UNIVERSITÉ YORK UNIVERSITY

The Logic of Software Design

Jonathan Ostroff

Richard Paige

Technical Report CS-98-04

July 6, 1998

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

The Logic of Software Design

Jonathan S. Ostroff and Richard Paige[1]
Department Of Computer Science, York University,
4700 Keele Street, Toronto, Ontario, Canada, M3J 1P3.
Email: {jonathan, paige}@cs.yorku.ca
Tel: 416-736-2100 x{77882,77878}   Fax: 416-736-5872.

Abstract: In recent years much progress has been made towards the development of mathematical methods ("formal methods") through which it is possible, in principal, to specify and design software to conform to specifications. Although formal methods have the potential to offer a basis for software engineering akin to the calculational methods and tools of other engineering disciplines, these methods have had only a limited effect on industrial practice. One reason (amongst many) for this state of affairs is that the software engineering curriculum needs to incorporate these methods and tools if the next generation of programming professionals are to use them. In this paper, we provide an overview of how formal methods can be used throughout the software development cycle, and what methods and tools can be introduced in the computer science curriculum to support software development.

Keywords: Software engineering education, formal methods, logic **E**, logical calculation, theorem provers, Eiffel.

1. This research was supported with the help of NSERC (National Science and Engineering Research Council of Canada).

Table of Contents

List of Figures

1.0 Introduction

Logic is the glue that binds together reasoning in many domains, such as mathematics, philosophy, digital hardware, and artificial intelligence. In software development, logic has played an important role in the area of program verification, but its use has not on the whole been adopted in practice.

Some researchers and practitioners have suggested that logic (and mathematics in general) should play a more significant part in software development than it currently does [4,23]. They argue that software behaviour cannot be specified, predicted, or precisely documented without the use of mathematical methods. Engineers traditionally use mathematics to describe properties of products. Similarly, software engineers can use mathematics to describe properties of their products which are programs.

This argument is not generally accepted by the profession at large[2] for a variety of reasons. It is argued that the use of mathematical methods is expensive, unproven in large-scale development, and unsupported by usable tools. Many papers (e.g. [5,8,10]) have discussed the reasons for practitioners not adopting mathematical methods in full or in part. These arguments will not be recounted in full here, but it is clear that software professionals will not adopt mathematical methods until they are easy to use, improve our ability to deliver quality code on time, provide tool support, and are founded on an appropriate educational programme.

Electrical engineers are taught mathematical methods (e.g. differential equations or Laplace transforms) and tools (e.g. Matlab or Spice) for describing the properties of circuits. Such methods and tools are a key component of an electrical engineering education. Similarly, engineers use mathematical descriptions in discussions of the deformation of a beam, the flow of fluid in a pipe and the evolution of a chemical reaction. Methods, tools, and curriculum components of similar simplicity and ease of use are needed for the education and practice of software engineering.

In this paper, we provide an overview of how mathematical methods ("formal methods") can be used throughout the software development cycle, and what methods and tools can be introduced in the computer science curriculum to support software development. We provide some simple examples of methods and tools to motivate the material.

Organization of the paper

The paper commences with an overview of software engineering, its purpose, and its fundamental definitions. We describe a method for software design, using logic as the foundation. Logic is used for describing requirements, specifications, design and programs. We recap the calculational proof format, presented in [7], and thereafter apply it to simple examples that illustrate the method. We discuss some of the existing tools that can support calculational proof and the use of logic for software design. Throughout the paper, we discuss how logic can be integrated into a CS curriculum.

2. For some examples of where formal methods have been applied in the industrial setting, see Table 1 (end of Sect. 5.0).

2.0 Software Engineering

A software application such as a word processor is a *machine* — one similar to a typewriter, but with more versatility. Similarly, a software telephone switch is a machine — one similar to an old-fashioned telephone exchange, except that the new kind of machine does not consist of rotary switches and clattering relays.

The purpose of software development is to build special kinds of machines — those that can be physically embodied in a general purpose computer — merely by describing them as programs. A general purpose computer accepts our description of the particular machine we want (as described in the program), and converts itself into the desired machine. We summarize below some insights into software development as described by Jackson [12].

2.1 Requirements, Specifications and Programs

To construct a "software" machine, we must go through normal product development that engineers perform when constructing "hard" machines like typewriters, bridges, and motors. This includes requirements elicitation, analysis and design, implementation, testing and documentation.

The purpose of the machine is to be installed in the world and to interact with it. The part of the world in which the machine's effects will be felt — and which is of most interest to the customers of the machine — is called the *real-world domain*[3], which we will denote by the letter *W (*for world). So we have a machine *M* and the world that it interacts with *W*.

It is always right to pay serious attention to the real-world domain *W*. If we are developing a program to control an airplane, we obviously need to understand how the airplane works, how it lands and takes off on runways, and how it can be controlled while in the air. We may also need to understand intangibles associated with the real-world domain, such as the rules for safe aviation. This understanding must be made prior to any attempt to lay out the data structures and data flow of the software program that will ultimately control the airplane.

For example, the one million line program GPS (Global Positioning System for satellite navigation) involves an understanding of celestial mechanics, gravity, atomic clocks and cryptography. The phenomena of the real-world domain for the GPS are clearly distinct from the phenomena (code and data structure) of the machine required to operate it. Similarly, a telephone switch deals with telephone calls, a word-processing program deals with text, and a process control program deals with a chemical plant. These domains (tele-

3. Jackson [12] calls *W* the "application domain". We prefer to call *W* the *real-world* domain, not because the machine phenomena are not real, but because the phenomena of *W* predate the requirements. We can describe the phenomena of *W* and possibly influence them; but the designer does not create the phenomena of the real-world. By contrast, the machine is initially undetermined (i.e. not "yet" real), and it is the designer who creates or controls the machine phenomena. We do not use the term "application domain" because this can be confused with a generic domain denoting a class of applications (e.g. the process control domain). The word "environment" is also used for *W*, but this suggests something that physically surrounds the machine, whereas *W* can also include intangible things such as the rules for safe aviation or employment legislation.

phones, text and plants) are very different and each has its own peculiar characteristics that determine how it interacts with the machine.

It is the phenomena of the real-world domain that determine the customer's *requirements*[4]. This is what makes requirements capture an almost impossible task, because there is no way of rigorously checking that we actually understood what the customer wanted when we deliver the final machine. It is easy for a software developer to ignore the real-world domain (the realm of their customer's true requirements), for it is more enjoyable to turn directly to the machine where one can start implementing the "solution" immediately ("coding"). But, to focus on the machine too soon, may quickly lead to confusion and ambiguity. If we were never quite clear on what our customers really wanted then the final product is likely to disappoint them. This is also why programmers do not always thoroughly understand the properties of their products, nor do they apply accepted theory where it leads to better or safer products.

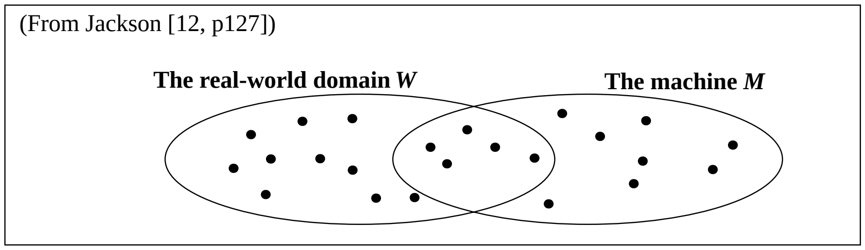
Requirements are therefore about the phenomena of the real-world domain *W* and not about the phenomena of the machine *M*. Not all the phenomena of the real-world domain are necessarily shared with the machine. But, the machine does share some phenomena with the real-world. The machine can try to ensure that the requirements are satisfied by manipulating the shared phenomena at the interface of *W* and *M*.

An example of a shared phenomenon is the event of a passenger sitting in an aircraft seat and pushing a button to turn on a light. The push of the button is a shared phenomenon between the passenger (who is part of *W*) and the aircraft (*M*). To the passenger the event is "push the button", and to the machine the event is "input signal on interrupt line L1". Similarly, the state in which the machine emits a continuous beep is the same state in which the user of the machine hears the continuous beep.

2.2 The gap between requirements and programs

As we mentioned, not all of the phenomena of the real-world domain are shared with the machine. There can thus be a gap between the customer's requirements and what the machine can deliver directly. We can think of the various phenomena with the help of Fig. 1, in which $W \cap M$ is the set of all shared phenomena.

FIGURE 1. The phenomena of the real-world domain *W* and the machine *M*



The requirements *R* are described in terms of *W*, so they may involve phenomena that are not shared with the machine. The program that will run on *M* will be written in terms

4. Certain select requirements may also refer to the phenomena of the machine, e.g. the machine code (the program) must be well-structured and efficient.

of the phenomena of M. The traditional progression from requirements to an implemented program is a way of bridging the gap between the phenomena of W and those of M.

A rational development process, where each step follows from the previous ones and everything is done in the most elegant and economic order, does not exist in reality for complex systems. Nevertheless, we can fake it [22]. We can try to follow an established procedure as closely as possible, and the final product and documentation is the ideal that would have resulted had we not departed from the established procedure. There are a number of advantages to faking it in this way, despite numerous departures from the ideal. The process will guide us even if we do not always follow it. We will come closer to rational development, and it will also be easier to measure progress

Rational software development:

1. Elicit and document the *requirements* R in terms of the phenomena of W.
2. From the requirements R, expressed in terms of W, we derive a *specification* $M.spec$ of the machine, expressed in terms of the shared phenomena $W \cap M$. Specifications thus describe the required interface or boundary between the machine and the application domain.
3. From the specification $M.spec$ we derive the program $M.prog$. The program refers to shared and internal phenomena of M.

We must now provide a justification that the program satisfies its requirement R. To justify this claim, we must reason as follows:

1. First, argue that if the machine behaves like $M.prog$, then the specification $M.spec$ is satisfied. i.e.,

   implementation correctness: $M.prog \rightarrow M.spec$. (Eq. 1)

   The implication states that $M.prog$ is a more specific or determinate product than the more abstract specification $M.spec$. This makes the program more useful and closer to implementation than the specification, for the program describes how the specification is implemented, whereas the specification describes what must be implemented, without any unnecessary appeal to internal detail. An example of a specification is $(x' = 0) \vee (x' = 1)$ where x' is the final value of the program variable x. The specification asserts that the final value of the program variable x must be either zero or one. An implementation of the specification is a program "$x{:=}1$", which can be described in logic by the assertion $(x' = 1)$. Since the predicate $(x' = 1) \rightarrow (x' = 0) \vee (x' = 1)$ is a theorem of propositional logic, it follows that the machine implementation satisfies its specification.

