The logic of software design

J.S.Ostroff and R.F.Paige

Abstract: The authors provide an overview of how logic can be used throughout the software development cycle, and discuss what methods can be introduced in the computer science curriculum to support software development. To see how logic is useful throughout the cycle, they present the WRSM reference model, and illustrate it with simple motivating examples. Reasoning is performed in Logic E, and PVS is used to illustrate automated proofs.

1 Introduction

Logic is the glue that binds together reasoning in many domains. In software development, the argument for the use of formal methods based on logic is that it provides precise documentation, allows us to predict software behaviour, facilitates design and early detection of faults, and produces correct implementations. Although there have been some notable successes, formal methods have not been adopted in practice. The arguments over the usefulness of formal methods will not be recounted here [1–6] 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. It is the educational programme that we address in this paper.

We provide an overview of how logic can be used throughout the software development cycle, and discuss what methods can be introduced in the computer science curriculum to support software development.

2 Software engineering

The ISO network reference model divides communication protocols into seven layers. Although the ISO reference model is informal and does not correspond perfectly to protocols in widespread use, it is widely used to describe actual network architectures. Similarly, a reference model for software development is useful for describing the software development life-cycle of requirements, design and implementation. Such a reference model should provide guidance to the main artifacts and deliverables; provide precise definitions of, and distinctions between, important artifacts such as requirements and specifications; and provide a rational software development process.

Two such reference models have been developed where the description of the model is itself in predicate logic. The two models are the Functional Documentation Model [7] of Parnas and Madey, and the WRSPM model [8] of Gunter, Jackson and Zave. The advantage of using logic in the description of such reference models is that logic provides the appropriate proof obligations that must be met to ensure that the final program correctly implements the requirements. Knowledge of these proof obligations makes sense just as much for natural-language documentation and informal testing, as it does for formal specifications that use theorem provers and model checkers. Most applications will benefit significantly just from the clarity of knowing what the objective of a component of the model should be, even without formalization, let alone machine-assisted proof. [8]

We present one such reference model which we call the WRSM reference model. The notation of WRSM is derived from [8], but for refinement of specifications to programs we use the timed predicative calculus of Hehner [9]. The WRSM reference model is primarily based on making a clear distinction between the environment (the 'plant') and the programs that interact with the environment. This distinction is classic in systems engineering, and it has a profound effect on the analysis of software as well. An early attempt that applies this distinction to software may be found in chapter one of a monograph by the first author [10].

2.1 The WRSM reference model

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. We let M denote the software machine consisting of the hardware, operating system and programs.

The machine M must ultimately be installed in and interact with the external world W. It is the effect of machine M on world W which is of most interest to the customers of the machine. The phenomena (e.g. states or events) of the external world determine the customer's *requirements*. Requirements R are about the phenomena of the external world W, and not about the phenomena of the machine M. The machine must try to ensure that the requirements are satisfied by manipulating the shared phenomena at the interface of W and M. We can consider the various phenomena with the help of Fig. 1 (see [11]), in which $W \cap M$ is the set of all shared phenomena.

The traditional development process, from requirements to an implemented program, is a way of bridging the gap

IEE Proceedings online no. 20000681

DOI: 10.1049/ip-sen:20000681

Paper first received 1st February and in revised form 6th July 2000

The authors are with the Department Of Computer Science, York University, 4700 Keele Street, Toronto, Ontario, Canada M3J 1P3

E-mail: {jonathan, paige}@cs.yorku.ca

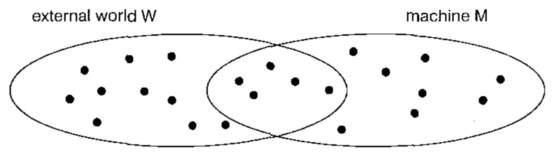

Fig. 1 *The phenomena of the external world W and the machine M*

between the phenomena of W and those of M. A rational development process, where each step follows on from the previous ones and everything is done in the most elegant and economic order, does not really exist for complex systems; nevertheless, we can fake it [12] . We can try to follow an established procedure as closely as possible, and when we finally have our solution (achieved as usual through numerous departures from the ideal process), we can produce the documentation that would have resulted if we had followed the ideal process. Apart from providing clarity of goals, a rational process will guide us when we become overwhelmed by the complexity of the task. Our WRSM reference model will provide a rational software development process. It works as follows.

1. Elicit and document the *requirements* R in terms of the phenomena of W.
2. From R, expressed in terms of W, derive a *specification* S of the machine, expressed in terms of the shared phenomena $W \cap M$. Customers should readily understand the requirements expressed as they are in terms of familiar world phenomena. By contrast, the specification may not make as much obvious sense to the customer. This is because a specification is derived from the requirements by a sequence of steps in answer to the question: what behaviour at the *shared interface* would produce the requirement effects in the *world at large*. Specifications must be free of implementation bias and must therefore not refer to irrelevant machine detail (this is called 'information hiding').
3. From the specification S derive the machine M. The phenomena of the machine include those phenomena shared with the world at the interface, as well as the hidden internal phenomena of the machine.

