Programs, Properties, and Data: Exploring the Software Development Trilogy

Daniel Le Métayer — Valérie-Anne Nicolas — Olivier Ridoux

IRISA / INRIA
Campus de Beaulieu
35042 Rennes France
email: {lemetayer,vnicolas,ridoux}@irisa.fr

Abstract Software development usually involves a collection of properties, programs and data as input or output documents. Putting these three kinds of documents at the vertices of a triangle, one sees that all three sides of the triangle have been exploited in formal methods, and that they have often been used in both directions. However, richer combinations have seldom been envisaged, and formal methods often amount to a strict orientation of the figure by imposing functional dependencies (e.g., infering test cases from specifications). Moreover, undecidability problems arise when properties are expressed in full predicate logic (or similar formalisms) or programs are written in Turing-equivalent programming languages. We advocate that (1) formal methods should provide more flexible ways to exploit the developer's knowledge and offer a variety of possibilities to construct programs, properties and test data and (2) it is worth restricting the power of logic formalisms and programming languages for the benefit of mechanization. We go one step in this direction, and present a formal method for generating test cases that combines techniques from abstract interpretation (*program→property*) and testing (*program+property→test data*), and takes inspiration from automated learning (test generation via a *testing bias*). The crucial property of the test suites generated this way is that they are robust with respect to a test objective formalized as a property. In other words, if a program passes the test suite, then it is guaranteed to satisfy the property. As this process leads to decision problems in very restricted formalisms, it can be fully mechanized.

Keywords Software engineering, testing, verification, program analysis, program learning.

1 Introduction

The only way to improve confidence that a software really achieves its intended purpose is to confront it with other means of expressing this purpose. Typically, such means can be properties that a software is supposed to satisfy or test suites with oracles characterizing the expected behavior of a software. *Programs*,

properties and *data* can thus be seen as three facets of the software system that must be related through specific coherence relations (Figure 1). This multi-facet view of a system does not impose any ordering in the design of each facet. As a consequence, a software development process should allow any possible choice (and iterations) depending on the feelings and the initial knowledge of the developers[1].

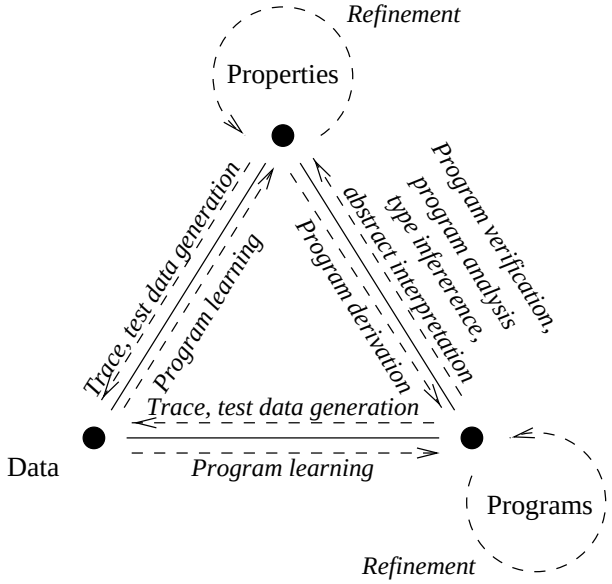

Figure 1: The software development trilogy

Properties, programs and data are, by essence, formal objects and the coherence relations between them can also be defined in a formal way (Figure 1). It is the case that formalization is usually not emphasised in industrial environments, but we believe that even partial, modest formalization efforts can have a positive impact on the whole software development cycle. For example, formalizing the coherence relations between the three facets of software mentioned above is helpful in both the development and the maintenance phase. In the development phase, it can help reducing the possible design choices; in the course of maintenance (or in case of evolution) of the software, it is necessary to detect the potential impact (on all facets) of the modification of a facet. Typical applications are the detection of which test cases or properties are affected by a change in the program (regression testing or verification). So, a formal method should encompass all possible functional dependencies between the three kinds of documents.