2. Next, argue that if the specification $M.spec$ is satisfied, then so is the requirement, i.e.

   specification correctness: $W.desc \wedge M.spec \rightarrow R$ (Eq. 2)

   where we may use our knowledge of the properties of the real-world domain ($W.desc$) to prove the implication.

3. Having shown implementation and specification correctness, we are then entitled to conclude that the machine correctly achieves the customer requirements, i.e.

$$\textbf{system correctness}: W.desc \wedge M.prog \to R. \qquad \text{(Eq. 3)}$$

In the development process described above, we made a distinction between *specifications* and *requirements*. The term "specification" is one of a trio of terms: requirements, specifications and programs.

Requirements are all about — and only about — the environment of the machine, i.e. the real-world phenomena. The customer is interested in these real-world phenomena — he wants the nuclear plant to run properly or the paychecks to be calculated correctly. Some of the customer's interests may coincidently involve shared phenomena at the specification interface $W \cap M$.

By contrast, programs are all about — and only about — the machine phenomena. Programmers will surely be interested in phenomena at the interface $W \cap M$, but this interest is motivated by the needs to obtain the data on which the machine must operate.

Specifications form a bridge between requirements and programs. Specifications are only about the shared phenomena $W \cap M$. Hence specifications are requirements of a kind (they are about some of the *W* phenomena) but they are also partly programs (they are about some of the *M* phenomena). Since specifications are derived from customer requirements by a number of reasoning steps, they may not make obvious sense to either the customer or the programmer. Although specifications are programs of a kind, they may not be executable. In fact, we would prefer that they not be tainted by implementation bias, i.e. with irrelevant machine detail.

The quality of the final software will depend critically on getting the description of the real-world domain $W.desc$ and the requirements R right. Jackson quotes a well-known incident in which a pilot landing his airplane had tried, correctly, to engage reverse thrust, but the system would not permit it, with the result that the pilot overshot the runway. The pilot could not engage reverse thrust because the runway was wet, and the wheels were aquaplaning instead of turning. But the control software only allowed reverse thrust to be engaged if pulses from the wheel sensors showed that the wheels were turning (which they were not; they were aquaplaning).

FIGURE 2. Airplane overshooting the runway

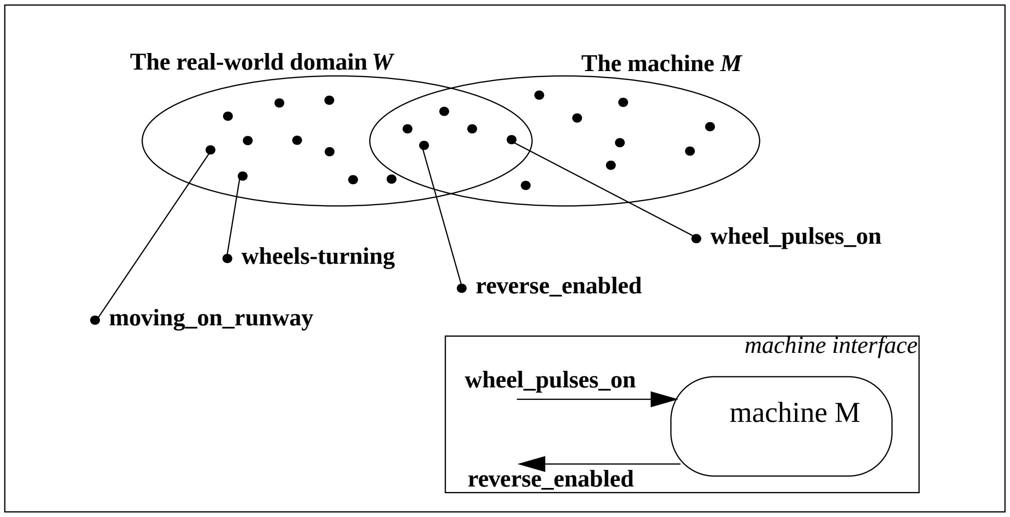


Fig. 2 shows the phenomena that we are concerned with. The requirement was

requirement R: $\text{reverse\_enabled} \equiv \text{moving\_on\_runway}$.

The developers thought that the real-world domain was described by

description of world $W.desc$:
$$\begin{cases} \text{wheel\_pulses\_on} \equiv \text{wheels\_turning} \\ \text{wheels\_turning} \equiv \text{moving\_on\_runway} \end{cases} \qquad \text{(Eq. 4)}$$

So they derived the specification

specification of machine $M.spec$: $\text{reverse\_enabled} \equiv \text{wheel\_pulses\_on}$.

For the above description of the real-world domain, specification correctness (Eq. 2) given by $W.desc \wedge M.spec \rightarrow R$ is indeed a theorem.

Unfortunately, the developers did not understand the real-world domain correctly. The first property listed in (Eq. 4) was indeed a correct description of the real-world domain. But, the second property "$\text{wheels\_turning} \equiv \text{moving\_on\_runway}$" was not. When the wheels are aquaplaning on a wet runway, the second property in fact fails to hold, because "moving_on_runway" is true but "wheels_turning" is false. The correct description of the real-world was instead

$$W.desc \begin{cases} \text{wheel\_pulses\_on} \equiv \text{wheels\_turning} \\ (\text{wheels\_turning}) \vee (\text{aquaplaning}) \equiv \text{moving\_on\_runway} \end{cases} .$$

With this correct description of the domain, a machine satisfying the specification $M.spec$ listed above will no longer satisfy the requirements, because specification correctness (Eq. 2) no longer holds. It is thus crucial to get an accurate description of the real-world domain.

2.3 Descriptions

The central activity of software development is description. Any software project will need many different kinds of descriptions. These descriptions provide essential documentation of the software. Here are some of the main types of descriptions [21].

- *Specifications* or *requirements* state the *required* properties of a product (e.g. $M.spec$ and R). The difference between a requirement and specification was described in the previous sub-section.
- *Behavioural descriptions* state the *actual* properties of an entity or product. Behavioural descriptions describe the visible properties of an entity without discussing how it was constructed. The real-world description (Eq. 4) is an example of a behavioural description — in this case it is not a product or program that is being described but the environment (runway) in which the product (the airplane) will operate.
- *Constructive descriptions* also state *actual* properties of a program, but they also describe how a program is composed of other programs. Program text is an example of a constructive description. For example, the program text for the module in Fig. 6 describes how the body of the module is constructed from two private routines.

Specifications and requirements are expressed in what grammarians call the optative mood, i.e. they express a wish. Behavioral and constructive descriptions are expressed in the indicative mood, i.e. they assert a fact. Thus, a description may include properties that are not required, and a specification may include properties that a (faulty) product may not possess.

We cannot necessarily tell from a list of properties whether we are dealing with a behavioural description of an already existing product, or whether the list of properties is a specification of what we hope will eventually become a product. It is therefore crucial for the writer to make the relevant distinction. Once we have demonstrated implementation correctness (Eq. 1), then the specification itself becomes a description.

Although mathematics can be used for all descriptions, not all descriptions need necessarily be mathematical. We can distinguish between rough sketches, designations, definitions and refutable descriptions [12].

A *rough sketch* (e.g. Fig. 1) is a tentative and incomplete description of something that is in the process of being explored or invented. It uses undefined terms to record half-formed or vague ideas, and is useful especially in the early development phase.

A *designation* singles out some particular kind of phenomenon that is of interest, tells us informally in natural language how to recognize it, and gives a name by which it will be denoted. Here are some designations:

```
wheel_pulses_on(x): BOOLEAN
                          -- designates the situation in which the sensor of airplane x
                          -- detects that its wheels are turning
aquaplane(x): BOOLEAN
                          -- designates the situation in which airplane x aquaplanes on the runway
```

Having made designations, we can now proceed with more precise descriptions. *Definitions* introduce new names in terms of already existing descriptions. Here is a definition of $\text{plane\_moving}(x)$:

$$\text{plane\_moving}(x) \overset{\text{def}}{=} [\text{wheel\_pulses\_on}(x) \vee \text{aquaplane}(x)]\,.$$

A *refutable description* describes some domain, saying something about it that can — in principal — be refuted or disproved.

Predicate logic provides a means for expressing refutable descriptions. A predicate can either be *valid* (true in all behaviours of the product), a *contradiction* (false in all behaviours) or *contingent* (true in at least one behaviour and false in at least one).

A useful predicate for specifications and requirements is one that is contingent. The predicate *true* (or any theorem for that matter) is not a useful specification of a product because any behaviour of the product satisfies *true*. So too, *false* is not a useful specification as it is satisfied by no behaviour. A useful specification is one that satisfies precisely and only those behaviours that we wish to observe in the product.

Real-world descriptions should also be refutable. For example, the real-world property "$\text{wheels\_turning} \equiv \text{moving\_on\_runway}$" (Eq. 4) is a refutable description. It would be refuted by an observation in which "moving_on_runway" is true but "wheels_turning" is false. This is exactly the behaviour that is observed when the wheels aquaplane.

The use of mathematical descriptions throughout software documentation and design is an idealization. Not all requirements can necessarily be captured by predicates, at least not easily. Sometimes rough sketches must be used, or we must resort to vague qualifications such as "approximately" or "preferably". The requirements will not necessarily remain constant. Any change may invalidate the entire logical structure (although engineers will often find ingenious ways of preserving work already completed). The over-riding imperative to deliver a product on time and within cost will often mean that logical analysis and calculation cannot always be performed, at least in full detail.

The reality of software development does not mean that precise mathematical descriptions cannot find a place. The software engineer will seek a balance between rough sketches and precise description and calculation. Useful software development methods will therefore allow the software engineer to choose the appropriate balance between mathematical and informal description.

3.0 Using Logic for Descriptions and Calculations

What kind of mathematics should software engineering students be taught? Like other engineering students they should have a working knowledge of classical mathematics such as calculus, linear algebra and probability and statistics. But, the description of software products requires the use of functions with many points of discontinuity. The study of continuous functions must thus be supplemented with that of predicate logic and discrete mathematics. We illustrate this type of knowledge with a simple example that will also illustrate how logic may be used to

- make informal descriptions precise,
- calculate properties of products (by proving theorems), and
- understand the role of counterexamples. A counterexample can be used to show that a conjecture about a product is not a theorem.

