Chapter 6

R. Representation

Formal Methods

Motivation

The title of Truss's (2004) book on punctuation, *Eats Shoots and Leaves*, could refer to either:

> A panda, if the punctuation is as published, or
> A criminal who refuses to pay his restaurant bill if a comma is added after the word "eats."*

Clearly the title of the book is not that of a system or software specification, but this anecdote illustrates that simple punctuation differences can convey a dramatically different message or intent. As has been stated by many, with respect to specifications, "syntax is destiny."†

Systems have tremendous sensitivity to errors in a requirements specification, design document, or computer code; even a single erroneous character can have severe consequences. In fact, in 1962 a missing hyphen character in a FORTRAN code statement led to the loss of the Mariner 1 spacecraft, the first American probe to Venus (NASA 2012).

---

* Actually, the author heard the joke differently from members of the Royal Australian Air Force almost 20 years ago. As told, the panda is an amorous, but lazy creature and is well known for its ability to scope out a tree occupied by a panda of the opposite sex, where it "eats, roots, shoots, and leaves." In a double entendre, the middle portion of the quote refers to the act of procreation.

† The author heard this quote first from Dr. George Hacken, Senior Director, Vital Systems, New York City Transit, sometime around 2007.

Aside from punctuation, there are a number of problems with conventional software specifications built using only natural language and informal diagrams. These problems include ambiguities, where unclear language or diagrams leave too much open to interpretation; vagueness or insufficient detail; contradictions, that is, two or more requirements that cannot be simultaneously satisfied; incompleteness or any other kind of missing information; and mixed levels of abstraction where very detailed design elements appear alongside high-level system requirements. To illustrate, consider the following hypothetical requirement for a missile launching system.

> 5.1.1 If the LAUNCH-MISSILE signal is set to TRUE and the ABORT-MISSILE signal is set to TRUE then do not launch the missile, unless the ABORT-MISSILE signal is set to FALSE and the ABORT-MISSILE-OVERRIDE is also set to FALSE, in which case the missile is not to be launched.

This requirement is written in a confusing manner, and the complexity of the logic makes it difficult to know just exactly what the user intends. And if the user's intent is wrongly depicted in the language of the requirements, then the wrong system will be built.

It is because we need precision beyond that which can be offered by natural languages that we frequently need to reach for more powerful tools, which can only be offered by mathematics.

What Are Formal Methods?

Formal methods involve mathematical techniques. To be precise, we look at three definitions for "formal methods."

Definition 1: (Encyclopedia of Software Engineering, 1994)

> A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and, more relevant, specification, implementation, and correctness (Shehata and Eberlein 2010).

Definition 2: (IEEE Standard Glossary, 1990)

> 1. A specification written and approved in accordance with established standards; 2. A specification written in a formal notation often for use in proof of correctness.

Definition 3: (Dictionary of Computer Science, Engineering, and Technology, 2001)

> A software specification and production method based on a precise mathematical syntax and semantics that comprises:
>
> - A collection of mathematical notations addressing the specification, design, and development phases of software production, which can serve as a framework or adjunct for human engineering and design skills and experience
> - A well founded logical inference system in which formal verification proofs and proofs of other properties can be formulated
> - A methodological framework within which software may be developed from the specification in a formally verifiable manner
>
> Formal methods can be operational, denotational, or dual (hybrid).

It is clear from the three different but similar definitions that formal methods have a rigorous mathematical basis.

Formal methods differ from informal techniques, such as natural language, and informal diagrams such as flowcharts. The latter cannot be completely transliterated into a rigorous mathematical notation. Of course, all requirements specifications will have informal elements, and there is nothing wrong with this fact. However, there are apt to be elements of the system specification that will benefit from formalization.

Techniques that defy classification as either formal or informal because they have elements of both are considered to be semi-formal. For example, the UML version 1.0 is considered to be semi-formal because not all of its metamodels have precise mathematical equivalents.

But what advantage is there in applying a layer of potentially complex mathematics to the already complicated problem of behavioral description? The answer is given by Thomas E. Forster (2003), a giant of formal methods:

> One of the great insights of twentieth-century logic was that, in order to understand how formulae can bear the meanings they bear, we must first strip them of all those meanings so we can see the symbols as themselves . . . [T]hen we can ascribe meanings to them in a systematic way . . . That makes it possible to prove theorems about what sort of meanings can be borne by languages built out of those symbols.

Formal methods are especially useful in requirements specification because they can lead to unambiguous, concise, correct, complete, and consistent specifications. But as we have seen achieving these qualities is difficult, and even with

formalization correctness and completeness can be elusive qualities. A thorough discussion of these issues along with a very detailed example can be found in Bowen, Hinchey, and Vassev (2010).

Formal Methods Classification

There are several formal methods classes. The first class, model-based, provides an explicit definition of state and operations that transform the state. Typical model-based formal methods include Z, B, and the Vienna Development Method. The next class, algebraic methods, provides an implicit definition of operations without defining state. Algebraic methods include Larch, PLUSS, and OBJ. Process algebras provide explicit models of concurrent processes, representing behavior with constraints on allowable communication between processes. Among these are CSP and CCS. Logic-based formal methods use logic to describe properties of systems. Temporal and interval logics fall into this category. Finally, net-based formal methods offer implicit concurrent models in terms of data flow through a network, including conditions under which data can flow from one node to another. Petri nets are a widely used net-based formal method.

It should be noted that formal methods are different from mathematically based specifications. That is, specifications for many types of systems contain some formality in the mathematical expression of the underlying algorithms. Typical systems include process control, avionics, medical, and so on. Such use of mathematics does not constitute use of formal methods, although such situations may lend themselves to formal methods.

A Little History

Formal methods have been in use by software engineers for quite some time. The Backus–Naur (or "Normal") Form (BNF) is a mathematical specification language originally used to describe the Algol programming language in 1959. Since then a number of formal methods have evolved. These include:

- The Vienna Development Method (VDM), 1971
- Communicating Sequential Process (CSP), 1978
- Z (pronounced "zed"), late 1970s
- Pi-calculus, early 1990s
- B, late 1990s
- And many others

Finite state machines, Petri nets, and Statecharts or other techniques regularly used by systems and software engineers can be used formally. Other formal methods derive from general mathematical frameworks, such as category theory, and a number of domain-specific and general languages have been developed over the years, often with specialized compilers or other toolsets.