We now consider W, R, S and M to be double-state predicates describing the various phenomena. A *double-state* predicate is a predicate in the prestate σ and the poststate σ' (see Section 3). To justify that a program M satisfies its requirement R, we reason as follows.

4. Argue that if the specification S is satisfied, then so is the requirement, i.e.

$$\textbf{specification correctness} : \forall \sigma, \sigma' \bullet W \wedge S \rightarrow R \qquad (1)$$

To make the argument work, we may need to appeal to domain knowledge W that provides presumed environment facts. We also require that there be at least one prestate of $W \wedge M$ with a satisfying post-state, i.e. $\exists \sigma, \sigma' \bullet W \wedge S$, which guarantees that (eqn. 1) is not vacuous. We refer the reader to [8] for a discussion of additional types of consistency constraints.

5. Then, argue that the machine M refines the specification S, i.e.

$$\textbf{implementation correctness} : \forall \sigma, \sigma' \bullet M \rightarrow S \qquad (2)$$

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

$$\textbf{system correctness} : \forall \sigma, \sigma' \bullet W \wedge M \rightarrow R \qquad (3)$$

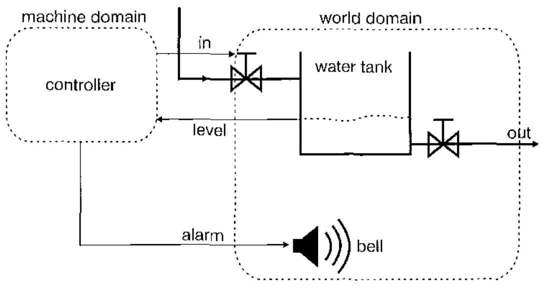

Fig. 2 *Rough sketch of the cooling tank, identifying the phenomena of interest*

7. Implementation correctness (eqn. 2) describes program refinement, a process in which we gradually transform specification S into implementation M. To make refinement rigorous, we must check that the specification is implementable. S is *implementable* if [9]

$$\forall \sigma \bullet (\exists \sigma' \bullet S) \qquad (4)$$

We have argued in [13, 14] that the Hehner notion of refinement is simpler than others such as Z, as well as being applicable to object-oriented and real-time systems.
8. It may be necessary to go through a sequence of successive refinements before achieving the final program M. For example, in refinement by parts, we may decide to break specification S into two parts S_1 and S_2, and refine each one independently to M_1 and M_2, respectively. If we can prove that

$$\begin{aligned} S_1 \wedge S_2 &\rightarrow S \\ M_1 &\rightarrow S_1 \\ M_2 &\rightarrow S_2 \end{aligned} \qquad (5)$$

are theorems, then $M_1 \wedge M_2 \rightarrow S$ is also a theorem and the parts implement the whole. Other laws such as refinement by steps, and refinement by cases can also be provided. The simplicity of these laws derives from the simplicity of the basic formula for refinement (2). Specifications can be implemented as classes in an OO framework [14].

The above eight points together with the constraints of eqns. 1–5, define the artifacts W, R, S, and M of the WRSM reference model.

As an example, consider the cooling tank in Fig. 2 with variables *in, out, level* of type *LEVEL* (1 to 10 units of volume) and *alarm* of type BOOLEAN. The variables *in, level* and *alarm* are shared phenomena and hence a specification of the controller may refer to them. The cooling tank (W) controls the *level* while the controller (M) controls *in* and *alarm*. The variable *out* is a hidden world variable and hence is not shared with S. Domain knowledge W relates the behaviour of *out* with other parts of the system (see Section 4). The hidden internal variables of the controller are not shown.

3 Using logic for descriptions and calculations

Like other engineering students, software engineering students should have a working knowledge of classical mathematics such as calculus, linear algebra and probability theory. 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. What key skills must software engineers master to use logic effectively? We believe that the critical skills are:

- the ability to make informal descriptions precise
- the ability to calculate properties of products (by proving theorems)
- the ability to show that a putative statement is not valid by providing a counterexample
- the ability to use theorem provers and model-checkers as well as produce manual proofs

We illustrate these skills by way of a simple example, and then argue for the use of Logic E as the right framework within which to learn these skills. 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 gets only five tries at entering the password; if the user entry matches the stored password, 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.'

In Fig. 3, we use an Eiffel class [15] to formally specify the password management module. Eiffel is an example of an available object-oriented language and method, based on the principles of seamlessness, reversibility and software *contracting*. In this approach, a class is an abstract data type whose functions (*features*) are specified with predicates (assertions). The prerequisite and resulting behaviour of a feature is specified by pre- and post-conditions, and class consistency is specified by a class invariant. These specifications form a contract between the supplier and clients of the class. A software system is viewed as a network of co-operating clients and suppliers whose exchange of requests and services are precisely defined through decentralised contracts.