3.1 Informal specification of the password module

Consider the following informal specification:

> *A personal digital assistant (PDA) needs a PASSWORD_MANAGEMENT module that allows the user of the PDA to enter a password. The user should not be allowed to access the verification routine more than six times. The user only gets five tries at entering the password; if the user entry matches the stored password, then the PDA can be operated on by the user. If the password does not match, the PDA remains inoperative. On the sixth try, no password checking is done — instead an alarm flag is immediately raised. The alarm flag might be used by other modules to turn off the PDA or inform the owner of unauthorized access.*

We use an Eiffel class [17] to specify the password management module. Eiffel is an example of a development environment that can be used to build software seamlessly from specifications to programs. At any one time the developer works on only one product — the machine — which successive stages and activities will progressively enrich. This does not mean that there is only one view of the machine. There are a variety of views available. Each view is a description of a different aspect of the machine. For example, the short format of a class documents the class interface, i.e. its exported features, their specifications (pre/post conditions) and the class invariant. The supplier of the class can view the interface as well as the implementation. Class relationships such as the client-supplier relationship or the inheritance relationship can be viewed. Classes can be grouped into clusters, which can be related to other clusters using the same relationships that are applied to single classes. The designer can start at the abstract architectural design level, and then generate the Eiffel class skeletons, or start working on individual classes and work up to the architectural design level, or alternate between these two views.

3.2 Formalizing the specification — design by contract

The password management module can be specified by the Eiffel class shown in Fig. 3. No implementation detail is given. The class starts by defining the various attributes (state) of the module. The behaviour of the routine *verify_user* is specified by a precondition (the *requires* clause) and a postcondition (the *ensures* clause). The precondition describes the set of all initial states (prestates) and the postcondition describes the set of all final states (poststates) for the routine. The intermediate states are irrelevant detail and hence are ignored in the specification.

FIGURE 3. Eiffel specification of the password management module

```
class PASSWORD_MANAGEMENT feature

     -- attributes (the "state space")
     alarm: BOOLEAN          -- signal illegal entry
     operate: BOOLEAN        -- user may operate PDA
     p1: PASSWORD            -- p1 is of password type
     i: INTEGER              -- count of password tries

     -- initialization routines and
     -- password change routines to be added.

     -- initially: $\neg alarm \wedge \neg operate \wedge (i = 0) \wedge (p1 = \text{some\_string})$

     verify_user(p2: PASSWORD)
                             -- routine to verify password p2
     require $\neg alarm \wedge \neg operate$
                             -- this is the precondition
     ensure $(g_1 \rightarrow e_1) \wedge (g_2 \rightarrow e_2) \wedge (g_3 \rightarrow e_3)$
                             -- this is the postcondition
                  $g_1 \overset{\text{def}}{=} (old\ i < 6) \wedge (old\ p1 = p2)$
                  $g_2 \overset{\text{def}}{=} (old\ i < 6) \wedge (old\ p1 \neq p2)$
                  $g_3 \overset{\text{def}}{=} (old\ i \geq 6) \wedge (old\ p1 \neq p2)$
     -- where     $e_1 \overset{\text{def}}{=} (i = 0) \wedge operate \wedge \neg alarm \wedge (p1 = old\ p1)$
                  $e_2 \overset{\text{def}}{=} (i = old\ i + 1) \wedge \neg operate \wedge \neg alarm \wedge (p1 = old\ p1)$
                  $e_3 \overset{\text{def}}{=} (i = 0) \wedge \neg operate \wedge alarm \wedge (p1 = old\ p1)$
     end

     invariant $i \geq 0$
                    -- all routines preserves the invariant
end
```

The precondition and postcondition express a contract between the client and the programmer. The client of the module has the obligation to invoke the routine only when the precondition holds; the client may benefit from the result of the routine as described by the postcondition. The supplier of the routine (the programmer) has the obligation to ensure that the postcondition holds; the precondition is a benefit to the supplier for the routine need not deal with cases not covered by the precondition. This is called *design-by-contract* in which the obligations and benefits of clients and suppliers are delineated.

In postconditions, the notation *old expression* denotes the value of *expression* in the prestate. Hence, $(i = old\ i + 1)$ specifies that the value of *i* in the poststate must be precisely one greater than the value of *i* in the prestate. The routine parameter $p2$ does not change value, hence there is no old value for $p2$. The class invariant $i \geq 0$ must be preserved by each routine.

The specification of the class via routine pre/postconditions and invariants is the formal counterpart of the informal specification. The precision of the formal specification improves the documentation of the program as well as serving as a contract between the client and the supplier. In addition, the formal specification of the class can now be used to calculate the properties of the class. Here are some questions that we might want to ask about the specified class.

Conjecture 1 — input coverage: Is every input handled?

The postcondition of the *verify_user* routine is in a special guarded expression format, where each guard g_i describes a specific input and its corresponding consequent e_i describes the required output (Fig. 3). The specifier of the contract might therefore want to show the validity of

$$\neg(old\ alarm) \wedge \neg(old\ operate) \rightarrow (g_1 \vee g_2 \vee g_3)\,. \qquad \text{(Eq. 5)}$$

The above conjecture expresses the assertion that any input satisfying the precondition must also satisfy the disjunction of the guards in the postcondition. It is up to the client to ensure that the precondition is satisfied. If the conjecture holds then the specification has the desirable property that it deals with all inputs allowed by the precondition.

Input coverage (Eq. 5) is an anti-theorem (i.e. it is not a theorem) because the state described by $\neg(old\ alarm) \wedge \neg(old\ operate) \wedge (old\ i = 6) \wedge (old\ p1 = p2)$ is a counterexample to the conjecture.

The counterexample to the conjecture informs the specifier that a certain input is unhandled. Which input? — the input for which the user of the PDA is on the sixth attempt with a correct password. The informal specification states that on the sixth try an alarm should be raised irrespective of whether the supplied password is correct or not. The formal specification would however allow the alarm to be disabled if the password is correct on the sixth try.

The counterexample to the input coverage conjecture suggests that the routine postcondition be rewritten as follows

$$(g_1 \rightarrow e_1) \wedge (g_2 \rightarrow e_2) \wedge (g_3 \rightarrow e_3)$$

$$where \begin{cases} g_1 \overset{\text{def}}{=} (old\ i < 6) \wedge (old\ p1 = p2) \\ g_2 \overset{\text{def}}{=} (old\ i < 6) \wedge (old\ p1 \neq p2) \\ g_3 \overset{\text{def}}{=} (old\ i \geq 6) \\ e_1 \overset{\text{def}}{=} (i = 0) \wedge operate \wedge \neg alarm \wedge (p1 = old\ p1) \\ e_2 \overset{\text{def}}{=} (i = old\ i + 1) \wedge \neg operate \wedge \neg alarm \wedge (p1 = old\ p1) \\ e_3 \overset{\text{def}}{=} (i = 0) \wedge \neg operate \wedge alarm \wedge (p1 = old\ p1) \end{cases} \qquad \text{(Eq. 6)}$$

which is the same as the first specification except for the guard g_3. Using Logic **E** (see [7] and discussion below), we can prove that the input coverage conjecture

$$\neg(old\ alarm) \wedge \neg(old\ operate) \rightarrow (g_1 \vee g_2 \vee g_3) \qquad \text{(Eq. 7)}$$

is a theorem. We assume the antecedent, and prove under this assumption that the consequent is a theorem. The proof transforms the consequent $(g_1 \vee g_2 \vee g_3)$ into a known theorem.

Assume: $\neg(old\ alarm) \wedge \neg(old\ operate)$.

$g_1 \vee g_2 \vee g_3$

= < definitions of g_1, g_2 >

$(old\ i < 6 \wedge old\ p1 = p2) \vee (old\ i < 6 \wedge old\ p1 \neq p2) \vee g_3$

= <$(old\ p1 \neq p2) \overset{\text{def}}{=} \neg(old\ p1 = p2)$ >

$(old\ i < 6 \wedge old\ p1 = p2) \vee (old\ i < 6 \wedge \neg(old\ p1 = p2)) \vee g_3$

= <distributivity of conjunction over disjunction (3.46) >

$(old\ i < 6 \wedge (old\ p1 = p2 \vee \neg(old\ p1 = p2)) \vee g_3$

= <excluded middle (3.28) can be replaced by *true*, by TE (theorem equivalence)>

$((old\ i < 6) \wedge true) \vee g_3$

= < identity of conjunction (3.39) >

$(old\ i < 6) \vee (old\ i \geq 6)$

= < arithmetic: $(old\ i < 6 \vee old\ i \geq 6) \equiv true$ >

$true$ -- (3.4). Q.E.D.

(In the end, the assumption was not needed for the proof).

The inference rules for Logic E are described in Fig. 4 and derived rules such as theorem equivalence (TE) are provided in the Appendix (Sect. 6.0). Each step is justified by the Leibniz inference rule (replacement of equals for equals). The hints in angled brackets mention which theorem was used to obtain the replacement expression (the numbers refer to theorem numbers in [7])[5]. The transitivity inference rule is applied five times to con-

5. A list of the basic theorems of logic **E**, including all the theorems used in this paper, can be obtained from http://www.ariel.cs.yorku.ca/~logicE/misc/logicE_theorems.pdf.

FIGURE 4. The calculational logic E

A textbook for logic **E** [7] provides a list of axioms for propositional logic, predicate logic and theories in various discrete domains (e.g. sets, integers, combinatorics, and universal algebra).