[1]Note that any development process starts in fact with a definition of the requirements; but requirements are, by definition, not expressed in a formal way, so we do not consider them here (otherwise they should just be added as an initial phase).

We are thus interested in the formal aspects of this three-facet approach, but with the aim that such a formalization should be effective in the sense that it should provide some help (ideally supported by a tool) or guidance in the development process (including maintenance).

In the following, we review in the first two sections (2 and 3) the various ways of describing programs, properties and data and to exploit them in the development process. In Section 4, we sketch an automatic test generation method that exploits a richer combination of these methods and tools. Further work is discussed in the conclusion.

2 The formal documents

In this section, we review the three kinds of documents mentioned in the introduction (programs, properties and data) and we discuss the different ways to specify them.

2.1 Programs

Programs are usually considered as the final outcome of software development. They are definitely formal documents in the sense that they must be fed into fully automatic tools like interpreters or compilers.

The formalization of programs is given by the syntax and semantics of the programming language they are written in. The syntax is the part that is usually best formalized. The semantics of most widely used programming languages is not fully formalized or it is formalized in a loosely way, e.g., leaving evaluation order of expressions unspecified. The main frameworks for specifying the semantics of a programming language are denotational semantics, operational semantics [NN92] and axiomatics semantics [Hoa69]. Each of these semantics styles has been used as a basis for program analysis and program verification.

Most general purpose programming languages are in fact Turing-complete (*while* statements or recursion are sufficient to get Turing-completeness). Turing-completeness ensures that the language is expressive enough (at least in a theoretical sense); but it also has a cost: most decision problems whose inputs are programs have no algorithmic solution. In particular, a basic result of computability theory is that no really "interesting" (non-trivial) property can be extracted automatically from a program in a Turing-complete language (e.g., the "halting problem").

It is quite a challenge to design a programming language that would not be Turing-complete, but would still be expressive enough. One promising research direction in this context is the design of "domain specific languages", which are suited to well identified classes of applications.

2.2 Properties

The specification expresses the expected property of the software. In idealized formal development processes (e.g., V-shaped process, waterfall process, etc), specifications are supposed to be provided in a first stage, before the design of the program. In this context, the language used to express properties is typically a very expressive logical formalism (predicate calculus, higher-order types systems). Another possibility is to extract properties from programs. When the language of properties is sufficiently restricted, this process can be achieved by automatic tools (program analyzers or type inference systems).

Specifications can also be executable, in which case they behave both like properties and programs. More generally, the difference between programs and specifications is not always clear-cut. Executable specifications are useful for animating the early design of a program, or for mechanizing a part of the testing process. In the latter case, specifications can be used as oracles for checking that the actual behavior of a program is the expected one.

2.3 Data

Data can be either input/output data, or internal data. Traces are sequences of data which can be extracted from programs using an instrumented interpreter. Checking that a trace conforms with the expected computation can be done either manually or with the help of a browsing tool.

Traces may also exist before programs, as a means of specifying them. In this case, they are often called *scenarios*. These scenarios can be considered as examples for a program learning process [Bie78].

Pairs of input and output data correspond to test cases or to learning examples. In the first case, the program is run with the inputs and the computed outputs are checked with respect to the expected results. As for traces, the checking can be done manually, the programmer playing the role of the oracle, or (semi-)automatically, when the oracle can be implemented by an executable specification.

3 Methods and tools

In this section, we consider in turn how each side AB of the triangle of Figure 1 can be formalized as relations between A and B.

3.1 Properties → Programs

Going from properties (specifications) to programs is the basic idea of a top-down development process. Formally deriving programs from specifications is a very early endeavor of the science of programming [Dij75]. Note that though it has gained little acceptance in the general case, it is used mechanically in every day work when the specification language is specialized enough, so that program derivation is more like compilation or program translation (derivation

of syntactic analyzer, GUI programs, etc). Even in less favorable situations, there exists now semi-automated systems that help in deriving programs from specifications. For example, the B method [Abr96] is very successful in the French railway industry (typically for the design of secure train control systems).