Formal methods are used throughout Europe, particularly in the United Kingdom, but not as widely in the United States. However, important adopters of formal methods include NASA, IBM, Lockheed, HP, and AT&T.

Using Formal Methods

Formal methods are used primarily for systems specification and verification. Users of UML 2.0 could rightly be said to be employing formal methods, but only in the specification sense. The languages B, VDM, Z, Larch, CSP, and Statecharts, and various temporal logics are typically used for systems specification. And although the clarity and precision of formal systems specification are strong endorsements for these, it is through verification methods that formal methods really show their power. Formal methods can be used for two kinds of verification: theorem proving (for program proving) and model checking (for requirements validation). We focus on the former, however.

Formal methods can be used in any setting, but they are generally used for safety critical systems, for COTS validation/verification (e.g., by contract), for high financial risk systems (e.g., in banking and securities), and anywhere that high-quality software systems are needed.

Formal methods activities include writing a specification using a formal notation, validating the specification, and then inspecting it with domain experts. Furthermore, one can perform automated analysis using theorem (program) proving or refine the specification to an implementation that provides semantics-preserving transformations to code. You can also verify that the implementation matches the specification (testing).

Another important use of formal methods is in modeling and analyzing requirements interaction. Requirements interactions can create conflicts between requirements due to differing stakeholder viewpoints. Formal methods such as finite-state machines, temporal logic, CSP, Z, and many others have been used in an attempt to resolve this vexing problem. An analysis of the challenges of identifying requirements interactions and research directions can be found in Shehata and Eberlein (2010).

Examples

To illustrate the use of formal methods in requirements engineering, we present a number of examples using several different techniques. Our purpose is not to present any one formal method and expect the reader to master it. Rather, we show a sampling of how various formal methods can be used to strengthen requirements engineering practice.

Formalization of Train Station in B

In their most straightforward use, formal methods can express system intent. In this use of formal methods, we exploit the conciseness and precision of mathematics to avoid the shortcomings of natural languages and informal diagrams. B is a model-based formal language that is considered an "executable specification" (collection of abstract machines) that can be translated to C++ or Ada source code. In this case, we use the modeling language B to provide a specification for a train station, similar to Paris's Line 14 (Meteor) and New York City's L Line (Canarsie). The model used is based on an example found in Lano (1996). Those familiar with Z will see a great deal of similarity in the structure and notation to B.*

The model begins as shown in Figure 6.1. The specification starts with the name of the machine and any included machines (in this case, **TrackSection**, which is not shown here). Then a list of variables is given and a set of program **INVARIANT**s, conditions that must hold throughout the execution of the program. Here **N** represents the natural numbers {0,1,2,3, . . .}. We continue with a partial description of the behavioral intent of this model.

Notice that the three variables of interest are the **platforms**, **max_trains**, and **trains_in_station**. The **INVARIANT** sets constraints on the number of trains in the station (must be a nonnegative number) and the maximum number of trains in the system (must be a nonnegative number and there can be no more than that number in the station). It continues with platforms as a sequence of train sections, in the range of the number of sections, which acts as an upper bound on the maximum number of trains (because only one train can occupy each section of track).

```
TRAIN STATION
MACHINE Station
INCLUDES TrackSection
VARIABLES
    platforms, max_trains, trains_in_station

INVARIANT
    trains_in_station ∈ N^
    max_trains ∈ N ^ trains_in_station ≤ max_trains ^
    platforms ∈ seq(sections) ^
    size(platforms) = card(ran(platforms)) ^
    max_trains ≤ size(platforms)
```

Figure 6.1 Set-up specification for train station.

* Thanks to Dr. George Hacken of the New York Metropolitan Transportation Authority for providing this example, which is derived from a 1999 talk he gave along with Sofia Georgiadis to the New Jersey Section of the IEEE Computer Society.


```
INITIALIZATION
    platforms, max_trains, trains_in_station : = [], 0, 0
```

Figure 6.2 Initialization specification for train station.

```
OPERATIONS
    train_arrives(ts) ≅
        PRE trains_in_station < max_trains ^ ts ran(platforms) ^
            tstate(ts) = free
        THEN
            arrival(ts) ||
            trains_in_station : = trains_in_station + 1
        END;
```

Figure 6.3 Train arrival behavior for station.

```
train_departs(ts) ≅
        PRE ts ran(platforms) ^
                tstates(ts) = blocked
        THEN
                departure(ts) ||
                trains_in_station : = trains_in_station – 1
        END;
```

Figure 6.4 Train departure behavior for station.

The next part of the B specification is the initialization section (Figure 6.2). This specification initializes the platform sequence to null and the maximum number of trains and trains in a station to zero.

Now we introduce a set of operations on the train station dealing with arrivals, departures, openings and closings, and resetting the maximum number of trains. For example, the arrival specification is shown in Figure 6.3. A precondition (**PRE**) before train **ts** can arrive is that there is room in the station for the train and that a platform is available. If so, mark the train as arriving and increment the count of trains in the station.

The next operation involves train departure (Figure 6.4). The precondition is that the train is in the range of the sequence of platforms (on a platform) and that the train is not moving (blocked). If so, then train **ts** is marked as having departed, and the number of trains in that station is decreased by one.

The next operation is the opening platform **ts** (Figure 6.5). The precondition is that the platform is in the range of the sequence of platforms for that station and that the current platform is closed. If so, then platform **ts** is marked as being opened.

The closing of a platform operation is similar (Figure 6.6). The precondition is that the platform is in the range of the sequence of platforms for that station

```
open_platform(ts) ≅
        PRE ts ∈ ran(platforms)
                tstate(ts) = closed
        THEN
                open(ts)
        END;
```

Figure 6.5 Platform opening behavior for station.

```
close_platform(ts) ≅
        PRE ts ran(platforms)
                tstate(ts) = free
        THEN
                close(ts)
        END;
```

Figure 6.6 Platform closing behavior for station.

```
set_max_trains(mt) ≅
        PRE mt N ^ mt ≤ size(platforms) ^
                trains_in_station ≤ mt
        THEN
                max_trains : = mt
        END
```

Figure 6.7 Behavior for setting the maximum number of trains for a train station.

and that the current platform is open. If so, then platform **ts** is marked as being closed.

Finally, an operation is needed to set the maximum number of trains, **mt**, that can be at a given station (Figure 6.7). The preconditions are that **mt** must be a natural number and cannot exceed the maximum number of platforms and that the maximum number of trains for a station cannot be less than the number of trains already there.