In logic **E**, the predicate $E[z:=P]$ is defined to be the same predicate as E except that every free occurrence of z in E is replaced by the expression P using safe substitution. For example, given the propositional axiom $q \vee \neg q$ (this is predicate E), then $(q \vee \neg q)[q:=(x>5)] = (x>5 \vee \neg(x>5))$. Using this notation for safe substitution, logic **E** has 4 rules of inference:

Leibniz: $\dfrac{P = Q}{E[z:=P] = E[z:=Q]}$ **Substitution:** $\dfrac{E}{E[z:=P]}$

Transitivity: $\dfrac{E_1 = E_2, E_2 = E_3}{E_1 = E_3}$ **Equanimity:** $\dfrac{E_1, E_1 \equiv E_2}{E_2}$

An inference rule states that the predicate below the line is a theorem provided the predicates above the line are also theorems. For example, Substitution asserts that $E[z:=P]$ is a theorem provided E is a theorem. From the axioms and rules of inference, the text derives a large number of useful theorems in various domains. Proofs are structured in the equational style:

$$\begin{array}{ll} & E[z:=P] \\ = & \langle P \equiv Q \rangle \\ & E[z:=Q] \end{array}$$

The above layout is justified by the Leibniz rule. The hint $P \equiv Q$ is usually obtained by applying the Substitution rule to an axiom or theorem. Substitution is often used without mention when it is obvious. The inference rule Transitivity is used to conclude that the first expression in a sequence of calculational steps is equal to the last expression (or vice versa). Equanimity allows us to conclude that if the first expression is a theorem, then the last expression is also a theorem. Since the use of inference rules is obvious from the structure of the proof, we achieve brevity and readability. It is clear at each step what the justification for the step is. Some additional theorems, that can be derived using the inference rules, include:

3.84(a): $(e_1 = e_2) \wedge E[z:=e_1] \equiv (e_1 = e_2) \wedge E[z:=e_2]$

3.84(b): $(e_1 = e_2) \rightarrow E[z:=e_1] \equiv (e_1 = e_2) \rightarrow E[z:=e_2]$

where e_1, e_2 are expressions of the same type, and E is a predicate containing an occurrence of z.

clude that the predicate at the top is equivalent to the predicate at the bottom. Finally, since the bottom predicate is itself a theorem, the equanimity inference rule allows us to conclude that $(g_1 \vee g_2 \vee g_3)$ is also a theorem.

The *equales* symbol (≡) is used for equality of two expressions that are both of type boolean. In general, a calculational proof in Logic **E** mixes equalities (= or ≡) and implications ($\rightarrow$) because the composition of the relations $\equiv$ and $\rightarrow$ yields the relation $\rightarrow$. Therefore, to prove that $A \rightarrow D$ is a theorem, we need only write the following:

```
  A
=        < hint why A = B >
  B
⇒        < hint why B → C >
  C
=        < hint why C = D >
  D
```

The brevity of the equational style of Logic **E** makes it easier to calculate program properties, and is usually much shorter than trying to show validity by constructing a truth table (or enumerating all possible states). More importantly, the software engineering student will want to make use of modern theorem provers to do routine calculations. The use of theorem provers presupposes the type of knowledge developed by familiarity with logic such as formalization of informal arguments, proofs of theorems from axioms and the use of counterexamples to show that there is no proof.

The following generalization of the input coverage conjecture illustrates the use of theorem provers such as PVS [20].

Conjecture 2 — implementability conjecture

An Eiffel specification of a routine with a precondition P and a postcondition E can be combined into a single before/after predicate $(old\ P) \rightarrow E$. A before/after predicate is any predicate that has occurrences of variables prefixed with “old” (which refers to the value of the variable in the prestate) as well as unadorned variables (which refers to the value of the variable in the poststate). A before/after specification asserts that if the precondition holds, then the routine must behave according to E, else any behaviour (including non-termination) is acceptable. This captures the Eiffel notion that the supplier of the routine is responsible for dealing only with inputs specified by the precondition.

The implementability conjecture states that *for any* input to a routine that satisfies the precondition there must *exist* a well-defined output. For example, consider a routine whose precondition is *true* and whose postcondition is $(old\ x \geq 0) \wedge (i = 0)$. The before/after specification of the routine is $true \rightarrow (old\ x \geq 0) \wedge (i = 0)$. This specification is not implementable because

$$(\forall old\ \mathrm{x} | (\exists i | true \rightarrow old\ x \geq 0 \wedge i = 0))$$

is an anti-theorem (in fact, there is a simple proof in Logic **E** to show that it is equivalent to *false*). No program can implement this specification because there is no output value for *i* that can “truthify” the postcondition for a negative input (e.g. *old x* = -5). Of course, it is of vital importance that a specification be implementable.

The before/after specification *spec* for the *verify_user* routine is

$$spec \overset{\mathrm{def}}{=} [\neg(old\ alarm) \wedge \neg(old\ operate)] \rightarrow [(g_1 \rightarrow e_1) \wedge (g_2 \rightarrow e_2) \wedge (g_3 \rightarrow e_3)] \quad \text{(Eq. 8)}$$

where the g_i and e_i are defined in (Eq. 6). The implementability conjecture for this specification is

$$(\forall old\ i, old\ p1, p2 | (\exists i, operate, alarm, p1 | spec)) . \qquad \text{(Eq. 9)}$$

The proof of the above conjecture can be done in Logic **E**, but it is easier to use the PVS theorem prover. The implementability conjecture is shown at the bottom of Fig. 5.

FIGURE 5. Using the PVS theorem prover to state and prove conjectures

```
password : THEORY
begin

% attributes or state
passwordtype: TYPE
alarm,old_alarm,operate, old_operate: VAR bool
i, old_i: VAR nat
p1,p2,old_p1: VAR passwordtype

%% Input Coverage Conjecture
inputs_covered1: CONJECTURE
    (NOT old_alarm AND NOT old_operate)
    IMPLIES
    ((old_i < 6 AND old_p1 = p2)
    OR (old_i < 6 AND old_p1 /= p2)
    OR (old_i >= 6 AND old_p1 /= p2))
%% Counter-example is old_i = 6 AND old_p1 = p2, i.e.

%% New before/after specification of verify_user
spec(i,old_i,operate,alarm,p1,old_p1,p2): bool =
    (NOT old_alarm AND NOT old_operate)
    IMPLIES
    ((old_i < 6 AND old_p1 = p2 IMPLIES
        (i = 0) AND operate AND NOT alarm and p1 = old_p1)
     AND
    (old_i < 6 AND old_p1 /= p2 IMPLIES
        (i = old_i + 1) AND NOT operate AND NOT alarm AND p1 = old_p1)
     AND
    (old_i >= 6 IMPLIES
        alarm AND NOT operate AND i = 0 AND p1 = old_p1))

%% Input Coverage Conjecture again
inputs_covered2: CONJECTURE
    (NOT old_alarm AND NOT old_operate)
    IMPLIES
    ((old_i < 6 AND old_p1 = p2)
    OR (old_i < 6 AND old_p1 /= p2)
    OR (old_i >= 6))
%% QED. That works!

%% Specification Implementability Conjecture
implentability : CONJECTURE
    (EXISTS i, operate, alarm, p1:
    NOT old_alarm AND NOT old_operate
    IMPLIES
    spec(i,old_i,operate,alarm,p1,old_p1,p2))
%% By convention, above is universally quantified over all free variables
%% This proves. Q.E.D.
end password
```

In PVS, the input coverage conjecture is proved automatically. However, the implementability conjecture was proved with some interaction from the user using existential instantiation three times. This illustrates one of the issues involved in using theorem provers: where a theorem cannot be discharged automatically, the user has to know a proof in outline in advance in order to provide proper guidance to the prover.

Another property that we might want specifications to be endowed with is that the guards should treat disjoint domains. Implementability and disjoint domains are examples of generic properties that all good specifications should have. In addition, there may be properties that are unique to a given specification or description. For example, we might want to show for *verify_user* that

$$spec \wedge (old\ P) \rightarrow ((old\ \mathrm{i}) \geq 6 \rightarrow alarm)$$

i.e. a consequence of the *verify_user* specification is that the alarm is always raised on the sixth attempt. There is a simple proof in Logic E to show that the above conjecture is a theorem. The above calculations show that logic is a powerful tool for validating the correctness and implementability of specifications.

Once we have validated a module specification, Logic **E** and theorem provers can be used to develop programs from their contracts [1,6,9,18]. Although the complete development from specifications to implementations can be done mathematically, this may not be necessary in all cases. Nor may it be necessary to provide a complete description or specification of all the properties of software products. Students need to develop skill in isolating useful and important properties.

3.3 Developing programs from specifications

We have seen that requirements and specifications are assertions in predicate calculus. But programs can also be described by predicates [9]. The fundamental construct of sequential programs is the assignment statement, e.g. $x{:=}x+y$, which causes a change of state in the machine. We have already see how a before/after predicate can be used to describe such changes. Using Eiffel notation we write

Eiffel convention for before/after predicates: $(x = old\ x + old\ y) \wedge (y = old\ y)$. (Eq. 10)

In the sequel, we will use the Z convention [27] in which primed names such as x' and y' denote the final values of the variable in the poststate, whereas unprimed names such as x and y stand for their initial values in the prestate. The effect of the assignment can then be formally described by the predicate

Z convention for before/after predicates: $(x' = x + y) \wedge (y' = y)$. (Eq. 11)

There is no essential difference between the Eiffel and Z convention. In both cases we have designations for prestates and poststates. For our purposes the prime notation is more concise.

The program $x, y{:=}x+y, 2y$, which is the simultaneous assignment to x and y, is described by: $(x, y{:=}x+y, 2y) \stackrel{\text{def}}{=} (x' = x+y) \wedge (y' = 2y)$. Consider a specification $m.spec$ of a routine m of a class C defined as follows:

```
class C feature
    x,y: INTEGER -- attributes
    m            -- routine to double y while keeping (x-y) constant, i.e.
                 -- m.spec : (x' – y' = x – y) ∧ (y' = 2y)
end C
```

We can use logical calculation to derive an implementation (code) for the routine r from the specification $m.spec$ as follows:

$m.spec$

= <definition of $m.spec$>

$(x' - y' = x - y) \wedge (y' = 2y)$

= <Leibniz substitution 3.84(a) (see Fig. 4)>

$(x' - 2y = x - y) \wedge (y' = 2y)$

= <arithmetic: $(x' - 2y = x - y) \equiv (x' = x + y)$>

$(x' = x + y) \wedge (y' = 2y)$

= <using the definition of simultaneous assignment>

$(x, y{:=}x + y, 2y)$ -- this is the implementation $m.prog$

The above calculation derives a not totally obvious program $m.prog$, defined by $(x, y{:=}x + y, 2y)$, from its specification $m.spec$. The program $m.prog$ satisfies the implementation correctness condition $m.prog \rightarrow m.spec$. Further refinement of the code might be needed if a programming language is used that does not support simultaneous assignment, but the same kinds of calculation apply to such derivations [9].

3.4 Logic as a design calculus

Logical connectives and quantifiers such as conjunction, implication and existential quantification can be used as a design calculus for software development.

Implication can be used for program refinement, also called program correctness (Eq. 1). A program $prog$ implements a specification $spec$ if $prog \rightarrow spec$, i.e. every behaviour satisfying the program description also satisfies the specification.