3.2 Programs → Properties

Different techniques have been proposed to extract properties from programs: programming language axiomatization [Hoa69], abstract interpretation [CC77], or type inference [Mil78] for example. The last two approaches have in common that they deal with restricted logics. In the case of abstract interpretation, the restriction is expressed in terms of abstract domains; in the case of type inference, it is fixed by the language of types.

3.3 Programs → Data

Structural (or *white-box*) testing [Bei90] methods are based on the following strategy: first a test criterion is chosen (e.g., all instructions or all definition-uses paths); then test suites are produced by the tester and the program is executed on the values of the test suite in an environment that measures coverage[2] (according to the chosen criterion); the process is iterated until a suitable coverage is achieved. Another (more challenging) option consists in producing the test suites mechanically [DO91]: paths that fulfill the test criterion are selected and the conditions that occur on them are accumulated as constraints; these constraints are then solved to compute input values. This process requires a constraint solver that is powerful enough to compute solutions when they exist, and to detect that none exists when it is the case (inaccessible paths). Note that detecting that the generated set of constraints has no solution (which means that it contains a contradiction) is out of reach when the language for expressing conditions is too rich (e.g., Turing-complete). This approach is quantitative in the sense that the confidence in the program grows with the coverage of the test criterion. However, there is no formal relation between the fact that a program passes a test suite generated in this way and its partial or total correctness.

3.4 Data → Properties or Programs

Synthesizing programs from examples [Bie78] is one of the first attempts for mechanizing the construction of programs. This trend of research is concerned with both the learning procedures and the classes of functions that can be learned. The framework that is common to all these methods is called *inductive inference*. In particular, there is no essential difference between learning programs and learning specifications.

The programs that are learned belong in fact to restricted fragments of a programming language. This is because inductive learning uses a notion of

[2]The coverage of a structural test is measured as a percentage of instructions of the program, of all paths, of specific paths, etc.

learning bias to narrow the search space. The role of the learning bias is to make the learning process feasible and as efficient as possible. A learning bias restricts the language in which the concept to be learned is expressed. Such a bias makes the learning process incomplete because the intended concept may be better expressed outside the learning bias, or even may not be expressible in the learning bias. However, a learning bias establishes a formal relation between the examples and the concept that is learned.

There is another way in which one may go from data (and a program) to another program. This is when a program does not pass a test case. The data (i.e., the test case) can be used to locate the error. There are two methods. First, the data can be used to select the part of the program that it exercises. This part must contain the bug; so the programmer can focus on this *slice* of the program [Tip95] to locate it. Second, the data and an oracle (e.g., the specification or the programmer) can be used to select a narrower part of the program that contains the error. This is known as *declarative debugging*, and has been well-explored for Prolog programs [Sha83]. Because this method uses both the failed test cases and an oracle as inputs (in addition to the original program), it does not lay strictly on the *Data*→*Program* edge of Figure 1. It is better described as laying on an hyper-edge {*Data, Program, Property*}→*Program.*

3.5 Properties → Data

The testing phase should not be neglected even when the development starts with a completely formalized specification and the derivation process is completely formal. This is because the specification itself might be wrong, and the only way to discover the problem is to confront the specification with another property. Testing the final product will reveal the error, but much too late. So, specifications must also be tested. This is called *functional* (or *black-box*) testing.

As for program testing (Section 3.3), test suites can be generated manually or mechanically. A notion of coverage also applies to specification testing. However, formal approaches consider test hypotheses [BGM91] or test objectives [VBL97] that are written in the specification language. These approaches can be called *qualitative* in the sense that the confidence is not qualified by a coverage percentage; it is rather characterized in a logical way by the test hypothesis or the test objectives, which provides a precise definition of the limits of the test.