The point behind this is that the mathematical specification is far more precise than the natural language "translation" of it. In addition, the mathematical representation of the behavior is less verbose than the natural language version. Finally, the representation in B enables correctness proofs, via standard predicate calculus. And as stated before, the B specification can actually be converted to C++ or Ada code. For example, the safety-critical portion of New York City Transit's "L" (subway) line, whose Communication-Based Train-Control (CBTC) system has been in revenue service for several years, was specified in B (over a decade ago). The L-Line's B-based specifications were autotranslated into more than 25,000 proof obligations and (their) proofs, over 95% of which were proved by machine, with the

remaining 5% proven using semi-automatic (human-guided) proofs using a proof engine. The proven B specifications were then translated automatically into Ada code (Hacken 2007).

Formalization of Space Shuttle Flight Software Using MurΦ

The next example of formalization uses the Prototype Verification System (PVS) language to model and prove various aspects of the space shuttle flight software system (Crow and Di Vita 1998). PVS is a state-driven language, meaning it implements a finite-state machine to drive behavior.

A finite-state machine model M consists of a set of allowable states S, a set of inputs I, a set of outputs O, and a state transition function described by Equation (6.1):

$$M : I \times S \rightarrow [O \times S] \tag{6.1}$$

The initial and terminal states need to be defined as well.

Part of the implementation code in PVS is shown in Figure 6.8. Without explicating the code, it is interesting to point out its expressiveness in modeling the finite-state machine. Here the **principal_function** defines an interface incorporating the inputs (**pf_inputs**) and states (**pf_state**) needed to drive the behavior. The output is an output expression and the next state in the machine.

The behavior specification continues as an infinite sequence of state transitions (functional transformations; Figure 6.9, top part) and as a set of functions associated with each state, in this case, on the state associated with firing small control jet rockets (Vernier rockets). Even without fully understanding the syntax of PVS, because it is similar to C, C++, or Java, we hope you can see how this behavior of the system can be clearly described.

In addition to clarity and economy, another advantage of using PVS to describe the behavior is that an interactive proofchecker and Stanford University's MurΦ (pronounced "Murphy") can be used for theorem proving (specification verification).

```
pf_results: TYPE = [# output: pf_outputs, state: pf_state #]
principal_function (pf_inputs, pf_state,
                    pf_I_loads, pf_K_loads,
                    pf_constants) : pf_result =
        (# output : = <output expression>,
        state : = <next-state expression> #)
```

Figure 6.8 Initialization code in PVS for space shuttle functionality. (Adapted from J. Crow and B. Di Vita, *ACM Transactions on Software Engineering and Methodology*, 7(3), 1998.)

```
finite_sequences [n: posnat, t: TYPE] : THEORY
BEGIN
        finite_seq: TYPE = [below[n] -> t]
END     finite_sequences
requirements: THEORY
BEGIN
IMPORTING finite_sequences
...
        jet: TYPE
        rot_jet: TYPE FROM jet

        jet_bound: posint
        jet_count: [rot_jet -> below[jet_bound]]
        jet_count_ax: AXIOM injective? (jet_count)

        finite_number_of_jets: LEMMA...

        vernier?: pred[rot_jet]
        vernier_jet: TYPE = (vernier?)
        there_is_a_varnier: AXIOM (EXISTS j: varnier?(j)

        primary_rot? (j): bool = NOT vernier?(j)
        primary_rot_jet: TYPE = (primary_rot?)
        there_is_a_primary: AXIOM
                (EXISTS j: primary_rot? (j))

        downfiring?: pred[vernier_jet]
...
END requirements
```

Figure 6.9 Behavior of Vernier rockets in space shuttle system. (Adapted from J. Crow and B. Di Vita, *ACM Transactions on Software Engineering and Methodology*, 7(3), 1998.)

Formalization of an Energy Management System Using Category Theory*

Category theory (CT) permits functional notation to be used to model abstract objects and is therefore useful in formal specification of those objects (Awodey 2005). This formalization facilitates good design and can assist in user interface design and permits users of the system to build on their domain knowledge as they learn the constraints that affect design decisions. These design decisions then become evident as the rules are applied from requirements to design to user interface functionality. Finally, for ease of maintenance and troubleshooting for the designers and programmers, the use of CT can show where an error might be contained, while allowing for the group of functions to be analyzed and troubleshooting on an input-by-input basis. The following is a brief introduction to category theory for software specification.

* This section is adapted from *Formalization of an EMS System Using Category Theory*, master's thesis, Ann Richards under the supervision of P. Laplante (Penn State University, 2006).

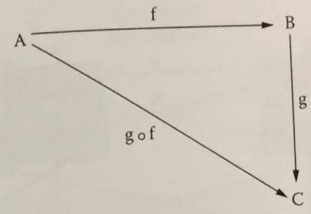

Figure 6.10 A simple functional relationship.

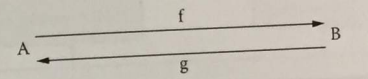

Figure 6.11 An infinite category.

Suppose we have classes of abstract objects, A, B, and C. Furthermore, suppose there are functions f and g such that $f: A \rightarrow B$ and $g: B \rightarrow C$. A category is composed of the objects of the category and the morphisms of the category. A basic functional relationship can be seen in Figure 6.10 where $g \circ f$ is the composition of f and g.

CT is widely used to model structures and their transformations over time (represented by finite or infinite sequences of functional compositions). For example, in Figure 6.11, the sequence of composed relationships between f and g would be given by f, fg, fgf, and so on.

These sequences also represent categories. For example, we just saw the space shuttle behavioral specification represented as an infinite sequence of state transitions using the PVS notation.

The composition of a system is also appropriate to model the modularity of a larger system and its interconnections. In comparison to a truth table, which is a great formalizing and testing tool, CT can associate related items and then decompose each into something that can be examined on a lower level. For example, business logic can be placed in a category either intuitively or architecturally.

Example: Energy Management System

The example we describe is a partial formalization of a power company's energy management system. Any power generation utility has to deal with a complex, real-time system involving a high level of automation, fault tolerance, and redundancy.