We can hide the internal behaviour of the program with the existential operator. The visible program behaviour is $(\exists v | prog)$ where v stands for the local program variables. What is observed inside the machine is of no concern to a client of the machine. Then, provided v does not occur free in $spec$, program refinement becomes $(\exists v | prog) \rightarrow spec$. This is because $(\forall v | prog \rightarrow spec) \equiv ((\exists v | prog) \rightarrow spec)$ (provided v does not occur free in $spec$).

Conjunction is a general way to express connection and interaction in an assembly constructed from two or more components. As an example, consider two classes which satisfy the class invariants i_1 and i_2 respectively. If both classes are in use in a given program then we are guaranteed that the assembly behaves according to $i_1 \wedge i_2$.

If a specification is complex, we can decompose it into two sub-specifications (or designs) D_1 and D_2, provided $D_1 \wedge D_2 \rightarrow spec$. Each design can then be implemented by a separate program $prog_1$ and $prog_2$, provided $(prog_1 \rightarrow D_1)$ and $(prog_2 \rightarrow D_2)$ are theorems. The final implementation is $prog_1 \wedge prog_2$, and we are guaranteed by propositional calculus that $prog \rightarrow spec$.

We have shown how logic can be used for describing requirements, specifications, and programs. We also showed that logic can be used as a descriptive calculus throughout the software life-cycle including design, implementation and documentation. Finally, the logical calculational format can be used in various phases of the software life-cycle, e.g. to derive a program that implements a specification, or to establish that an assembly of components satisfies a requirement if the components satisfy their specifications. The calculational format has the virtues of brevity and readability that make it easy to use, and the availability of the text [7] means that the calculational format can be taught to students early in a Computer Science programme.

At York University in Toronto, we are updating our mathematics and computer science curriculum to adopt the use of the calculational format. Our first-year logics and discrete mathematics courses for computer science students are using the calculational approach, based on the text [7]. The calculational method has also been applied in third year program verification course. Future changes in our curriculum will likely see the calculational method applied throughout our software engineering curriculum.

4.0 A simple case study — cooling tank

In the previous section we described how calculational logic can be used in all phases of software design. In this section, we present a small and simple case study that will illustrate the use of logical methods and tools through all phases of software design from requirements to implementations.

The case study will involve the use *conditional expressions* such as

$$\text{if } b \text{ then } e_1 \text{ else } e_2 \tag{Eq. 12}$$

where b is of type boolean and e_1, e_2 are any two expressions of the same type. For conciseness we also use the abbreviation

$$b\Big|_{e_2}^{e_1} \tag{Eq. 13}$$

(see Appendix in Sect. 6.0). Logic **E** as described in [7] provides the two axioms

$$\text{(10.9) } b \rightarrow \left(b\Big|_{e_2}^{e_1} = e_1\right) \qquad\qquad \text{(10.10) } \neg b \rightarrow \left(b\Big|_{e_2}^{e_1} = e_2\right)$$

for conditional expressions. We will need more powerful theorems to simplify calculation. We therefore refer the reader to the Appendix (Sect. 6.0) in which further theorems of conditional expressions are listed. The Appendix also provides a proof of theorem (10.14a) below which is an illustration of the utility of Logic E for stating and developing new theory.

(10.14a): $\left(p \rightarrow E\left[z{:=}b\Big|_{e_2}^{e_1}\right]\right) \equiv (p \rightarrow E[z{:=}e_1])$ provided that $p \rightarrow b$ is a theorem.

Theorem (10.14a) provides a method for simplifying a complex expression consisting of conditional subexpressions to a simpler expression with the conditional eliminated. Consider a variable x with $type(x) = NATURAL$. It follows that $(x = 0) \vee (x = 1) \vee (x > 1)$ is a theorem. Using "IF-transform" reasoning (Appendix Sect. 6.2.3) the following is a theorem:

$$[x' = x + (x \le 1\Big|_{y}^{9}) - (x \ge 1\Big|_{z}^{1})] \equiv \begin{bmatrix} (x = 0) \rightarrow (x' = x + 9 - z) \\ \wedge\, (x = 1) \rightarrow (x' = x + 9 - 1) \\ \wedge\, (x > 1) \rightarrow (x' = x + y - 1) \end{bmatrix}. \tag{Eq. 14}$$

We now present an informal description of the case study.

4.1 A cooling tank example

“A tank of cooling water shall generate a low level warning when the tank contains 1 unit of water or less. The tank shall be refilled only when the low level sensor comes on. Refilling consists of adding water until there are 9 units of water in the tank. The maximum capacity of the tank is 10 units, but the water level should always be between 1 and 9 units. The sensor readings are updated once every cycle, i.e. once every 20 seconds. Every cycle, one unit of water is used. It is possible to add up to 10 units of water in a cycle”

A programmer, looking at the above problem, might immediately write plausible code for the *controller* module as shown in Fig. 6. The body of the module executes “*set_alarm; fill_tank*” once every cycle.

FIGURE 6. Faulty code for the cooling tank example

```
Module controller
Inputs
        level: LEVEL          -- input from tank, where type LEVEL = {0 .. 10}
Outputs
        alarm: BOOLEAN        -- raises tank alarm.Initially false.
        in: LEVEL             -- setpoint for tank input valve. Initially 0.
Body
        every 20 seconds
        do
        set_alarm; fill_tank
        end
Private routines used in Body
        set_alarm is          -- set the alarm if tank level is low
            do
            alarm := (level <= 1)
            end
        fill_tank is          -- fill tank if level is low, otherwise do nothing
            do
            if       level = 0 then in := 9
            elseif   level = 1 then in : = 8
            else     in : = 0
            end
end
```

The *set_alarm* routine raises the flag *alarm* if the tank level goes below 1 unit. The *fill_tank* routine sets the tank input setpoint *in* to 9 units if the tank level is already at 0 units and to 8 units if the tank level is at 1 unit. In this way, the tank is refilled to exactly 9 units at the end of the cycle.

Apart from the fact that the above program is wrong (as we shall see later), we have also not followed the recommended design method presented earlier (Sect. 2.0). In fact, without a specification that satisfies specification correctness (Eq. 2), we cannot even begin to debug the program.

Our rational software design method (Sect. 2.0) requires that we first divide the system of interest into the real-world domain *W* and the machine *M*, and identify the relevant phenomena. The real-world domain, in this case, is the cooling tank with its outflow of water *out* and inflow of water *in*.

The rough sketch in Fig. 7 illustrates the phenomena of the real-world domain, including phenomena shared with the machine (*in*, *level* and *alarm*). The water outflow *out* is not a shared phenomenon as the machine cannot measure it. The comment in the figure indicates that the informal requirements cannot be precise; the figure therefore provides a precise description of the outflow as a function of water level. One of the benefits of mathematical descriptions is that they can be used to remove ambiguities present in the informal descriptions.

FIGURE 7. Rough sketch of the cooling tank identifying the phenomena of interest

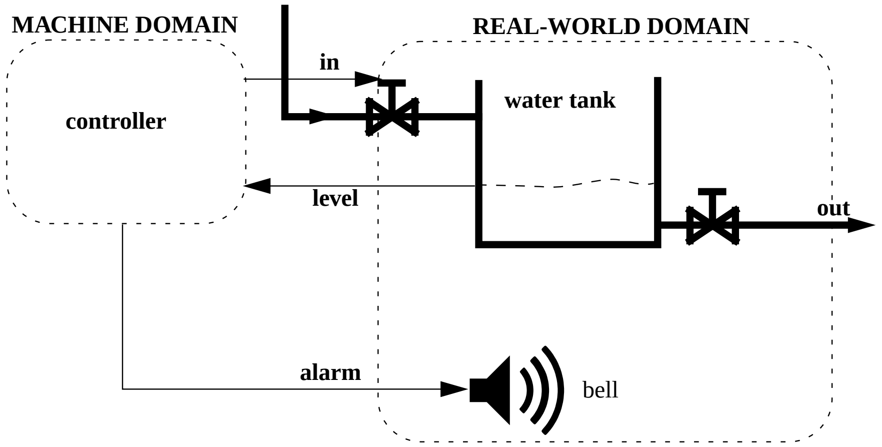
Real-world phenomena

in, level: LEVEL	-- phenomena shared with the machine
alarm: BOOLEAN	
out: LEVEL	-- phenomena not shared with the machine

Real-world description

$$out = \text{if } level \geq 1 \text{ then } 1 \text{ else } 0 \qquad \text{(Eq. 15)}$$

The informal requirements state: “every cycle, one unit of water is used”. This cannot be precise. If $(level = 0)$ at the beginning of a cycle, then there may not be outflow in that cycle. We will assume that the above law (Eq. 15) describes the outflow as a function of level. This corresponds to a scenario in which the outflow valve (a) is automatically opened only when the level reaches 1 unit, and (b) releases exactly 1 unit every cycle so long as it is open. It is up to the software engineer to ascertain from the domain specialists the precise behaviour of the real-world phenomena.



Having identified the phenomena of interest, the next step is to write the requirements for the cooling tank. We assume that the machine will read the *level* sensor at the beginning of a cycle, immediately calculate the new values for *in* and *alarm,* and then repeat this action at the beginning of the next cycle 20 seconds later. We may therefore describe the requirements in terms of the variables of interest at the beginning and at the end of an arbitrary cycle.

cooling tank requirement $R \overset{\text{def}}{=} (R_1 \wedge R_2 \wedge R_3) : \begin{cases} R_1 \colon (1 \le level' \le 9) \\ R_2 \colon level' = (level \le 1|^{9}_{level - out}) \\ R_3 \colon (alarm \equiv level \le 1) \end{cases}$ (Eq. 16)

The initial value of the water level, the alarm signal, and the outflow are designated by *level*, *alarm* and *out* respectively. The value of the water level at the end of the cycle is designated by $level'$. The requirement thus states that the final value of the water level must be between the stated bounds, the tank must be filled (at the end of the cycle) if it goes low (at the beginning of the cycle), and the alarm bell must be sounded (at the beginning of the cycle) if the level is low.

The next step in the recommended design method is to describe the properties characterizing the real-world domain.

real-world description $W.desc \overset{\text{def}}{=} W.d_1 \wedge W.d_2 : \begin{cases} W.d_1 \colon level' = (level + in - out) \\ W.d_2 \colon out = (level \ge 1|^{1}_{0}) \end{cases}$ (Eq. 17)

The domain property $W.d_1$ is derived from a physical law that says flow must be preserved, i.e. the flow at the end of a cycle is what the original level was, adjusted for inflows and outflows. The property $W.d_2$ asserts that the outflow at the beginning of a cycle is one unit (see informal description) unless there is no water left to flow out (this part was not in the informal description, but must be added if the description is to be precise).