Class PASSWORD_MANAGEMENT starts by defining the various attributes (state) of the module. The behaviour of routine *verify_user* is specified by a precondition (the *require* clause) and a postcondition (the *ensure* clause). The precondition describes the set of all initial states

```
class PASSWORD_MANAGEMENT feature
    -- attributes, i.e. the state space
    alarm : BOOLEAN                        -- signal illegal entry
    operate : BOOLEAN                      -- user may operate PDA
    p1 :  PASSWORD                         -- the password
    i : INTEGER                            -- number of password tries

    make(p2:PASSWORD)                      -- initialization routine
        ensure $\neg alarm \wedge \neg operate \wedge i = 0 \wedge p1 = p2$

    verify_user(p2: PASSWORD)     -- routine to verify password p2
        require $\neg alarm \wedge \neg operate$
        ensure $(g_1 \rightarrow e_1) \wedge (g_2 \rightarrow e_2) \wedge (g_3 \rightarrow e_3)$
             $g_1 \cong \textbf{old}\ i < 6 \wedge \textbf{old}\ p1 = p2$
             $g_2 \cong \textbf{old}\ i < 6 \wedge \textbf{old}\ p1 \neq p2$
             $g_3 \cong \textbf{old}\ i \geq 6 \wedge \textbf{old}\ p1 \neq p2$
    – where  $e_1 \cong i = 0 \wedge operate \wedge \neg alarm \wedge p1 = \textbf{old}\ p1$
             $e_2 \cong i = \textbf{old}\ i + 1 \wedge \neg operate \wedge \neg alarm \wedge p1 = \textbf{old}\ p1$
             $e_3 \cong i = 0 \wedge \neg operate \wedge alarm \wedge p1 = \textbf{old}\ p1$

    invariant $i \geq 0 \wedge (operate \rightarrow \neg alarm)$
                                        –all routines preserve the invariant
end
```

Fig. 3 *Eiffel specification of the password management module*

(prestates) for which the routine is guaranteed to produce a final state (a poststate) that satisfies the postcondition; if the precondition is not satisfied, then nothing is guaranteed.

In postconditions, the notation (***old*** *expression*) denotes the value of *expression* in the prestate. Thus $i = \textbf{old}\ i + 1$ 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 specified as $i \geq 0 \wedge operate \rightarrow \neg alarm$ must be preserved by each routine.

The class as shown has no implementation; it is an abstract class. Even so, it can be compiled as is (with its contracts) by the Eiffel compiler. The compiler will also ensure type correctness. The class contract is a precise specification of the informal description of the password module. This precision provides significant benefits. First, the contract provides precise self-documentation. Second, the contract provides a consistent error handling mechanism, violations of which can be monitored at run-time. Third, the formal specification of the class can now be used to calculate properties of the class. To illustrate the last point, we start by asking if an appropriate response is defined for every legal input to the feature *verify_user*.

3.1 Input coverage conjecture 1 – is every input handled

The postcondition of *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). Each consequent e_i fully specifies the poststate. Therefore, to show input coverage, it is sufficient to show the validity of

$$\neg(\textbf{old}\ alarm) \wedge \neg(\textbf{old}\ operate) \rightarrow (g_1 \vee g_2 \vee g_3). \quad (6)$$

Is (eqn. 6) a theorem? Unfortunately, not, because the state described by the observation $\neg(\textbf{old}\ alarm) \wedge \neg\ (\textbf{old}\ operate) \wedge (\textbf{old}\ i = 6) \wedge (\textbf{old}\ p1 = p2)$ is a counter-example that makes the disjunction of the guards false for at least one legal input; for this input the poststate of *verify_user* may then be anything including unintended violations such as setting both *operate* and *alarm* to true.

The above example clearly indicates that the specifier needs to know how to challenge putative specifications with counterexamples. In this case, the counterexample also provides guidance to fix the problem. The counterexample shows that the current contract allows six attempts at the password rather than the required five, and no more. This suggests that the guard g_3 of the *verify_user* routine be redefined to $g_3 \cong \textbf{old}\ i \geq 6$.

With this new definition of g_3 we can prove that (eqn. 6) is a theorem by using the calculational Logic E [6]. We assume the antecedent and prove under this assumption that the consequent is a theorem. The proof, shown in Fig. 4, transforms the consequent $g_1 \vee g_2 \vee g_3$ into a known theorem. [Note 1]

The proof in Fig. 4 illustrates the proof step *Leibniz*. In a Leibniz proof step, we substitute equals for equals by arguing as follows:

$E[z := P]$
$=$ < a Leibniz step: replace every occurrence of P in expression E by Q >
$E[z := Q]$

Note 1: In this particular case, the assumption (antecedent) was not required for the proof.