In an energy management system (EMS) an open access gateway (OAG) serves as a communication liaison for many communication protocols between computer systems and remote terminal units (RTU). The OAG is also used to communicate

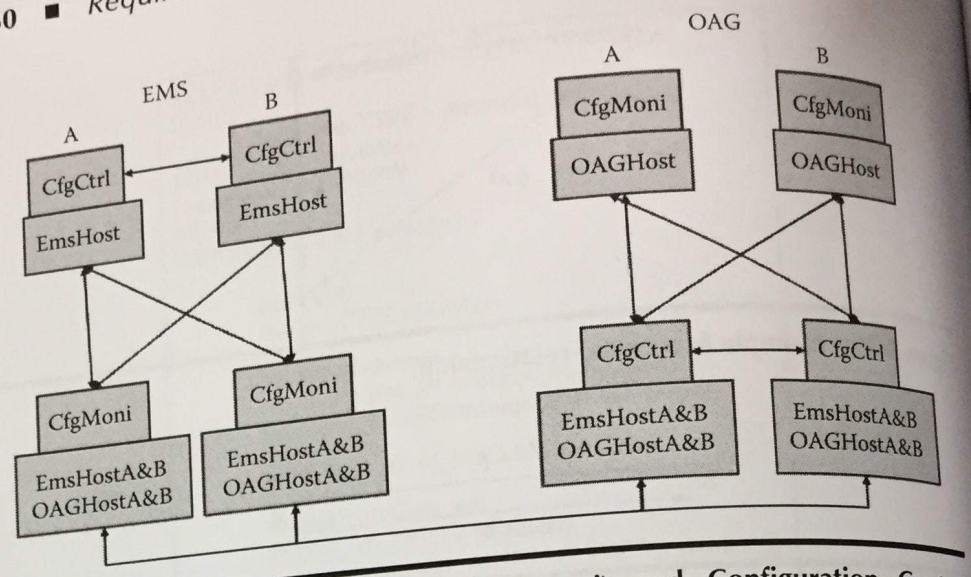

Figure 6.12 Configuration Monitor (CfgMoni) and Configuration Control (CfgCtrl) EMS and OAG relationship.

A1 – EMSA CFGCTRL	A2 – OAGA CFGCTRL
B1 – EMSB CFGCTRL	B2 – OAGB CFGCTRL
C1 – EMSA CFGMONI	C2 – OAGA CFGMONI
D1 – EMSB CFGMONI	D2 – OAGB CFGMONI
f1 – EMS CFGCTRL	f2 – OAG CFGCTRL
g1 – EMS CFMONI	g2 – OAGEMS CFMONI
j(g) – EMS CFGMONI to OAG CFGCTRL	h(g) – OAG CFGCTRL to EMS CFGMONI

Figure 6.13 Data dictionary and category reference.

with the plant digital interface (PDI) via intercompany communications protocol (ICCP) and to other entities such as the interconnection and the TMS as shown in Figure 6.12.

To formalize Figure 6.12 using category theory, we first translate the entities using shorthand symbols as shown in Figure 6.13.

The resultant relabeled system is shown in Figure 6.14. Functional notations have been added to Figure 6.14, but these would need to be defined further.

Let's discuss a small portion of the formalization in Figure 6.14. We begin with EMS A and EMS B. If A1 = EMS A and B1 = EMS B, then suppose $f1$ is a bidirectional mapping between EMS A and EMS B that describes communications between CfgCtrl and CfgMoni. Taking one side of the mapping, suppose we have $f1$: $A1 \rightarrow B1$. Next, we assert that $f1$ is defined on all $A1$ and all the values of $f1 \in B1$. That is, $range(f1) \subseteq B1$.

Continuing, suppose there is another bidirectional mapping, $g1$ between EMS A and EMS B. Defining one side of $g1 : A1 \rightarrow D1$ we have an associated composed function $g1 \circ f1 : A1 \rightarrow C1 \cap D1$; that is, $(g1 \circ f1)(a1) = g1(f1(a1))$ and $a1 \in A1$.

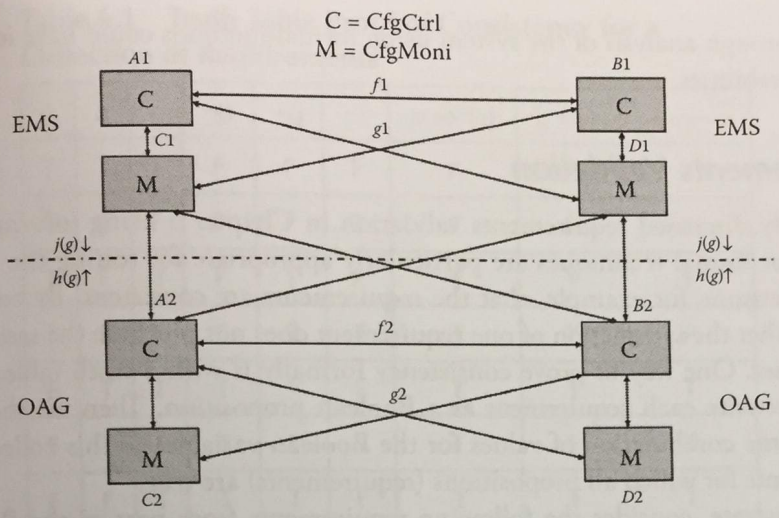

Figure 6.14 Partial formalization of Figure 6.12 using relabeled components.

The composed function is associative, as follows. If there is another function j : $C1 \rightarrow A2$ and forming $j \circ g1$ and $g1 \circ f1$ then we can compare $j \circ g1 \circ f1$ and $j \circ (g1 \circ f1)$. As indicated in Figure 6.14 it becomes apparent that these two functions are always identical.

At this point, it should be noted that the set of all inputs and outputs from $A1 \cap B1 \subset C1 \cup D1$ where $C1$ and $D1$ is CfgMoni on both the EMS A and B servers. The set of inputs and outputs from $C1 \cap D1 \subset A1 \cup B1$ where $A1$ and $B1$ is the CfgCtrl on the EMS A and B servers. CfgCtrl communicates with itself on both the EMS and OAG servers. The function $j(g) : C1 \rightarrow A2 \cup B2$ shows the configuration from the EMS CfgMoni to OAG CfgCtrl. OAG also communicates with CfgCtrl on its home server; that is, there is a function $g1(g) : C1 \rightarrow A1$. This feature is needed because CfgCtrl communicates with each redundant server so that controls can be issued while CfgMoni sends and receives status to CfgCtrl for up-to-date status on the server that it is monitoring.

To relate this to the EMS and OAG configurations, the set A1 is EMS CfgCtrl A, the set $B1$ is EMS CfgCtrl B, and $j(g) : (C1 \cup D1) \rightarrow (A2 \cup B2)$ shows that CfgMoni from either EMS A or B maps to CfgCtrl on the OAG A or B. We could continue with this formalization, but you should realize by now that a high level of attention to detail and rigor is necessary to create this kind of specification, potentially helping to avoid important interface definition and other problems.