In the absence of a controller, the "free" behaviour of the cooling tank will not satisfy the requirements. This is because the inflow setpoint *in* can be set to any value. In order to satisfy the requirements, we must therefore specify a machine.

The requirements and real-world descriptions are allowed to refer to the outflow *out*. However, since there was no sensor for it, *out* is not a shared phenomenon, and the machine (the controller) may therefore *not* refer to it. Here is a first attempt at the machine specification:

$$\begin{bmatrix} in = (\text{if} & level = 0 \text{ then } 9 \\ & \text{elseif } level = 1 \text{ then } 8 \\ & \text{elseif } level > 1 \text{ then } 0) \end{bmatrix} \wedge (alarm \equiv level \le 1) \qquad \text{(Eq. 18)}$$

We have assumed that the machine works much faster than the cycle time of the cooling tank. Therefore, the machine instantaneously sets *in* and *alarm* to the values described above at the beginning of each cycle. The specification refers to shared phenomena only.

The controller module described earlier (Fig. 6) implements the specification of (Eq. 18). The specification might at first sight appear correct, for it adds 9 units of water if the level is zero, and 8 units of water if the level is one (1 + 8 = 9), else nothing is added. However, the machine specification is wrong, as can be seen by a counterexample. Consider a state at the beginning of a cycle in which $(level = 1)$. According to the above specification, $(in = 8)$. Thus $(out = 1)$ by $W.d_2$. Hence, by $W.d_1$,

$$\begin{aligned} level' &= (level + in - out) \\ &= 8 \end{aligned}$$

so the requirement R_2 will not be satisfied because the tank is supposed to be at 9 units of water at the end of the cycle. The failed specification did not take into account the fact that there is an outflow of 1 unit when the level is at 1 unit (recall that there is zero outflow when the level is zero). The counterexample was detected when the logical calculation for specification correctness (Eq. 2) was performed.

A correct specification for the controller is:

$$\textbf{machine specification } M.spec \overset{\text{def}}{=} M.s_1 \wedge M.s_2 : \begin{cases} M.s_1\text{: } in = (level \le 1|_0^9) \\ M.s_2\text{: } alarm \equiv (level \le 1) \end{cases} \qquad \text{(Eq. 19)}$$

which states that 9 units must be added irrespective of whether the level is zero units or one unit of water at the beginning of a cycle. Specification correctness (Eq. 2) holds if we can show the validity of

$$(\forall level\text{: } LEVEL | W.\text{desc} \wedge M.spec \rightarrow R) \qquad \text{(Eq. 20)}$$

which asserts that no matter what the *level* is at the beginning of a cycle (provided it is of type *LEVEL*), and provided the application domain satisfies the real-world description *W.desc* (Eq. 17) and the machine its specification, then the requirements will be satisfied. By Logic **E**, this is the same as proving that

$$(0 \le level \le 10) \rightarrow (W.\text{desc} \wedge M.spec \rightarrow R) . \qquad \text{(Eq. 21)}$$

Gathering together all the information, we must prove:

$$\begin{array}{l} W.d_0\text{: } (0 \le level \le 10) \\ W.d_1\text{: } level' = (level + in - out) \\ W.d_2\text{: } out = (level \ge 1|_0^1) \\ M.s_1\text{: } in = (level \le 1|_0^9) \\ M.s_2\text{: } (alarm \equiv level \le 1) \\ \hline R_1\text{: } (1 \le level' \le 9) \\ R_2\text{: } level' = (level \le 1|_{level-out}^9) \\ R_3\text{: } (alarm \equiv level \le 1) \end{array}$$

The proof follows from three lemmas. R_3 can be obtained directly from $M.s_2$ (using reflexivity of implication (3.71) $p \rightarrow p$), i.e.,

$$\text{Lemma1: } M.s_2 \rightarrow R_3 . \qquad \text{(Eq. 22)}$$

Next, we prove the more specific requirement R_2 first, in anticipation that it may also be useful in deriving R_1. In the proof of R_2, it seems worth starting with $W.d_1$ as it has the most precise information (it is an equality, not an inequality). The resulting calculation (see Fig. 8), which also uses the assumptions $W.d_2$ and $M.s_2$, yields:

$$\text{Lemma2: } W.d_2 \wedge M.s_1 \wedge W.d_1 \rightarrow R_2 . \qquad \text{(Eq. 24)}$$

The proof of Lemma2 is long (in fact, longer than we had hoped). The proof length is due to the need to do case analysis (see IF-transform in Fig. 8). It was precisely this case analysis that provided a counterexample to the naive specification (Eq. 18).

As we originally anticipated, R_1 can be derived from R_2 (see Fig. 9) to obtain

FIGURE 8. Calculational proof of Lemma2

$W.d_1$

= < definition of $W.d_1$ >

$level' = (level + in - out)$

= < **assumption** $W.d_2$ >

$level' = (level + in - (level \geq 1|_0^1))$

= < **assumption** $M.s_1$ >

$level' = (level + (level \leq 1|_0^9) - (level \geq 1|_0^1))$

= < IF-transform — but, leave "IF" in last conjunct to conform with final form>

$(level = 0) \rightarrow (level' = level + 9 - 0)$
$\wedge (level = 1) \rightarrow (level' = level + 9 - 1)$
$\wedge (level > 1) \rightarrow (level' = level + 0 - (level \geq 1|_0^1))$

= < Leibniz substitution 3.84(b) to first two conjuncts>

$(level = 0) \rightarrow (level' = 0 + 9 - 0)$
$\wedge (level = 1) \rightarrow (level' = 1 + 9 - 1)$
$\wedge (level > 1) \rightarrow (level' = level + 0 - (level \geq 1|_0^1))$

= < arithmetic simplification >

$(level = 0) \rightarrow (level' = 9)$
$\wedge (level = 1) \rightarrow (level' = 9)$
$\wedge (level > 1) \rightarrow (level' = level - (level \geq 1|_0^1))$

= < theorem of prop. logic: $((p \rightarrow r) \wedge (q \rightarrow r)) \equiv (p \vee q \rightarrow r)$, to first two conjuncts>

$(level \leq 1) \rightarrow (level' = 9)$
$\wedge (level > 1) \rightarrow (level' = level - (level \geq 1|_0^1))$

= < **assumption** $W.d_2$ to reinsert out >

$(level \leq 1) \rightarrow (level' = 9)$
$\wedge (level > 1) \rightarrow (level' = level - out)$

= < IF-transform >

$level' = (level \leq 1|_{level-out}^9)$

= < definition of R_2 >

R_2 .

The above proof is based on the assumptions $W.d_2$ and $M.s_1$. By EDT (see extended deduction theorem in the Appendix) we have thus established the lemma $(W.d_2 \wedge M.s_1) \rightarrow (W.d_1 \equiv R_2)$ from which it is simple to derive the lemma:

Lemma2: $(W.d_2 \wedge M.s_1 \wedge W.d_1) \rightarrow R_2$. (Eq. 23)

Lemma 3: $(W.d_0 \wedge W.d_2) \rightarrow (R_2 \rightarrow R_1)$. (Eq. 26)

Using the three lemmas, a quick calculational proof shows the validity of specification correctness $(\forall level: LEVEL | A.\mathrm{desc} \wedge M.spec \rightarrow R)$.

The cooling tank example can be checked automatically with the help of PVS (Fig. 10) The PVS descriptions of the real-world domain, requirements, and machine specification for the cooling tank are shown in the figure. The conjecture *system_correctness* (end of

FIGURE 9. Calculational proof of Lemma3

R_2

= < definition of R_2; IF-transform; **assumption** $W.d_2$ to replace out >

$[level \le 1 \wedge (level' = 9)]$

$\vee [level > 1 \wedge (level' = level - (level \ge 1|_0^1))]$

= < (10.14b) with $(level > 1) \to (level \ge 1)$ >

$((level \le 1) \wedge (level' = 9))$

$\vee [(level > 1) \wedge (level' = level - 1)]$

⇒ < arithmetic $(level' = 9) \to R_1$ and monotonicity MON (see appendix Sect. 6.1) >

$R_1 \vee [(level > 1) \wedge (level' = level - 1)]$

= < *true* is the identity of conjunction (3.39) >

$R_1 \vee [(level > 1) \wedge true \wedge (level' = level - 1)]$

= < **assumption** $W.d_0$: $(0 \le level \le 10)$ and theorem equivalence (TE) >

$R_1 \vee [(level > 1) \wedge (0 \le level \le 10) \wedge (level' = level - 1)]$

= < arithmetic theorem $(level > 1) \wedge (0 \le level \le 10) = (2 \le level \le 10)$ >

$R_1 \vee [(2 \le level \le 10) \wedge (level' = level - 1)]$

= < Leibniz substitution 3.84(a) with $level = level' + 1$ >

$R_1 \vee [(2 \le level' + 1 \le 10) \wedge (level' = level - 1)]$

⇒ < weakening theorem (3.76b) $p \wedge q \to p$ and MON >

$R_1 \vee (2 \le level' + 1 \le 10)$

= < arithmetic simplification>

$R_1 \vee (1 \le level' \le 9)$

= < definition of R_1 and theorem (3.26), i.e. idempotency of disjunction: $(p \vee p) = p$ >

R_1.

By EDT, we have established the theorem

$$\text{Lemma3: } (W.d_0 \wedge W.d_2) \to (R_2 \to R_1) \qquad \text{(Eq. 25)}$$

Fig. 10) is proved automatically when submitted to the PVS prover. The PVS file also shows an example of a sanity check to ensure that the outflow is correctly described.

4.2 Tools

There are currently a variety of tools available that contain expressive specification languages, theorem provers and model-checkers that will do large calculations automatically; such tools can be used to support the design method. We have shown the usefulness of PVS [20], but there are now a variety of tools available that have been used in selected industrial applications (Table 1).