3.6 Properties→ Properties

As we have seen in Section 2.2, there is a continuum between programs and specifications. Some formal methods provide a framework for refining specifications iteratively until a fully explicit version is reached, which can be considered as a program. If each refinement step preserves the specification, then the whole process yields a correct program. These methods are often only semi-automated. A recent example of this is the B method [Abr96].

3.7 Programs → Programs

Compilation is the most frequent way of deriving a program from another one. In this case the output is generally not meant to be read by a human being. Compilation can be a purely syntactic process during which a program is translated into another one, but it can also take into account the semantics of the programming language for optimizing the result.

Programs can also be transformed in order to extend their functionality or to derive higher quality versions. One trend of research falling into the second category is partial evaluation [CD93] which promotes the design of generic programs that can be used in a variety of contexts and specialized mechanically in order to recover an acceptable level of efficiency.

4 A robust test process

In this section we present a test generation method that makes a sophisticated usage of a number of the possibilities enumerated above. Our method generates test suites that are both finite and robust with respect to a given property (which means that a program passing the test suite is guaranteed to satisfy the given property). Furthermore, it is fully automatic. This goal can be achieved through careful restrictions on the class of programs and the class of properties considered.

4.1 Learning vs. testing

We have seen that all sides of the software triangle of Figure 1 have been explored in both directions. In particular, tools and methods exist for synthesizing programs from examples or traces, and other tools and methods exist for synthesizing test suites from programs. However, a deeper examination of these two activities shows that they are based on very different hypotheses.

In the learning activity, a finite suite of examples or traces is used to generate a program that is guaranted to satisfy the examples. In other words, if one considers the suite of examples as a test suite, it is certain that the generated program will pass it. Note also that the input document is assumed to be correct. A learning *bias* is used to narrow the search space and to infer only regular programs. This restriction is in the same spirit as testing hypotheses, which also assume regularity in the tested programs.

In the context of structural testing, test cases are generated from a program according to a test criterium (e.g., all-instructions, all-def-use). But finite test suites are generally not robust (which means that they can accept incorrect programs). Finally, the input document (the program) cannot be assumed to be correct. This shows that the situation is less favourable than in the context of program learning. The basic reason is that classical approaches to the generation of test cases deal with general programs (i.e., written in Turing-complete languages), whereas program learning deals with biased programs.

A solution to make the situation of test cases generation more favourable is to borrow from program learning some of its hypotheses. We will call *testing bias* a syntactic restriction that corresponds in testing to a *learning bias* in program learning. We will use in the following section a hierarchy of testing biases. Their most important property is that a testing bias characterizes an infinite set of programs which can be discriminated from each others by a finite test suite. Moreover, in contrast with test hypotheses, testing biases are syntactic, so they can be checked mechanically.

4.2 The test process

We present a testing process that produces robust test suites and can still be mechanized. The key idea is to lower our ambition to prove the complete correctness of a program *via* testing. Instead, *partial correctness* properties will be proved *via* testing.

The core of this testing process is as follow:

1. The property to be checked is provided as input (in addition to the program). This property must be written in a restricted language.
2. The program to be tested is abstracted (automatically) into the domain of properties.
3. A common testing bias for the abstract program and the property is computed.
4. The associated abstract test suite is derived from the testing bias.
5. A concrete test suite is computed from the abstract test suite.

If the program passes the concrete test suite successfully, then it is correct with respect to the input property. A key point here is that the steps 2 to 5 are carried out automatically.

The whole process is schematized in Figure 2, and illustrated step by step in the following example. It has been implemented in a prototype which accepts input programs and properties written in a dialect of functional programming, and produces a representation of the common testing bias, with an explanation, and a robust test suite.