Assume : $\neg(\textbf{\textit{old}}\ alarm) \wedge \neg(\textbf{\textit{old}}\ operate)$.

$$
\begin{array}{ll}
 & g_1 \vee g_2 \vee g_3 \\
= & \quad < \text{definitions of } g_1, g_2 > \\
 & (\textbf{\textit{old}}\ i < 6 \wedge \textbf{\textit{old}}\ p1 = p2) \vee (old\ i < 6 \wedge \neg(\textbf{\textit{old}}\ p1 = p2)) \vee g_3 \\
= & \quad < \text{distributivity of conjunction over disjunction 3.46} > \\
 & (\textbf{\textit{old}}\ i < 6 \wedge (\textbf{\textit{old}}\ p1 = p2 \vee \neg(\textbf{\textit{old}}\ p1 = p2)) \vee g_3 \\
= & \quad < \text{excluded middle 3.28 makes} \\
 & \qquad\qquad (\textbf{\textit{old}}\ p1 = p2 \vee \neg(\textbf{\textit{old}}\ p1 = p2)) \equiv true > \\
 & (\textbf{\textit{old}}\ i < 6 \wedge true) \vee g_3 \\
= & \quad < \text{identity of conjunction 3.39; definition of } g_3 > \\
 & \textbf{\textit{old}}\ i < 6 \wedge \textbf{\textit{old}}\ i \geq 6 \\
= & \quad < \text{arithmetic} : (\textbf{\textit{old}}\ i < 6 \wedge \textbf{\textit{old}}\ i \geq 6) \equiv true > \\
 & true
\end{array}
$$

Fig. 4 *Proof of input coverage (Conjecture 1)*

In the second step of the proof in Fig. 4,

E is the expression $z \vee g_3$,
P is $(\textbf{\textit{old}}\ i < 6 \wedge \textbf{\textit{old}}\ p1 = p2) \vee (old\ i < 6 \wedge \neg\ (\textbf{\textit{old}}\ p1 = p2))$, and
Q is $\textbf{\textit{old}}\ i < 6 \wedge (\textbf{\textit{old}}\ p1 = p2 \vee \neg(\textbf{\textit{old}}\ p1 = p2)$.

In the justification, we must state which theorem makes $P = Q$ valid (in this case, the distributivity of conjunction which is theorem 3.46 in [16]). An alternative way of writing the Leibniz proof step is shown below to the left; to the right we also provide the other main type of proof step called *Monotonicity*.

$$
\textbf{Leibniz :}\ \frac{P = Q}{E[z := P] = E[z := Q]}
$$

$$
\textbf{Monotonicity :}\ \frac{P \rightarrow Q}{E[z := P] \rightarrow E[z := Q]}
$$

Leibniz applies under all circumstances. The Monotonicity proof step has a *proviso* as explained in [16, 17]. At each monotonicity step we must check the proviso and state which theorem makes $P \rightarrow Q$ valid. The mutual transitivity of implication and boolean equality means that we can mix Leibniz and Monotonicity steps in the proof of an implication.

In addition to brevity, Logic E is practical because it comes with a toolbox of useful theorems [16] for predicate calculus and a general theory of quantification. The text uses the logic to develop a variety of discrete mathematical theories such as sets, functions, relations, induction, lists, recurrence relations, algebra and combinatorics. Alternative approaches such as natural deduction systems are unwieldy outside the area of strict logic. The granularity of a proof step in Logic E is adjustable; the hints at each step can be sufficiently precise to allow the step to be rigorously checked if necessary, while allowing the proof writer the option of adjusting the size of the step so as to keep the proof short. Thus, a concise version of the proof in Fig. 4 is

$$
\begin{array}{ll}
 & g_1 \vee g_2 \vee g_3 \\
= & < \text{propositional logic and definitions of } g_i > \\
 & \textbf{\textit{old}}\ i < 6 \vee \textbf{\textit{old}}i \geq 6 \\
= & < \text{arithmetic} > \\
 & true
\end{array}
$$

The software engineer will also want to make use of theorem provers to do routine calculations. The use of theorem provers presupposes the type of knowledge developed by familiarity with Logic E. The following generalisation of the input coverage conjecture illustrates the use of theorem provers such as PVS [18].

```
password: THEORY
begin
passwordtype: TYPE

% attributes and routine argument p2
alarm, old.alarm, operate, old_operate: VAR Bool
i, old_i; VAR nat
p1, p2, old_p1: VAR passwordtype

% double-state 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))

% Specification Implementability Conjecture
implementability : 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
% PVS returns Q.E.D.
end password
```