The specification language of PVS is based on a typed higher-order logic. The base types include uninterpreted types that may be introduced by the user, and built-in types such as the booleans, integers, reals, as well as type-constructors that include functions, sets, tuples, records, enumerations, and recursively-defined abstract data types, such as lists and binary trees. PVS specifications are organized into parameterized theories that may contain assumptions, definitions, axioms, and theorems. PVS expressions provide the usual arithmetic and logical operators, function application, lambda abstraction, and quan-

FIGURE 10. Automated PVS proof of the cooling tank system

```
tank: THEORY
BEGIN
LEVEL: TYPE  = {x:nat | x  <= 10}

% Designations. We use "level_f" for the final value of "level"
level, level_f, inn, out: VAR LEVEL
alarm: VAR bool

% Description of the real-world domain
real_world_description(inn, out, level, level_f): bool =
      out =  (IF level >= 1 THEN 1 ELSE 0 ENDIF)
      AND
      (level_f = level + inn - out)

% The requirements document
requirement(level,level_f,out,alarm): bool =
      (1 <= level_f AND level_f <= 9)
      AND
      (level_f = (IF level <= 1 THEN 9 ELSE level-out ENDIF))
      AND
      (alarm = (level <= 1))

% The machine specification
machine_spec(level,inn,alarm): bool =
      inn = (IF level <= 1 THEN 9 ELSE 0 ENDIF)
      AND
      alarm = (level <= 1)

system_correctness: CONJECTURE
      real_world_description(inn,out,level,level_f)
      AND
      machine_spec(level,inn,alarm)
      IMPLIES
      requirement(level,level_f,out,alarm)

sanity_check: CONJECTURE
      real_world_description(inn,out,level,level_f)
      IMPLIES
      (out = 0 OR out = 1)

END tank
```

tifiers, within a natural syntax. An extensive prelude of built-in theories provides useful definitions and lemmas.

The description language Z is based on a typed version of ZF set theory [27]. It is perhaps the most widely used formal specification notation in industry, particularly in Europe. It has been harder to develop mechanized help for Z since it was not designed with automation in mind. Nevertheless, tools such as Z/Eves support the analysis of Z specifications by syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving [26].

PVS and Z do not provide explicit support for the transition from specifications to implementations. The Eiffel programming language does provide lightweight formal methods support, especially with its clean implementation of design-by-contract. It is an ideal tool for the development of programs from specifications. By contrast, the newer

Java language — for all its important features, such as type safety, automatic garbage collection, and web applets — does not even have the simple assert statements of C++. Although a certain amount of assert functionality can be implemented in a Java program [24], it does not match the Eiffel features for design-by-contract. This means that components in Java cannot be specified with the same degree of precision or ease as those in Eiffel.

The B-Method (with associated machine support from the B-Tool) uses a Z-like Abstract Machine Notation (AMN), and it supports development of specifications in AMN all the way down to executable programs [1]. Perhaps the most well known example using B is the development of the Paris Metro braking system software. In the Paris Metro, the choice was between reducing the timing between trains by increasing the assurance in the system as a whole, or building a new tunnel at vast cost.

Students can be introduced to the use of automated tools, such as PVS or the B-Tool, in the later stages of their undergraduate education, e.g., third or fourth year software engineering courses, and in particular after they have a thorough grounding in the calculational logic **E**. Without a grounding in logic, students will have difficulty understanding the proof steps that they are applying, and will certainly have complications in continuing proofs when difficulties or apparent dead-ends arrive.

4.3 Timed and Hybrid descriptions

In the cooling tank example we abstracted out time by restricting our attention to a single arbitrary cycle. This prevents us from describing liveness properties such as “eventually the tank will be filled to 9 units of water”. To describe such properties we can extend our logic with temporal operators so that we can assert conjectures such as: $\Box\Diamond(level = 9)$. The temporal formula $\Diamond p$ means eventually at some time after the initial state p must hold, and $\Box q$ means q must hold continually. Thus $\Box\Diamond p$ means that in every state of a computation there is always some future occurrence of p (see [16]).

Sometimes, even more specific timing information must be described. To express the property that the tank should always be filled to 9 units every 10 cycles (i.e. every 200 seconds) can be expressed as $\Box\Diamond_{\leq 200}(level = 9)$ in real-time temporal logic [19].

In some situations a hybrid approach must be followed in which there is a mixture of continuous and discrete mathematics. For example, in a more precise model of the outflow we might want to express the relationship between the tank outflow and the valve setting $v(t)$ as

$$\frac{d}{dt}out(t) = c_1 v(t) + c_2 level(t)$$

where $out(t)$ is the total amount drained from the tank up to time t, and $v(t)$ is the outflow valve setting as a function of time.

The StateTime [19], STeP [15] and Hytech tools [2] are examples of toolsets that can analyze and calculate properties of systems described with real-time temporal logic or hybrid descriptions using algorithmic and theorem proving techniques. These tools enable the designer to analyze concurrent and nondeterministic reactive programs.

Upper-year undergraduate, and introductory graduate-level courses, may best make use of tools for real-time systems. At York University, our fourth-year real-time systems course makes use of such tools. But the course also requires a grounding in mathematical methods, in particular, the calculational logic, which the students can use for hand calculations, for designing small systems or small components, and as a supplement to the automated tools when difficulties arise in proof.

5.0 Conclusion

Logic can be used throughout the software development life-cycle both as a design calculus and for documenting requirements, specifications, designs and programs. The use of logic provides both precision and the ability to predict software behaviour, thus providing the developer with a tool akin to that used in other Engineering disciplines. Learning the methods and tools of logic should be an important component in the education of software professionals.

Logic and logical calculation methods can and should be used right at the beginning of a Computer Science education. Here we summarize briefly a curriculum that makes use of calculational methods, from introductory undergraduate courses, through upper-year software engineering courses.

- The logic text by Gries and Schneider [7] can be used in two courses (each lasting a semester) in logic and discrete mathematics in the first and second years. This will provide the student with familiarity and comfort in logical calculation right from the beginning. This course will also help in future material such as understanding design-by-contract and theorem provers. The first-year mathematics programme for CS students at York University is now teaching such courses, based on Gries' text.
- The usual CS1 and CS2 courses can be taught in Eiffel stressing design-by-contract [13,17]. The trend currently is to use Java in the first year. This provides an opportunity for a text book for Java that will develop suitable design-by-contract constructs for Java [24]. Until such books, and assertional techniques, for Java appear, use of mathematical logic in CS1 and CS2 courses that use Java may occur by treating pre- and postconditions as comments or annotations. The table specification methods developed by Parnas [23] may also be of help for languages that do not have design-by-contract built in.
- A third year course in the use of tools such as PVS and the B-Tool can build on the material of the first few years. Such a course could use languages that support design-by-contract, such as Eiffel, in a software engineering project. PVS or the B-Tool could be used to formally derive programs from specifications (that would be eventually implemented in Eiffel). The calculational logic would be used as the foundation for understanding proofs and provers, and to do small calculations by hand. A comprehensive text on object-oriented specification, design, and programming, with emphasis on the production of quality software using BON/Eiffel is also available [17].
- A fourth year course can introduce the formal methods of reactive systems (e.g using STeP [15], SPIN [11] or SMV [3]). Suitable textbooks are available for each of these courses, but more need to be written, emphasizing the use of mathematical methods and calculation in design.

A variety of applications of formal methods to industrial systems have been reported as shown in the Table 1. These applications can be used for case studies in more advanced classes. Students should also apply their skills to case studies such as that of the Therac-25 radiotherapy machines [14] and the Ariane 5 heavy launcher [13], which illustrate the need for professional standards in all aspects of design.

TABLE 1. Some examples of the use of tools in industrial practice

Tool	System	Application
PVS	hardware	AAMP5 Microprocessor[a].
SMV	hardware	HP Summit Bus.
Spin	communication protocol	Ethernet collision avoidance.
	software	Requirement analysis of Space Shuttle GPS Change Requests
Z/Eves	communication protocol	A Micro-flow modulator that controls flow of information from a private system to a public system.
PVS	hardware/software	IEEE-compliant subtractive division algorithm.
B-tool	software	Paris metro.

a. PVS was used to specify and verify the Rockwell AAMP5 microprocessor having 500,000 transistors; 108 out of the 209 instructions of the microcode were described. The exercise found one error that was a missing requirement. Also found, was a coding error (improperly sized stack) that would not have been detected in ordinary assurance testing [28].

We should not underestimate the effect that education can have in practice. “Spice” is a general purpose electronic circuit simulation program that was designed by Donald Pederson in the early 1970s at the University of Berkeley. Circuit response is determined by solving Kirchoff’s laws for the nodes of a circuit. During the early 1970s, Berkeley was graduating over a 100 students a year who were accustomed to using Spice. They started jobs in industry and loaded Spice on whatever computers they had available. Spice quickly caught on with their co-workers, and by 1975 it was in widespread use. Spice has been used to analyze critical analog circuits in virtually every IC designed in the United States in recent years [25].

In software development, the practitioner has to sub-ordinate everything to the overriding imperative to deliver an adequate product on time and within budget. This means that the theory and tools we do teach must be useful and as simple as possible. Logic **E**, design-by-contract, Eiffel and PVS embody useful theory and tools that can be taught and used now, and that will contribute to professional engineering standards for software design and documentation.

6.0 Appendix on Logic E

6.1 Derived Inference Rules

The fact that conjunction is monotonic in its first argument is expressed by the theorem:

(4.2) Monotonicity of conjunction: $(p \rightarrow q) \rightarrow (p \wedge r \rightarrow q \wedge r)$.

Modus Ponens (**MP**) $\dfrac{p,\ p \to q}{q}$	Theorem Equivalence (**TE**) $\dfrac{p,\ q}{p \equiv q}$
Case Replacement (**CR**) $\dfrac{q_1 \vee q_2 \vee q_3}{p \equiv \begin{pmatrix} (q_1 \to p) \\ \wedge (q_2 \to p) \\ \wedge (q_3 \to p) \end{pmatrix}}$	Monotonicity (**MON**) $\dfrac{p \to q}{E[z{:=}p] \to E[z{:=}q]}$ provided (a) z occurs exactly once in predicate E, and (b) z does not occur in an operand of an equivalence or inequivalence, a negation or the antecedent of an implication. (Conjunction and disjunction are monotonic in both operands, and implication is monotonic in its consequent.)

Conjunction and disjunction are monotonic in both arguments, and implication is monotonic in its second argument (its consequent). The derived rule MON provided above, can be extended to include predicates with quantifiers.

Extended Deduction Theorem (EDT): Suppose we can prove Q provided we add the (temporary) axioms $P_1, P_2, \ldots, P_n$ to Logic **E** with the variables of the P_i considered to be constants. Then $P_1 \wedge P_2 \wedge \ldots \wedge P_n \to Q$ is a theorem.
(In the course of the proof of Q, Substitution may not be applied to any temporary axiom or to any temporary theorem that is derived in the course of the proof, if the variable being substituted for appears in one of the original assumptions.)