For example, security vulnerability is of great concern in power management systems, a fact that has been illustrated in cyberattacks on certain utility company computer systems, causing power outages. The attacks were based on vulnerabilities in SCADA (supervisory control and data acquisition) systems, on which many energy management systems run (including the one described here). It is possible

that a thorough analysis of the system using formal methods could have identified the vulnerabilities.

Requirements Validation

We already discussed requirements validation in Chapter 5 using informal techniques, but formal techniques are particularly appropriate for testing the requirements to ensure, for example, that the requirements are consistent. By consistent we mean that the satisfaction of one requirement does not preclude the satisfaction of any other. One way to prove consistency formally is with a truth table. In this case, we rewrite each requirement as a Boolean proposition. Then we show that there is some combination of values for the Boolean variables of this collection of requirements for which all propositions (requirements) are true.

To illustrate, consider the following requirements from part of the Pet Store POS system:

1.1 If the system software is in debug mode, then the users are not permitted to access the database.
1.2 If the users have database access, then they can save new records.
1.3 If the users cannot save new records, then the system is not in debug mode.

We convert the requirements into a set of Boolean propositions. Let, p, q, and r be Boolean variables and let

p: be the statement "the system software is in debug mode"
q: be the statement "the users can access the database"
r: be the statement "the users can save new records"

Clearly, the specifications are equivalent to the following propositions:

1.1 $p \Rightarrow \neg q$
1.2 $q \Rightarrow r$
1.3 $\neg r \Rightarrow \neg p$

Now we construct a truth table (Table 6.1) and determine if any one of these rows has all "T"s in the columns corresponding to Propositions 1.1, 1.2, and 1.3, meaning each of the requirements is satisfied. If we can find such a row, then the requirements are consistent.

We see that there are four rows where all three propositions representing the requirements are true for all combinations of Boolean values. Therefore, this set of requirements is consistent.

Notice that Requirement 1.3 likely does not make logical sense. If the users cannot save new records then it is likely it was intended to say that the

Table 6.1 Truth Table Proof of Consistency for a Collection of Requirements

p	q	r	$\neg p$	$\neg q$	$\neg r$	$p \Rightarrow \neg q$	$q \Rightarrow r$	$\neg r \Rightarrow \neg p$
T	T	T	F	F	F	F	T	T
T	T	F	F	F	T	F	F	F
T	F	T	F	T	F	**T**	**T**	**T**
T	F	F	F	T	T	T	T	F
F	T	T	T	F	F	**T**	**T**	**T**
F	T	F	T	F	T	T	F	T
F	F	T	T	T	F	**T**	**T**	**T**
F	F	F	T	T	T	**T**	**T**	**T**

system is in debug mode (as opposed to not in debug mode). We deliberately left this logically questionable requirement in place to make a point: this system of requirements is consistent but not necessarily correct (in terms of what the customer intended). This consistency checking process will not uncover problems of intent (validity). These must be uncovered through other means such as requirements reviews.

Consistency checking using Boolean satisfiability is a powerful tool. However, although the process can be automated, the problem space grows large very quickly. For n logical variables in the requirements set, the problem is $O(2^n)$. This kind of problem quickly becomes intractable even for supercomputers: it is a well-known NP-complete problem, the Boolean satisfiability problem. There are some special cases in which the Boolean satisfiability problem can be structured so that it can be solved efficiently using computer programs. Even so, we would likely use this technique for requirements verification only for very critical logic situations. For example, the launch/no-launch decision logic for a weapon, dosage administration logic for some kind of medical device, shut-down logic for a nuclear power plant, and so on.

Theorem Proving

Theorem-proving techniques can be used to demonstrate that specifications are correct. That is, axioms of system behavior can be used to derive a proof that a system (or program) will behave in a given way. Remember, a specification and program are both the same thing: a model of execution. Therefore program-proving techniques can be used for appropriately formalized specifications. These techniques, however, require mathematical discipline.

Program Correctness

A system is correct if it produces the correct output for every possible input and if it terminates. Therefore, system verification consists of two steps:

- Show that, for every input, the correct output is produced (this is called partial correctness).
- Show that the system always terminates.

To deal with this we need to define assertions. An assertion is some relation that is true at that instant in the execution of the program. An assertion preceding a statement in a program is called a precondition of the statement. An assertion following a statement is called a postcondition. Certain programming languages, such as Eiffel and Java, incorporate runtime assertion checking. Assertions are also used for testing fault-tolerance (through fault-injection).

Hoare Logic

Much of the following is based on the presentation found in Gries and Gries (2005). Suppose we view a requirements specification as a sequence of logical statements specifying system behavior. In 1969 C.A.R. (Tony) Hoare introduced the notation (called the Hoare triple):

$$P\,\{S\}\,Q$$

where P and Q are assertions (pre- and postconditions, respectively) and S is a system behavioral segment. Then, the precondition, statement, postcondition triple has the following meaning. Execution* of the statement begun with the precondition true is guaranteed to terminate, and when it terminates, the postcondition will be true.

The intent is that P is true before the execution of S; then after S executes, Q is true. Note that Q is false if S does not terminate.

Hoare's logic system can be used to show that the system segment (specification) is correct under the given assertions. For example, suppose we have the following:

```
//{x = 5}                    an assertion (precondition)
z = x + 2;
//{z = 7}                    an assertion (postcondition)
```

* "Execution" means either program execution or effective change in system behavior in the case of a nonsoftware behavioral segment.

Proof: Suppose that {x = 5} is true, then z = 5 + 2 = 7 after the system begins execution. So the system is partially correct. That the system terminates is obvious, as there are no further statements to be executed. Hence, correctness is proved.

Hoare added an inference rule for two rules in sequence. We write this as:

$$\frac{P\,\{S_1\}\,Q \qquad Q\,\{S_2\}\,R}{P\,\{S_1;\,S_2\}\,R}$$

This inference rule means that if the postcondition (Q) of the first system segment (S_1) is the antecedent (precondition) of the next segment, then the postcondition of the latter segment (R) is the consequent of the concatenated segments. The horizontal line means "therefore" or, literally, "as a result."