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

3.2 Conjecture 2 – implementability conjecture

In Conjecture 1, we were able to do a simple check for implementability of routine *verify_user* because its postcondition was in the special guarded format. For general Eiffel specifications, the pre- and poststates are denoted by $\textbf{\textit{old}}\ \sigma$ and σ, respectively. Thus,

$$
\begin{array}{l}
\text{Eiffel specification } spec \text{ is implementable iff} \\
\quad \forall \textbf{\textit{old}}\sigma \bullet (\exists \sigma \bullet spec).
\end{array} \qquad (7)
$$

To determine if an Eiffel specification of a routine is implementable, we must provide a formal semantics for Eiffel contracts in our logic. A formal semantics was provided in [14] as follows: an Eiffel routine with precondition P and postcondition Q corresponds to the double-state Logic E predicate $spec$ defined by $spec \cong (\textbf{\textit{old}}\ P) \rightarrow Q$. The predicate $spec$ asserts that if P holds in the prestate, then the routine terminates with Q true; otherwise any behaviour including non-termination is acceptable. The state σ for a routine in a class consists of the attributes of the class as well as the arguments of the specified routine. The double-state specification for routine *verify_user* is

$$
spec \cong (\neg \textbf{\textit{old}}\ alarm \wedge \neg \textbf{\textit{old}}\ operate) \rightarrow
$$
$$
(g_1 \rightarrow e_1 \wedge g_2 \rightarrow e_2 \wedge g_3 \rightarrow e_3) \qquad (8)
$$

where the g_i and e_i are defined as before. Conjecture 2 states that $spec$ as defined in eqn. 8 is implementable.

The proof of the conjecture can be done in Logic E, but we will do it using the PVS theorem prover as shown in Fig. 5. PVS proved Conjecture 2 with some interaction from the user, which involved suggesting existential instantiation three times.

4 A simple case study – cooling tank

In this section, we present a small case study that will illustrate the use of logical methods and tools in requirements and specifications. For the case study we need a theory for conditional expressions such as (**if** b **then** e_1 **else** e_2) where b is of type Boolean and e_1,e_2 are two expressions of the same type. For conciseness we use the abbreviation

$$b|_{e_2}^{e_1} \qquad (9)$$

As an illustration of the utility of Logic E, we refer the reader to the technical report [19], wherein a number of powerful theorems for conditional expressions are listed and proved from their basic axioms, e.g.

$p \to E\left[z := b|_{e_2}^{e_1}\right] \equiv p \to E[z := e_1]$ is a theorem provided $p \to b$ is a theorem. (10)

Now consider the cooling tank, presented earlier in Fig. 2. The following are the informal requirements [20].

'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 (every 20 s). Every cycle, one unit of water flows out. 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.

Routine *set_alarm* raises *alarm* flag if the tank level goes below 1 unit. Routine *fill_tank* 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 program in Fig. 6 is wrong (as we shall see later), we have also not followed the recommended design method presented earlier (Section 2). The rough sketch of Fig. 2 illustrates the world phenomena including hidden variables such as *out* which is inaccessible to the machine, as well as the shared phenomena *in, level*, and *alarm*.

Having identified the phenomena of interest, the next step is to write the requirements. We assume that the machine will read sensor *level* 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

```
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
```

Fig. 6 *Faulty code for the cooling tank example*

next cycle 20 s 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 $R \cong R_1 \wedge R_2 \wedge R_3$

$$\text{where:} \begin{cases} R_1 : 1 \le level' \le 9 \\ R_2 : level' = (level \le 1)|_{level-out}^{9} \\ R_3 : alarm' \equiv level \le 1 \end{cases} \qquad (11)$$

The requirements may refer to any world phenomena including the hidden variable *out*. The initial values at the beginning of a cycle are designated by *level, alarm* and *out* respectively, and their final values at the end of the cycle are given by *level', alarm'* and *out'*. The requirement states that the final value of the water level must be between the stated bounds of one and nine, the tank must be appropriately filled (at the end of the cycle) if it goes low (at the beginning of the cycle), and the alarm bell must be sounded if the level is low.

The next step in the recommended design method is to describe the properties W presumed to characterize the external world. In this case we have $W \cong W_0 \wedge W_1 \wedge W_2$ where

$$\begin{cases} W_0 : 0 \le level \le 10 \\ W_1 : level' = level + in - out \\ W_2 : out = (level \ge 1)|_0^1 \end{cases} \qquad (12)$$

W_0 is a property of the cooling tank – it cannot hold more than 10 units of water. W_1 is derived from a physical law that says the flow at the end of a cycle is what the original level was, adjusted for in-flows and outflows. W_2 asserts that the outflow at the beginning of a cycle is one unit unless there is no water left to flow out. [Note 2]

We must now derive the specification which is *not* allowed to refer to the hidden variable *out*. A first attempt for S is:

$$in = \begin{bmatrix} \textbf{\textit{if}} & level = 0 & \textbf{\textit{then}}\ 9 \\ \textbf{\textit{elseif}} & level = 1 & \textbf{\textit{then}}\ 8 \\ \textbf{\textit{elseif}} & level > 1 & \textbf{\textit{then}}\ 0 \end{bmatrix} \qquad (13)$$
$$\wedge\ alarm' \equiv level \le 1$$

(Our assumption is that the machine works, 'much faster' than the cycle time of the water tank. Therefore, the machine can instantaneously set *in* as described above at the beginning of each cycle.)