6.2 Conditional expressions

We denote the conditional expression $b|_{e_2}^{e_1}$ by *IF* where *IF* is a function with three parameters, i.e.

$$IF\colon BOOLEAN \times T \times T \to T$$

Hence, $type(b) = BOOLEAN$ and $type(e_1) = type(e_2) = T$ for some type T. It also follows that $type(IF) = T$. We assume that any use of *IF* satisfies these typing constraints. The two axioms for reasoning about conditional expressions are [7]:

$$\text{(10.9)}\ b \to \left(b|_{e_2}^{e_1} = e_1\right) \qquad \text{(10.10)}\ \neg b \to \left(b|_{e_2}^{e_1} = e_2\right)$$

6.2.1 Theorems of conditional expressions derived from the axioms:

(10.11) $\left(true|_{e_2}^{e_1}\right) = e_1$	(10.12) $\left(false|_{e_2}^{e_1}\right) = e_2$
(10.13a): $\left(b|_{e_2}^{e_1}\right) = ((b \to e_1) \wedge (\neg b \to e_2))$ provided $type(e_1) = type(e_2) = BOOLEAN$	(10.13b) $\left(b|_{e_2}^{e_1}\right) = ((b \wedge e_1) \vee (\neg b \wedge e_2))$ provided $type(e_1) = type(e_2) = BOOLEAN$

(10.14a): $\left(p \to E\left[z{:=}b|_{e_2}^{e_1}\right]\right) \equiv (p \to E[z{:=}e_1])$ provided that $p \to b$ is a theorem.

(10.14b): $\left(p \wedge E\left[z{:=}b\Big|_{e_2}^{e_1}\right]\right) \equiv (p \wedge E[z{:=}e_1])$ provided that $p \rightarrow b$ is a theorem.

(10.14c): $\left(p \rightarrow E\left[z{:=}b\Big|_{e_2}^{e_1}\right]\right) \equiv (p \rightarrow E[z{:=}e_2])$ provided that $p \rightarrow \neg b$ is a theorem.

(10.14d): $\left(p \wedge E\left[z{:=}b\Big|_{e_2}^{e_1}\right]\right) \equiv (p \wedge E[z{:=}e_2])$ provided that $p \rightarrow \neg b$ is a theorem.

Proof of theorem (10.11)

$true \rightarrow \left(\left(true\Big|_{e_2}^{e_1}\right) = e_1\right)$ --- (10.9)$[b{:=}true]$

= <left identity of implication (3.73) (i.e. $(true \rightarrow p) \equiv p$) >

$\left(\left(true\Big|_{e_2}^{e_1}\right) = e_1\right)$

Hence, by equanimity, (10.11) is a theorem.

6.2.2 Proof in Logic E for theorem (10.14a)

By the derived rule Modus Ponens (MP), it is sufficient to prove that

$$(p \rightarrow b) \rightarrow \left[\left(p \rightarrow E\left[z{:=}b\Big|_{e_2}^{e_1}\right]\right) \equiv (p \rightarrow E[z{:=}e_1])\right]$$

is a theorem. Here is the proof.

$(p \rightarrow b) \rightarrow \left[\left(p \rightarrow E\left[z{:=}b\Big|_{e_2}^{e_1}\right]\right) \equiv (p \rightarrow E[z{:=}e_1])\right]$

= < distributing implication over equales (3.63) >

$(p \rightarrow b) \rightarrow \left[p \rightarrow \left(E\left[z{:=}b\Big|_{e_2}^{e_1}\right] \equiv E[z{:=}e_1]\right)\right]$

= < shunting (3.65) >

$(p \wedge (p \rightarrow b)) \rightarrow \left(E\left[z{:=}b\Big|_{e_2}^{e_1}\right] \equiv E[z{:=}e_1]\right)$

= < (3.66) to antecedent >

$(p \wedge b) \rightarrow \left(E\left[z{:=}b\Big|_{e_2}^{e_1}\right] \equiv E[z{:=}e_1]\right)$

= <replace *b* by *true* in the *consequent* because *b* is in the *antecedent* (3.85b) >

$(p \wedge b) \rightarrow \left(E\left[z{:=}true\Big|_{e_2}^{e_1}\right] \equiv E[z{:=}e_1]\right)$

= < axiom (10.11) for conditional expressions >

$(p \wedge b) \rightarrow (E[z{:=}e_1] \equiv E[z{:=}e_1])$

= < identity of equales (3.3), (3.4) and derived rule TE >

$(p \wedge b) \rightarrow true$

= < right zero of implication (3.72) >

$true$ -- (3.3)

6.2.3 "IF-transform" reasoning uses case replacement (CR) and (10.14)

Consider a variable x with $type(x) = NATURAL$. It then follows that

$$(x = 0) \vee (x = 1) \vee (x > 1) \qquad \text{(Eq. 27)}$$

is a theorem. We may then use the derived rule CR, (10.14a) and (10.14c) to show that the following is a theorem:

$$\textbf{IF-transform}: [x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z)] \equiv \begin{bmatrix} (x = 0) \rightarrow (x' = x + 9 - z) \\ \wedge (x = 1) \rightarrow (x' = x + 9 - 1) \\ \wedge (x > 1) \rightarrow (x' = x + y - 1) \end{bmatrix}$$

Here is the proof.

$[x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z)]$

= < case replacement (CR) with (Eq. 27) >

$(x = 0) \rightarrow (x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z))$

$\wedge (x = 1) \rightarrow (x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z))$

$\wedge (x > 1) \rightarrow (x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z))$

= < (10.14a) with $(x = 0) \rightarrow (x \le 1)$ to first conjunct >

$(x = 0) \rightarrow (x' = x + 9 - (x \ge 1|^1_z))$

$\wedge (x = 1) \rightarrow (x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z))$

$\wedge (x > 1) \rightarrow (x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z))$

= < (10.14c) with $(x = 0) \rightarrow \neg(x \ge 1)$ to first conjunct >

$(x = 0) \rightarrow (x' = x + 9 - z)$

$\wedge (x = 1) \rightarrow (x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z))$

$\wedge (x > 1) \rightarrow (x' = x + (x \le 1|^9_y) - (x \ge 1|^1_z))$

= < applying the same type of reasoning to the 2nd and 3rd conjunct >

$(x = 0) \rightarrow (x' = x + 9 - z)$

$\wedge (x = 1) \rightarrow (x' = x + 9 - 1)$

$\wedge (x > 1) \rightarrow (x' = x + y - 1)$

7.0 References

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

[2] Alur, R., T.A. Henzinger, and P.-H. Ho. "Automatic Symbolic Verification of Embedded Systems." *IEEE Transactions on Software Engineering*, 22(3): 181-201, 1996.

[3] Burch, J.R., E.M. Clarke, K.L. MacMillan, D.L. Dill, and L.J. Hwang. "Symbolic Model Checking: 10^20 States and Beyond." *Information and Computation*, 98(2): 142-170, 1992.

[4] Dean, C.N. and M.G. Hinchey, eds. *Teaching and Learning Formal Methods*. Vol. London: Academic Press, 1996.

[5] Glass, R.L. "The Software Research Crisis." *IEEE Software*, 11(6): 42-47, 1994.

[6] Gries, D. *The Science of Programming*. Springer-Verlag, 1985.

[7] Gries, D. and F.B. Schneider. *A Logical Approach to Discrete Math*. Springer Verlag, 1993.

[8] Hall, A. "Seven Myths of Formal Methods." *IEEE Software*, 11-19, 1990 (September).

[9] Hehner, E.C.R. *A Practical Theory of Programming*. Springer Verlag, New York, 1993.

[10] Hinchey, M. and J. Bowen. *Applications of formal methods*. Prentice Hall, 1995.

[11] Holzmann, G. "The Model Checker Spin." *IEEE Trans. on Software Engineering*, 23(5): 279-295, 1997.

[12] Jackson, M. *Software Requirements & Specifications*. Addison-Wesley, 1995.

[13] Jezequel, J.-M. and B. Meyer. "Design by Contract: the Lessons of the Ariane." *IEEE Computer*, 30(1): 129-130, 1997.

[14] Leveson, N.G. and C.S. Turner. "An Investigation of the Therac-25 Accidents." *Computer*, 26(7): 18-41, 1993.

[15] Manna, Z. "STeP: The Stanford Temporal Prover." Dep. of Computer Science, Stanford University. STAN-CS-TR-94-1518, 1994.

[16] Manna, Z. and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[17] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[18] Morgan, C. *Programming from Specifications*. International Series in Computer Science, ed. Prentice Hall, 1994.

[19] Ostroff, J.S. "A Visual Toolset for the Design of Real-Time Discrete Event Systems." *IEEE Trans. on Control Systems Technology,* 5(3): 320-337, 1997.

[20] Owre, S., J. Rushby, N. Shankar, and F.v. Henke. "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS." *IEEE Trans. on Software Engineering*, 21(2): 107-125, 1995.

[21] Parnas, D.L. "Mathematical Descriptions and Specification of Software." In *Proceedings of IFIP World Congress 1994*, Volume I August 1994, 354-359, 1994.

[22] Parnas, D.L. and P.C. Clements. "A Rational Design Process: How and Why to Fake it." *IEEE Trans. on Software Engineering*, SE-12(2): 251-257, 1986.

[23] Parnas, D.L., J. Madey, and M. Iglewski. "Precise Documentation of Well-Structured Programs." *IEEE Transactions on Software Engineering*, 20(12): 948-976, 1994.

[24] Payne, J.E., M.A. Schatz, and M.N. Schmid. "Implemeting assertions for Java." *Dr. Dobb's Journal*, 281): 40-44, 1998.

[25] Perry, T.S. "Donald O. Pederson." *IEEE Spectrum*, 35(6): 22-27, 1998.

[26] Saaltink, M. " Proceedings ZUM'97: The Z Formal Specification Notation (10th International Conference of Z Users)." In Reading, UK (April 1997), Springer-Verlag, Lecture Notes in Computer Science 1212, 72-85, 1997.

[27] Spivey, J.M. *The Z Notation: A Reference Manual*. Prentice-Hall, Englewood Cliffs, N.J., 1989.

[28] Srivas, M. and S.P. Miller. "Applying Formal Verification to a Commercial Microprocessor." In *Proceedings of the 1995 IFIP International Conference on Computer Hardware Description Languages*, Chiba, Japan, 493-502, 1995.