The first input document is the program to be tested. This is the lower-right vertex in Figure 2. There is no constraint, syntactic or semantic, on programs at this level. Consider for instance a selection sort program written in a simple functional language:

$$
\begin{array}{lcl}
Selsort(nil) & = & nil \\
Selsort(x:l) & = & \textbf{let } (x_1, l_1) = Maxl(l, x) \\
 & & \textbf{in } x_1 : Selsort(l_1)
\end{array}
$$

$$
\begin{array}{lcl}
Maxl(nil, y) & = & (y, nil) \\
Maxl(x:l, y) & = & \textbf{let } (x_1, l_1) = Maxl(l, x) \\
 & & \textbf{in } (Max(y, x_1), Min(y, x_1) : l_1)
\end{array}
$$

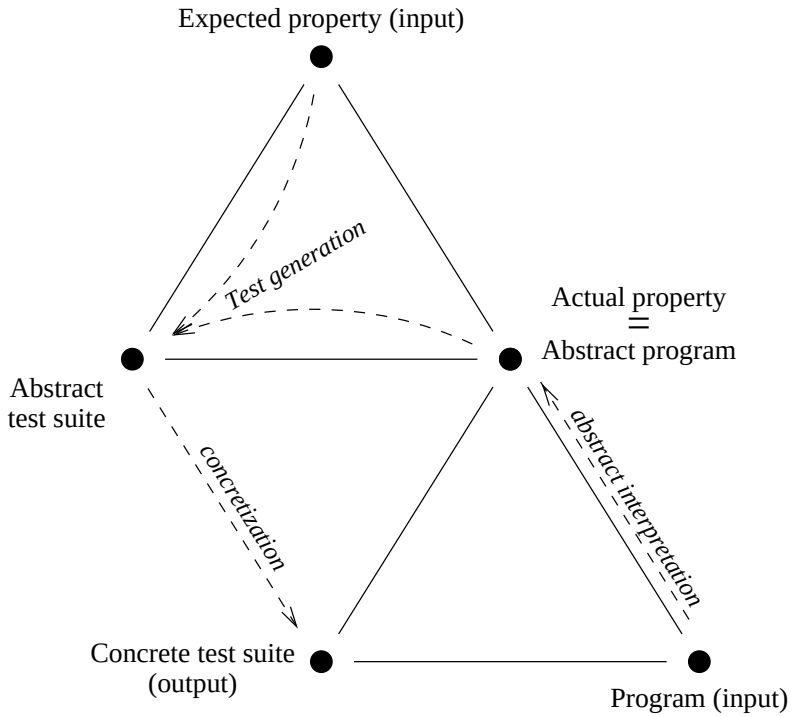

Figure 2: The method

In this example, we focus on a partial correctness property: $Selsort$ should preserve the length of input lists. The property to be proven must be provided in a restricted language. This forms the second input document (top vertex in Figure 2). All formal documents in the upper triangle must belong to restricted languages so that decision problems that are introduced by the method are decidable. In our example, the fact that a program preserves the length of input lists can be encoded as the identity function:

$$\begin{array}{lcl} Ident(0) & = & 0 \\ Ident(n+1) & = & 1 + Ident(n) \end{array}$$

The intuition is that $Ident(n)$ represents the expected length of the result of $Selsort$ when applied to an argument of length n.

The actual property of the program must be extracted from it *via* an abstraction. This process is automatic; it is based on the well-established framework of abstract interpretation [CC77].

In our example, lists are abstracted by their lengths (nil is abstracted into 0 and $x : l$ into $1 + n$, where n represents the length of l):

$$
\begin{array}{lcl}
Lselsort(0) & = & 0 \\
Lselsort(n+1) & = & 1 + Lselsort(Lmaxl(n))
\end{array}
$$

$$
\begin{array}{lcl}
Lmaxl(0) & = & 0 \\
Lmaxl(n+1) & = & 1 + Lmaxl(n)
\end{array}
$$

We have now two versions of the property of interest. The first one ($Ident$) is the test objective and the second one ($Lselsort$) is the property that is effectively satisfied by the program. The next step is to find a testing bias which contains both versions. Here, testing biases are defined as recursive function schemes and the test suite corresponding to the scheme including both $Ident$ and $Lselsort$ is $\{0, 1, 2, 3\}$. This test suite is expressed in the abstract domain. It is possible to concretize it so that it can be provided as input to the original program. In our example, a correct concrete test suite is made of four randomly chosen lists of lengths 0, 1, 2, and 3 respectively.