The controller module described earlier (Fig. 6) implements the specification of (eqn. 13). The specification might at first sight appear correct, for it adds 9 units of water if the level is 0 units, and 8 units of water if the level is 1 unit ($1 + 8 = 9$); nothing is added otherwise. However, the machine specification is wrong, as can be seen by a counterexample. Consider a state at the beginning of a cycle in which $level = 1$. By the above specification $in = 8$. By W_2 it follows that $out = 1$, and hence by W_1, $level' = 8$, so the requirement R_2 will not be satisfied. The failed specification did not take into account the fact that there is

Note 2: The informal requirements state that 'every cycle, one unit of water is used'. This cannot be precise. If $level = 0$ at the beginning of a cycle, then there may be no outflow in that cycle. W_2 corresponds to a scenario in which the outflow valve is (a) automatically opened only when the level reaches 1 unit, and (b) releases exactly 1 unit every cycle as long as it is open. It is up to the software engineer to ascertain from the domain specialists the precise behaviour of the external world phenomena.

an outflow of 1 unit when the level is at 1 unit. The correct specification for the controller is

machine specification $S \cong S_1 \wedge S_2$ where:

$$\begin{cases} S_1: & in = (level \leq 1)|_0^9 \\ S_2: & alarm' \equiv level \leq 1 \end{cases} \tag{14}$$

which states that 9 units (or nothing) must always be added. The specification correctness (eqn. 1) holds if we can show the validity of

$$W \wedge S \rightarrow R \tag{15}$$

Gathering together all the information, we must prove

$$\begin{array}{l} W_0 : 0 \leq level \leq 10 \\ W_1 : level' = level + in - out \\ W_2 : out = (level \geq 1)|_0^1 \\ S_1 : in = (level \leq 1)|_0^9 \\ S_2 : alarm' \equiv level \leq 1 \\ \hline R_1 : 1 \leq level' \leq 9 \\ R_2 : level' = (level \leq 1)|_{level-out}^9 \\ R_3 : alarm' \equiv level \leq 1 \end{array}$$

The proof follows from three lemmas. R_3 can be obtained directly from S_2 (using reflexivity of implication 3.71):

$$\text{Lemma 1} : S_2 \rightarrow R_3. \tag{16}$$

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_1 since it has the most precise information (it is an equality, not an inequality). The resulting calculation (Fig. 7) which uses assumptions W_2 and S_1, yields:

$$\begin{array}{ll} & W_1 \\ = & \quad \langle \text{definition of } W_1 \rangle \\ & level' = level + in - out \\ = & \quad \langle \textbf{assumption } W_2 \rangle \\ & level' = level + in - (level \geq 1)|_0^1 \\ = & \quad \langle \textbf{assumption } S_1 \rangle \\ & level' = level + (level \leq 1)|_0^9 - (level \geq 1)|_0^1 \\ = & \quad \langle \text{case replacement and } level = 0 \vee level = 1 \vee level > 1; (10) \rangle \\ & \quad (level = 0 \rightarrow level' = level + 9 - 0) \\ & \wedge (level = 1 \rightarrow level' = level + 9 - 1) \\ & \wedge (level > 1 \rightarrow level' = level + 0 - (level \geq 1)|_0^1) \\ = & \quad \langle \text{Leibniz substitution 3.84(b) to first two conjuncts} \rangle \\ & \quad (level = 0 \rightarrow level' = 0 + 9 - 0) \\ & \wedge (level = 1 \rightarrow level' = 1 + 9 - 1) \\ & \wedge (level > 1 \rightarrow level' = level + 0 - (level \geq 1)|_0^1) \\ = & \quad \langle \text{arithmetic simplification} \rangle \\ & \quad (level = 0 \rightarrow level' = 9) \\ & \wedge (level = 1 \rightarrow level' = 9) \\ & \wedge (level > 1 \rightarrow level' = level - (level \geq 1)|_0^1) \\ = & \quad \langle \text{theorem of prop. logic} : ((p \rightarrow r) \wedge (q \rightarrow r)) \\ & \qquad \qquad \equiv (p \vee q \rightarrow r) \text{ to first two conjuncts} \rangle \\ & \quad (level \leq 1 \rightarrow level' = 9) \\ & \wedge (level > 1 \rightarrow level' = level - (level \geq 1)|_0^1) \\ = & \quad \langle \textbf{assumption } W_2 \text{ to reinsert } out \rangle \\ & \quad (level \leq 1 \rightarrow level' = 9) \\ & \wedge (level > 1 \rightarrow level' = level - out) \\ = & \quad \langle \text{arithmetic } level \leq 1 \vee level > 1; (10) \rangle \\ & level' = (level \leq 1)|_{level-out}^9 \\ = & \quad \langle \text{definition of } R_2 \rangle \\ & R_2. \end{array}$$