To illustrate, we show that the following specification is correct under the given assertions:

```
//{x = 1 ∧ y = 2}            an assertion (precondition)
x = x + 1;
z = x + y;
//{z = 4}                    an assertion (postcondition)
```

Proof: Suppose that {x = 1} is true, then the system executes the first instruction, x = 2. Next, suppose that {y = 2} is true. Then after the execution of the second statement z = 2 + 2 = 4. Hence z = 4 after the execution of the last statement and the final assertion is true, so the system is partially correct. That the system terminates is obvious, as there are no further statements to be executed.

An inference rule is needed to handle conditionals. This rule shows that

$$\frac{(P \wedge \text{condition})\{S\}Q \qquad (P \wedge \neg\,\text{condition}) \Rightarrow Q}{\therefore\ P\,\{\text{if condition then } S\}Q}$$

To illustrate, we show that the specification segment is correct under the given assertions:

```
//{Any}                    an assertion (precondition) that is always true
if x > y then
y = x;
//{x ≤ y}                  an assertion (postcondition)
```

Proof: If, upon entry to the segment, $y \leq x$, then the if statement fails and no instructions are executed and, clearly, the final assertion must be true. On the other hand, if upon entry, $x > y$, then the statement $y = x$ is executed. Subsequently $y = x$ and the final assertion $x \leq y$ must be true. That the system terminates is obvious, as there are no further statements to be executed.

Another inference rule handles if-then-else situations; that is,

$$(P \wedge \text{condition})\{S_1\}Q$$

$$(P \wedge \neg \text{condition})\{S_2\}Q$$

$$\text{———}$$

$$\therefore P \{\text{if condition then } S_1 \text{ else } S_2\}Q$$

To illustrate, we show that the specification segment is correct under the given assertions:

```
//{Any}                    an assertion (precondition) that is always true
if x < 0 then
abs = −x;
else
abs = x;
//{abs = |x|}              an assertion (postcondition)
```


Proof: If $x < 0$, then $abs = -x \Rightarrow$ abs is assigned the positive x. If $x \geq 0$ then abs is also assigned the positive value of x. Therefore $abs = |x|$. That the system terminates is obvious, as there are no further statements to be executed.

Finally, we need a rule to handle "while" statements:

$$(P \wedge \text{condition})\{S\}P$$

$$\text{———}$$

$$\therefore (P \wedge \text{while condition})\{S\}(\neg\text{condition} \wedge P)$$

We illustrate this rule through example shortly.

So far the specification snippets (one might say they are really "code" snippets) that have been proven represent very simplistic but low-level, detailed behavior unlikely to be found in a requirements specification. Or is this the case? It is not so hard to imagine that for critical behavior, it might be necessary to provide specifications at this level of detail. For example, the behavior required for the dosing logic for an insulin pump (or machine that delivers controlled radiation therapy) might require such logic. The decision to launch a missile, control the life support system in the Space Station, or shut down a nuclear plant might be based on simple, but critical logic. This logic, incidentally, could have been implemented entirely in hardware.

Even our running examples might have some critical logic that needs formal proof. For example, the baggage counting logic for the baggage handling system or certain inventory control logic might require the following behavior:

```
//{sum = 0 ∧ count = n ≥ 0}
while count > 0
  {
  sum = sum + count;
  count = count— ;
  }
//{sum = n(n + 1)/2}
```

And so it would be necessary to show that the specification segment is correct under the given assertions. To prove it, we use induction.

Basis:

Suppose that *sum* = 0 and *n* = 0 as given by the assertion. Now upon testing of the loop guard, the value is false and the system terminates. At this point the value of *sum* is zero, which satisfies the postcondition. That the system terminates was just shown.

Induction Hypothesis:

The program is correct for the value of *count* = *n*. That is, it produces a value of *sum* = *n*(*n* + 1)/2 and the system terminates.

Induction Step:

Suppose that the value of *sum* = 0 and *count* = *n* + 1 in the precondition.

Upon entry into the loop, we assign the value of sum = *n* + 1, and then set *count* = *n*. Now, from the induction hypothesis, we know that for *count* = *n*, sum = *n*(*n* + 1)/2. Therefore, upon continued execution of the system, sum will result in (*n* + 1) + *n*(*n* + 1)/2, which is just (*n* + 1)(*n* + 1 + 1)/2 and the system is partially correct.

According to the induction hypothesis, by the *n*th iteration of the loop *count* has been decremented *n* times (because the loop exited when *count* was initialized to *n*). In the induction step, *count* was initialized to *n* + 1, so by the nth iteration it has a value or 1. So after the *n* + 1st iteration it will be decremented to 0, and hence the system exits from the loop and terminates.

Does this approach really prove correctness? It does, but if you don't believe it, try using Hoare logic to prove an incorrect specification segment to be correct, for example:

$$//\{sum = 0 \wedge count = n \geq 0\}$$

while count < *n*

{

sum = *sum* + *count*;

count = *count*++;

}

$$//\{sum = n(n + 1)/2\}$$

This specification is incorrect because *n* is left out of *sum* and, therefore, no proof of correctness can be obtained.

It is easy to show that a *for* loop uses a similar proof technique to the *while* loop. In fact, by Böhm and Jacopini (1966), we know we can construct verification proofs just from the first two inference rules. For recursive procedures, we use a similar, inductive, proof technique applying the induction to the *n* + 1st iteration in the induction step, and using strong induction if necessary.

Model Checking

Given a formal specification, a model checker can automatically verify that certain properties are theorems of the specification. Model checking has traditionally been used in checking hardware designs (e.g., through the use of logic diagrams), but it has also been used with software. Model checking's usefulness in checking software specifications has always been problematic due to the combinatorial state explosion and because variables can take on non-Boolean values. Nevertheless, Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis won the 2007 A.M. Turing Award for their body of work making model checking more practical, particularly in the design of chips. A great deal of interesting and important research continues in model checking and practical implementations exist.

Integrated Tools

There are several integrated toolsets that combine formal specification tools, with model checkers, consistency checkers, and code generators.

For example, consider a requirements-focused formal method called Software Cost Reduction (SCR), which was first introduced in the 1970s. SCR uses a tabular notation (inherently formal) to represent the system behavior in a way that is understandable to stakeholders. Then a set of formal tools can be used to check the consistency, completeness, and correctness of the specification. In addition, the toolset includes a model checker, a front end to PVS, an invariant generator, a property checker, and a test case generator.

Many companies including Lockheed Martin have been using SCR for many years. Notable successes include certifying that a security-critical module of an embedded software device enforced data separation and specifying the requirements of three safety critical software modules for NASA systems (Heitmeyer 2007).