The program $Selsort$ can then be tested on these four lists and the length of its results automatically compared with the value indicated by $Ident$.

Note that a standard proof of this property (that program $Selsort$ respects the length) would have used an induction on the length of the argument. So, it would have required a non-trivial proof technique, while our method boils down to the comparison of test outputs. However, it should be clear that this verification process does not prove that the program is correct. It only proves that it preserves the lengths of its arguments. In particular, a function that implements the identity on lists instead of a sort will also pass the test.

5 Conclusion

Several directions that we have described in the previous sections have already been studied in detail and implemented as tools. We think that the systematic and concerted exploration of the program-property-data trilogy should lead to richer classes of software development environments. These new environments would consider properties, data, and programs on a par, without requiring that one is necessarily completed before the others. Providing information about any of these kinds of documents would then be considered as a way to progressively narrow the software design space. This would lead to a more flexible formal development process, offering various ways to adapt to the developer's initial knowledge or personal preferences. The key issue in this context is in fact to maintain the coherence between the formal documents.

The relation between program learning and testing has been recognized in the past by several authors [Wey83, BG96]. In fact, Bergadano *et al.* actually use program learning as a means for generating test suites. In their case, a program learning process generates incrementally a family of programs that are "close" to the program to be tested. Each time a new program is produced, a new test case is added to the set of examples. The new test case must be

such that it distinguishes the new program from the program to be tested. Our technique does not actually perform program learning. We mainly use it as a fruitful metaphor to establish a tight relation between a finite set of input/output data, a program and a property. The key idea is that the program must belong to a "learnable" family. On the contrary, usual testing theories either involve infinite sets of data, or lack a well defined relation between test suites and programs.

We would like to stress again the importance of studying restricted languages and their impact on the design of automatic software development tools. Classical programming languages are equivalent to Turing machines, which is the origin of a never-ending stream of problems: no formal tool will ever exist for proving such properties as termination, non-violation of array bounds, etc. Very rich fragments of the λ-calculus have been proposed in which all programs terminate [PDM89]. They are not popular as programming languages, but we expect that one can build upon them sophisticated and mostly automatic software development tools. This should raise some interest for these languages, especially if they can be connected to specific application areas, leading to practical domain specific programming languages.

References

[Abr96] J.R. Abrial. *The B-Book: assigning programs to meanings*. Cambridge University Press, 1996.

[Bei90] B. Beizer. *Software testing techniques*. Van Nostrand Reinhold, 1990. 2nd edition.

[BG96] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM trans. Software Engineering and Methodology*, 5(2):119–145, 1996.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6), 1991.

[Bie78] A. Biermann. The inference of regular LISP programs from examples. *IEEE trans. Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. Principles of Programming Languages*, pages 238–278, 1977.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. pages 493–501, 1993.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.

[DO91] R.A. DeMillo and A.J. Offutt. Constraint-based automatic test-data generation. *IEEE trans. Software Engineering*, 17(9), 1991.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10), 1969.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[NN92] H.R. Nielson and F. Nielson. *Semantics with Applications : A Formal Introduction*. John Wiley & Sons, Chichester, 1992.

[PDM89] B. Pierce, S. Dietzen, and S. Michaylov. Programming in higher-order typed lambda-calculi. Research Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, 1989.

[Sha83] E.Y. Shapiro. *Algorithmic Programming Debugging*. MIT Press, 1983.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[VBL97] L. Van Aertryck, M. Benveniste, and D. Le Métayer. CASTING: A formally based software test generation method. In *1st Int. Conf. Formal Engineering Methods*. IEEE, November 1997.

[Wey83] E.J. Weyuker. Assessing test data adequacy through program inference. *ACM trans. Programming Languages and Systems*, 5(4):641–655, 1983.