Fig. 7 *Calculational proof of Lemma 2*

$$\text{Lemma 2} : W_1 \wedge W_2 \wedge S_1 \rightarrow R_2. \tag{17}$$

The proof length is due to the need to do case analysis. It was precisely this case analysis that provided a counter-example to the naive specification of (eqn. 13). As we originally anticipated, R_1 can be derived from R_2. The technical report [19] presents the calculational proof, from which we obtain

$$\text{Lemma 3} : W_0 \wedge W_2 \wedge R_2 \rightarrow R_1. \tag{18}$$

Using the three lemmas, a quick calculational proof shows the validity of specification correctness $W \wedge S \rightarrow R$.

The cooling tank example can be checked automatically with the help of PVS (Fig. 8) The PVS descriptions of the external world, requirements, and specification for the cooling tank are shown in the figure. Conjecture *specification_correctness* (end of Fig. 8) proves automatically when submitted to the prover. The PVS file also shows an example of a sanity check to ensure that the outflow is correctly described. The final step involves refining the specification into an implementation M, which is easily achieved by changing routine *fill_tank* in Fig. 6 to 'if level <= 1 then in:=9 else in:=0'.

```
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 external world domain
external_world(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 specification
specification(level, inn, alarm) :  Bool =
        inn = (IF level <= 1 THEN 9 ELSE 0 ENDIF)
        AND
        alarm = (level <= 1)

specification_correctness :  CONJECTURE
        external_world (inn, out, level, level_f)
        AND
        specification (level, inn, alarm)
        IMPLIES
        requirement (level, level_f, out, alarm)

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

END tank
```

Fig. 8 *Automated PVS proof of the cooling tank system*

5 Discussion and Conclusions

Mathematical logic can be used throughout the software development life-cycle both as a design calculus and for documenting requirements, specifications, designs, and programs. 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 software engineering 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 [16] can be used in two courses (each lasting a semester) in logic and discrete mathematics in the first and second years. This is based on the idea of first teaching calculational logic, and then actually using the logic to reason about various discrete domains. This makes the logic immediately useful and reinforces its use. Simple programming examples serve to further motivate the material. The first-year mathematics programme for CS students at York University teaches such courses, based on the Gries-Schneider text. These courses are taught by mathematicians in the Mathematics department. At first, there was discomfort and opposition to the non-classical approach – both by faculty and students. Experience has gradually worn away the opposition, and former opponents of the change are now supportive. In one experiment, we discovered a high correlation between students who do poorly in the first year logic course, and students who do poorly in the first year programming course. [Note 3]
- The usual CS1 and CS2 courses can be taught in Eiffel, stressing design-by-contract [15, 21]. Currently the trend 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 [22]. Until such books 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.
- A third-year course in the use of tools such as PVS and B-Tool can build on the material of the first few years. Such a course uses languages that support design-by-contract, such as Eiffel, in a software engineering project. The tools are used to formally derive programs from specifications [9, 23–25].
- Comprehensive texts on object-oriented specification, design, and programming, with emphasis on the production of quality software using design-by-contract and BON/Eiffel are also available [15, 26]. These texts can form the basis of object-oriented design courses in upper years using 'lightweight' formal-methods.
- A fourth year course can introduce the formal methods of reactive systems (e.g using STeP [27], SPIN [28] or SMV [29]). Suitable textbooks are available for each of these courses [30], but more need to be written, emphasising the use of mathematical methods and calculation in design.

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. During the early 1970s, Berkeley was graduating over a 100 students a year who were accustomed to using Spice. They started working 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 analyse critical analogue circuits in virtually every IC designed in the United States in recent years [31].

The use of mathematical descriptions throughout software development is an idealisation. Not all requirements can be captured by predicates, at least not easily. Sometimes we need rough sketches or other types of descriptions. 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. 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.

6 Acknowledgments

We thank Prof. David Gries for his input to this work. We also thank the anonymous referees for their helpful remarks.