Objections, Myths, and Limitations

There are a number of "standard" objections to formal methods, for example, that they are too hard to use or expensive, hard to socialize, or that they require too much training. The rebuttals to these objections are straightforward, but situational. For example, how much is too much training? Certain organizations spend

millions of dollars to attain certain capability maturity model (CMM) levels, but balk at investing significantly less money in training for the use of formal methods. In any case, be wary of out-of-hand dismissal of formal methods based on naïve points. But if you needed an implantable insulin pump or defibrillator what kind of requirements verification and validation would you want? You probably would want every kind of verification and validation available including formal techniques. The same goes for a nuclear plant near your home, a radiation dosing machine that you needed, the safety and restraint system for your automobile, and so on. Even in relatively mundane home devices, such as a smart clothes washer, high fidelity requirements are required, and these can be achieved only with a combination of formal and informal techniques. Formal methods are not for every situation, but they really ought to be considered in mission-critical situations.

Objections and Myths

Some objections to using formal methods include the fact that they can be error-prone (just as mathematical theorem proving or computer programming are) and that sometimes they are unrealistic for some systems. These objections are valid, sometimes. But formal methods are not intended to be used in isolation, nor do they take the place of testing. Formal methods are complementary to other quality assurance approaches. Some of the other "standard" objections to formal methods are based on myth. Papers by Hall, and Bowen and Hinchey help capture and rebut these misconceptions. The first set of myths is as follows (Hall 1990):

1. *Myth: Formal methods can guarantee that software is perfect.* Truth: Nothing can guarantee that software will be perfect. Formal methods are one of many techniques that improve software qualities, particularly reliability.
2. *Myth: Formal methods are all about program proving.* Truth: We have shown that formal methods involve more than just program proving and involve expressing requirements with precision and economy, requirements validation, and model checking.
3. *Myth: Formal methods are only useful for safety-critical systems.* Truth: Formal methods are useful anywhere that high-quality software is desired.
4. *Myth: Formal methods require highly trained mathematicians.* Truth: While mathematical training is very helpful in mastering the nuances of expression, using formal methods is really about learning one or another formal language and about learning how to use language precisely. Requirements engineering must be based on the precise use of language, formal or otherwise.
5. *Myth: Formal methods increase the cost of development.* Truth: While there are costs associated with implementing a formal methods program, there are associated benefits. Those benefits include a reduction in necessary rework downstream in the software lifecycle—at a more expensive stage in the project to make corrections.

6. *Myth: Formal methods are unacceptable to users.* Truth: Users will accept formal methods if they understand the benefits that they impart.
7. *Myth: Formal methods are not used on real, large-scale systems.* Truth: Formal methods are used in very large systems, including high-profile projects at NASA.

Five years after Hall's myths, Jon Bowen and Mike Hinchey (1995) gave us "Seven More Myths of Formal Methods":

1. *Myth: Formal methods delay the development process.* Truth: If properly incorporated into the software development life cycle, the use of formal methods will not delay, but actually accelerate the development process.
2. *Myth: Formal methods lack tools.* Truth: We have already seen that there are a number of tools that support formal methods—editing tools, translation tools, compilers, model checkers, and so forth. The available tool set depends on the formal method used.
3. *Myth: Formal methods replace traditional engineering design methods.* Truth: Formal methods enhance and enrich traditional design.
4. *Myth: Formal methods only apply to software.* Truth: Many formal methods were developed for hardware systems, for example, the use of Petri Nets in digital design.
5. *Myth: Formal methods are unnecessary.* Truth: This statement is true only if high-integrity systems are unnecessary.
6. *Myth: Formal methods are not supported.* Truth: Many organizations use, evolve, and promote formal methods. Furthermore, see Myth 2.
7. *Myth: Formal methods people always use formal methods.* Truth: "Formal methods people" use whatever tools are needed. Often that means using formal methods, but it also means using informal and "traditional" techniques too.

Parnas (2010) raised another compelling objection to traditional formal methods noting that

> Funding agencies often require that larger research-funded projects include some cooperation with industrial organizations and demonstrate the practicality of an approach on "real" examples. When authors report such efforts, they state that they are successful. Paradoxically, such success stories reveal the failure of industry to adopt formal methods as standard procedures; if using these methods was routine, papers describing successful use would not be published.

It is difficult to dispute his analysis; the use of formal methods in real industrial-strength systems is still relatively rare. Parnas's suggested antidote, however, is to invent a new set of formal methods that are more likely to be adapted by mainstream engineers. To date no such antidote has been found.

Limitations of Formal Methods

No method can guarantee absolute correctness, safety, and so on, and formal methods have some limitations. For example, they do not yet offer good ways to reason about alternative designs or architectures. Formal specifications must be converted to a design, then a conventional implementation language. This translation process is subject to all the potential pitfalls of any programming effort.

For example, consider the following specification segment for an accelerometer processing function in an avionics system. This operation is to be initiated by an interrupt to read the accumulated acceleration pulses collected every 10 milliseconds. From these, velocity and position (with respect to one axis) can be determined.

3.2.2 Compute accumulated acceleration every 10 milliseconds as follows:

Precondition: $x = x_{prev}$

$x \Leftarrow x + \Delta x;$

Postcondition: $x = x_{prev} + \Delta x$

Let's use a modified Hoare logic approach to show that the specification is correct under the given assertions. Suppose that $x = x_{prev}$ is true, then after the addition and assignment operations are performed, $x = x_{prev} + \Delta x$. So the specification fragment is correct. There is no need to show "termination" inasmuch as there are no more operational requirements in this fragment. So this specification fragment is verified.

But what if x is a 16-bit number and Δx is 64-bit number? Then there is the possibility of an overflow condition in x. These are design details and errors therein cannot be identified through verification or validation of the requirements specification.

Jean-Raymond Abrial, the father of both Z and B and the intellectual leader behind the formal verification of the Paris Meteor railway line previously described, suggested the distinction between requirements and design verification in the context of the crash of the European Union's Ariane 5 in 1996 (Abrial 2009). In this case an unmanned rocket crashed 30 seconds into flight due to a loss of guidance and attitude information. The lost information was caused by an incorrectly handled data conversion from a 64-bit floating point to 16-bit signed integer value, a problem that was not caught in requirements verification but was more aptly due to an implementation error or testing inadequacy (Nuseibeh 1997).

