focuses on their application during design. But many modern applications integrate legacy components and legacy programs, which may not include annotations and invariants. Moreover, access to source code may not be possible, and thus it may be difficult to actually carry out annotation of programs with invariants and other properties.

Daikon [17] is an automated invariant detector for programs; it currently can detect invariants for Java, C, C++ and Perl programs. Given a program (in some cases without source code), the Daikon system *instruments* the program automatically and then executes the program. The instrumentation process effectively allows program variables to be examined (and invariants involving these variables to be inferred). Instrumenters are language-specific; some are source based, others can instrument object code.

The Daikon instrumenter produces variable trace data which is then passed automatically to Daikon’s detection engine, which generates potential invariants from the trace data. The basic approach taken is to generate a potential invariant, then to check the invariant against trace data. Optimisations are applied to eliminate redundant invariants. The inference engine can detect many different patterns of invariant, including non-null variables (e.g., `var != null`), orderings, and containment. The list of patterns that Daikon can detect can be extended by knowledgeable users.

For example, consider a typical Java interface of a fixed-size Stack, implemented using an array:

```java
Object[] contents; // elements of the stack
int top;          // index of top element, -1 if empty
```

When the Daikon system is applied to such a stack (which possesses usual operations, such as `push(x), pop()`, and `isFull()`) invariants such as the following will be automatically generated:

```plaintext
contents != null
top >= -1
top <= contents.length - 1
```
Note that Daikon can detect invariants that involve both public (instrumented) variables, as well as variables that are not explicitly used by the programmer (such as the length of the array).

Output from Daikon can be in a variety of formats, including that for input to other tools, such as pure Java, JML, and Parasoft’s JTest tool. Thus, Daikon feeds in to other phases of the software engineering lifecycle.

There are other tools for invariant detection, including the Houdini tool [19] (which uses theorem proving to disprove unsatisfiable guessed conditions, thus allowing unsatisfiable invariants to be refined to satisfiable ones). Agitator [7] aims to improve tests by calculating invariants. Henkel et al [21] have generated algebraic specifications from code, particularly for use in testing and for generating test cases.

Invariant detection is a promising technique that can provide an effective way of gradually adopting formal methods at the programming phase, even when dealing with substantial programs and legacy systems.

Verisoft

3.3 Test-based approaches

When software is annotated with formal specifications, such as pre- and postconditions, the specifications can be used for several tasks, including proof of correctness (e.g., showing that a program satisfies its annotations). However, proof of correctness, when applied in practice, often needs to be used with other verification and validation techniques, particularly testing. While testing and proof of correctness conceptually target the same problems – demonstrating validity and correctness – they achieve it in different ways, and in practice each also achieves things that the other does not. For example, proof of correctness is able to demonstrate completeness, i.e., that over all inputs the program produces an acceptable output; testing is able to demonstrate coverage, i.e., that all program paths in an execution are reached.

When programs are annotated with formal specifications, the complementary, yet consistent nature of testing and formal methods can be brought to bear. In particular, formal specifications can be used as the basis for test generation, e.g., as exemplified by Parasoft’s JTest tool [41] or Korat [8]. These tests may then be supplemented by user-written tests, and the testing process can then be automated. However, a test oracle is necessary.

The AutoTest project [32] has focused on fully automated testing of Eiffel programs [36]. Eiffel programs generally include class invariants, and method pre- and postconditions. These annotations act as test oracles and as the source for test case generation. The AutoTest process is fully automated: from test case generation, to running the tests, to using the test oracles, to generating a report.

Generating test cases from Eiffel annotations (i.e., contracts) is non-trivial, because most annotations in programming languages are incomplete, and some may also be incorrect. Moreover, there are other issues:

– To test a method of a class, the generated test case must satisfy the precondition. If the precondition is very strong, then an automated test case generator may take a very long time to produce acceptable values.
In a language like Eiffel (and in many other languages), postconditions should specify what attributes of an object can be changed by a method; this is called the frame, and knowing the frame is helpful for test case generation. However, Eiffel’s support for specifying frames is still developing.

AutoTest builds on random testing, i.e., random test case generation. When testing a method, AutoTest first generates a method target (i.e., the object on which the method is called) and suitable arguments randomly. Then, if the generated data satisfies the method precondition, the method is executed. If a violation of an annotation occurs, there is a bug, and a suitable report entry is generated, otherwise the test passes. For example, consider the following segment of Eiffel code.

```eiffel
class BANK_ACCOUNT
feature
make
  end
  balance := 300
end

balance : INTEGER
deposit (d: INTEGER)
  do
    ensure
      balance > old balance
      balance = old balance + d
  end
invariant
  balance >= 0
end
```

AutoTest would generate a test case for the method `deposit` as follows.

```eiffel
class TEST_CASE
feature
test1
  local
    ba: BANK_ACCOUNT
  do
    ba := new_object("BANK_ACCOUNT")
    set_field (ba, "balance", 300)
    check_invariant (ba)
    ba.deposit (50)
  end
end
```

AutoTest uses reflective techniques to generate test cases that create the necessary objects and arguments, and invoke the relevant methods. Of course, the test case above will fail (there is no implementation of `deposit` yet).
Work related to AutoTest is exploring how to record test cases in test suites; the CDD toolset [33] provides this ability, which is important so that developers can repair failed tests when they choose to do so.

AutoTest is an excellent example of how formal methods can be used to help to improve the quality of programs, and how they can be used unintrusively and predominantly automatically, thus disrupting developers’ ways of working as little as possible.

There are other important testing tools available that build on formal specifications and formal methods; we have already mentioned Korat and JTest. Other important tools include Java Pathfinder (JPF) [50] and Agitator [7]. JPF, for example, uses a range of techniques to systematically check all possible execution paths. In common with other model checking approaches, many systems result in an infeasibly large state-space to search. JPF thus uses heuristics and abstraction techniques to reduce the number of states it really has to examine. JPF is notable in that it checks systems written in a widely-deployed language (Java) with relatively few constraints (e.g., native methods cannot be checked).

4 Summary

References