7 References

1 BICARREGUI, J.C., DICK, A.J.J., and WOODS, E.: 'Quantitative analysis of an application of formal methods'. In Proc. of FME'96, Third International Symposium of Formal Methods Europe, 1996, GAUDEL, M.C. and WOODCOCK, J.C.P. (Eds.) Springer-Verlag, LNCS 1051
2 DEAN, C.N., and HINCHEY, M.G. (Eds.): 'Teaching and learning formal methods' (Academic Press, London, 1996)
3 GLASS, R.L.: 'The software research crisis', *IEEE Softw.*, 1994, **11**, (6), pp. 42–47
4 HALL, A.: 'Seven myths of formal methods', *IEEE Softw.*, 1990, **7**, (5), pp. 11–19
5 HINCHEY, M., and BOWEN, J.: 'Applications of formal methods' (Prentice Hall, 1995)
6 PARNAS, D.L., MADEY, J., and IGLEWSKI, M.: 'Precise documentation of well-structured programs', *IEEE Trans. Softw. Eng.*, 1994, **20**, (12), pp. 948–976
7 PARNAS, D.L., and MADEY, J.: 'Functional documentation for computer systems', *Sci. Comput. Program.*, 1995, **25**, pp. 41–61
8 GUNTER, C.A., GUNTER, E.L., JACKSON, M., and ZAVE, P.: 'A reference model for requirements and specifications', *IEEE Softw.*, 2000, **17**, (3), pp. 37–43
9 HEHNER, E.C.R.: 'A practical theory of programming' (Springer-Verlag, New York, 1993)
10 OSTROFF, J.S.: KRAMER, J. 'Temporal logic for real-time systems. Advanced software development series' (Research Studies Press Limited (John Wiley and Sons, Taunton, England, 1989)
11 JACKSON, M.: 'Software requirements & specifications' (Addison-Wesley, 1995), pp. 127
12 PARNAS, D.L., and CLEMENTS, P.C.: 'A rational design process: How and why to fake it', *IEEE Trans Softw. Eng.*, 1986, **SE-12**, (2), pp. 251–257
13 OSTROFF, J.S. and R.F. PAIGE. 'The timed predicative calculus as a framework for comparative semantics'. York University. CS-2000-01, 2000. http://www.cs.yorku.ca/techreports/2000/CS-2000-01.html
14 PAIGE, R.F. and OSTROFF, J.S. 'An object-oriented refinement calculus'. Department. of Computer Science, York University, Toronto. CS-1999-07, 1999. http://www.cs.yorku.ca/techreports/1999/CS-1999-07.html
15 MEYER, B.: 'Object-oriented software construction' (Prentice Hall, 1997)
16 GRIES, D., and SCHNEIDER, F.B.: 'A logical approach to discrete math' (Springer Verlag, 1993)
17 GRIES D., On presenting monotonicity and on EA => AE. Cornell University, Department of Computer Science, TR95-1512, 1995, http://www.cs.cornell.edu/gries/Papers/Monotonicity.ps
18 OWRE, S., RUSHBY, J., SHANKAR, N., and HENKE, F.V.: 'Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS', *IEEE Trans. Softw. Eng.*, 1995, **21**, (2), pp. 107–125
19 OSTROFF, J.S. and R. PAIGE. 'The logic of software design'. Computer Science, York University. CS-98-04, 1998. http://www.cs.yorku.ca/General/techreports/98/CS-98-04.html
20 NASA. Formal methods specification and analysis guidebook. NASA Office of Safety and Mission Assurance. NASA-GB-001-97, 1997.
21 JEZEQUEL, J.-M., and MEYER, B.: 'Design by contract: the lessons of the ariane', *IEEE Comput.*, 1997, **30**, (1), pp. 129–130
22 PAYNE, J.E., SCHATZ, M.A., and SCHMID, M.N.: 'Implementing assertions for Java', *Dr. Dobb's Journal*, 1998, **281**, pp. 40–44
23 ABRIAL, J.-R.: 'The B-Book: Assigning programs to meanings' (Cambridge University Press, 1996)
24 GRIES, D.: 'The science of programming' (Springer-Verlag, 1985)

Note 3: A comparison was made between students in the mid-term test of the logic and programming courses in the fall term of 1998; 57 out of 64 students (89%) who failed the logic course also failed the programming course. The correlation between good students in logic and programming was less; 25 out of 46 students (54%) who got a B or higher in logic also got a B in programming.

25 MORGAN, C.: 'Programming from specifications' (Prentice Hall, International Series in Computer Science, 1994)

26 WALDEN, K., and NERSON, J.-M.: 'Seamles object oriented software and architecture' (Prentice Hall, 1995)

27 MANNA Z. 'STeP'. The Stanford Temporal Prover. Dep. of Computer Science, Stanford University, STAN-CS-TR-94-1518 1994

28 HOLZMANN, G.: 'The model checker spin', *IEEE Trans Softw. Eng.*, 1997, **23**, (5), pp. 279–295

29 BURCH, J.R., CLARKE, E.M., MACMILLAN, K.L., DILL, D.L., and HWANG, L.J.: 'Symbolic model checking: 10 ∧ 20 states and beyond', *Inf. Comput.*, 1992, **98**, (2), pp. 142–170

30 HUTH, M., and RYAN, M.: 'Logic in computer science: Modelling and reasoning about systems' (Cambridge University Press, 1999)

31 PERRY, T.S.: 'Donald O. Pederson', *IEEE Spectr.*, 1998, **35**, (6), pp. 22–27