In other words a formal proof such as the one just demonstrated would not likely have prevented the Ariane 5 disaster. But we shouldn't conclude from the Ariane 5 case that formal methods are inadequate. We can conclude, however, that formal methods used for verification or validation in one phase of systems engineering do not eliminate the need for other forms of verification and validation such as system testing.

Bowen and Hinchey's Advice

Some final advice on the use of formal methods is adapted from "Ten Commandments of Formal Methods" by Bowen and Hinchey (1995b).

1. **Thou shalt choose the appropriate notation.**
   Comment: Whether you use B, Z, VDM, PVS, or whatever, pick the formal notation that is most appropriate for the situation. We do not advocate one formal language or another. What is appropriate for a situation will depend largely on what the organization is most comfortable using and the supporting tools available. An organization is not typically fluent in multiple formal languages, however, and it is not recommended that a company change notation from one situation to the next. Training in the use of a formal language has to be considered if there are not enough engineers capable of using the tool of choice.
2. **Thou shalt formalize, but not overformalize.**
   Comment: Everything in moderation (including moderation).
3. **Thou shalt estimate costs.**
   Comment: Modern project management practice requires cost estimation at every step of the way.
4. **Thou shalt have a formal method guru on call.**
   Comment: It is always recommended to have access to an "expert" in formal methods. That expert may be an in-house specialist or a consultant.
5. **Thou shalt not abandon thy traditional development methods.**
   Comment: It has been continuously noted that formal methods are augmentative in nature.
6. **Thou shalt document sufficiently.**
   Comment: Documentation is critical in all systems engineering activities.
7. **Thou shalt not compromise thy quality standards.**
   Comment: Any objection here?
8. **Thou shalt not be dogmatic.**
   Comment: Dogmatism is not the way to promote the benefits of formal methods. We have tried to be balanced in our approach to formal methods usage.
9. **Thou shalt test, test, and test again.**
   Comment: Formal methods do not replace testing; they are supplemental to testing. An appropriate life-cycle testing program is essential to modern systems engineering.

10. Thou shalt reuse.
Comment: Take advantage of the power of formal methods, and one of the greatest of these is confidence in reuse. A software module that has been formally validated (as well as tested in the traditional sense) will be a good candidate for reuse because it is trusted.

Exercises

6.1 Are customers more likely to feel confident if formal methods are explained to them and then used?

6.2 Where in the software development process life cycle do formal methods provide the most benefit?

6.3 Rewrite the train station specification in another formal language, such as Z or VDM.

6.4 Conduct a consistency check for the requirements found in Section 8.2 of the smart home SRS (video entry).

6.5 Conduct a consistency check for the requirements found in Section 8.3 of the smart home SRS (video playback).

6.6 Consider the following set of requirements for an insulin pump system:
- 3.1 If the morphine dose button is pressed then the morphine dose is administered.
- 3.2 If the morphine dose is administered then the dose indicator light is off.
- 3.3 If the morphine dose button is pressed and the dose indicator light is on then the morphine dose is administered.
- 3.4 If the morphine dose is administered then the dose indicator light is on.

Determine using a truth table if these requirements are consistent or not.

6.7 Which requirements (groups) in the requirements in the SRS of Appendix A would benefit from formalization? (Be specific as to formalization type.)

*6.8 For the formalizations in Sections 3.2.1 and 3.2.5 in the SRS of Appendix B discuss the problems that exist. Using whatever assumptions you need, rewrite these formalizations using Z, B, or VDM or any other formal method choose.

References

Abrial, J.R. (2009). Faultless systems: Yes we can! *Computer*, 42(9): 30–36.

Awodey, S. (2005). *Category Theory*, Pittsburgh, PA: Carnegie Mellon University.

Böhm, C. and Jacopini, G. (1966). Flow diagrams, Turing machines, and languages with only two formation rules, *Communications of the ACM*, 9(5): 366–371.

* This exercise is suitable for a small research assignment.

Bowen, J.P. and Hinchey, M.G. (1995a). Seven more myths of formal methods, *Software*, 12(4): 34–41.

Bowen, J.P. and Hinchey, M.G. (1995b). Ten commandments of formal methods, *Computer*, 28(4): 56–63.

Bowen, J.P., Hinchey, M., and Vassev, E. (2010). Formal requirements specification. In *Encyclopedia of Software Engineering*, P. Laplante (Ed.), Boca Raton, FL: Taylor & Francis. Published online: 10 Nov 2010; 321–332.

Clarke, E.M. and Wing, J.M. (1996). Formal methods: State of the art and future directions, *ACM Computing Surveys*, 28(4): 626–643.

Crow, J. and Di Vita, B. (1998). Formalizing Space Shuttle software requirements: Four case studies, *ACM Transactions on Software Engineering and Methodology*, 7(3).

Forster, T.E. (2003). *Logic, Induction and Sets*. Cambridge, UK: Cambridge University Press.

Gries, D. and Gries, P. (2005) *Multimedia Introduction to Programming Using Java*. New York: Springer.

Hacken, G. (2007). Private e-mail communication to P. Laplante.

Hall, A. (1990). Seven myths of formal methods, *Software*, 7(5): 11–19.

Heitmeyer, C. (2007). Formal methods for specifying, validating, and verifying requirements, *Journal of Universal Computer Science*, 13(5): 607–618.

Hinchey, M.G. (1993). Formal methods for system specification, *Potentials*, 12(3): 50–52.

Hoare, C.A.R. (1969). An axiomatic basis for computer programming, *Communications of the ACM*, 12(10): 576–583.

IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology, IEEE Standards, Piscataway, NJ.

Lano, K. (1996). *The B Language and Method*. New York: Springer-Verlag, pp. 94–98.

NASA (2012). National Space Science Data Center: Mariner 1, http://nssdc.gsfc.nasa.gov/nmc/masterCatalog.do?sc=MARIN1, last accessed, April 2013.

Nuseibeh, B. (1997). Ariane 5: Who dunnit? *Software*, 14(3): 15–16.

Parnas, D.L. (2010). Really rethinking "formal methods," *Computer*, 43(1): 28–34.

Shehata, M. and Eberlein, A. (2010). Requirements interaction detection In *Encyclopedia of Software Engineering*. P. Laplante (Ed.), Boca Raton, FL: Taylor & Francis. Published online: 979–998.

Truss, L. (2004). *Eats Shoots and Leaves: The Zero Tolerance Approach to Punctuation*, London: Profile Books.