Formal Methods in the Classroom: The Logic of Real-Time Software Design

Jonathan S. Ostroff and Richard F. Paige
Department of Computer Science, York University
Toronto, Ontario M3J 1P3, Canada.
{jonathan,paige}@cs.yorku.ca

Abstract

In recent years, much progress has been made towards the development of mathematical methods ("formal methods") through which it is possible, in principle, to specify and design software to conform to specifications. In this paper, we provide an overview of how formal methods – and particularly real-time 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.

1. 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 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 [3, 21]. 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 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 [4, 6, 7] 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.

This paper is based on the report [17], which is available via the WWW. In this paper, we focus on some of the considerations associated with reactive or real-time systems, and refer the reader to the report for the general overview of the use of mathematical methods in software design.

2. Rational Software Development

In real-time software development, the progression from requirements to an implemented program is a way of bridging the gap between the phenomena of the real-world domain W and those of the software controller, M [11].

A rational development process, where each step follows from the previous ones in the most elegant and economic order, does not exist in reality for complex systems. Nevertheless, we can fake it [20]. We can try to follow an established process as closely as possible.

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 controller, expressed in terms of the shared phenomena $W \cap M$. Specifications describe the interface or boundary between the controller 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 controller behaves like *M.prog*, then the specification *M.spec* is satisfied, i.e.,

$$M.prog \rightarrow M.spec \tag{1}$$

   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 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 controller implementation satisfies its specification.

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

$$W.desc \wedge M.spec \rightarrow R \tag{2}$$

   where we may use our knowledge of the properties of the real-world domain $(W.desc)$ to prove (2).

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

$$W.desc \wedge M.prog \rightarrow R \tag{3}$$

In the process described above, a distinction is made between specifications and requirements. "Specification" is one of a trio of terms: requirements, specifications and programs. Requirements are all and only about the environment of the controller, i.e., the real-world phenomena. By contrast, programs are all and only about the controller phenomena. Programmers are interested in phenomena at the interface $W \cap M$, but this interest is motivated by the need to obtain the data on which the controller 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 sense to either the customer or the programmer. Although specifications are programs, they may not be executable. In fact, we would prefer that they not be tainted by implementation bias, i.e., with irrelevant controller 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).

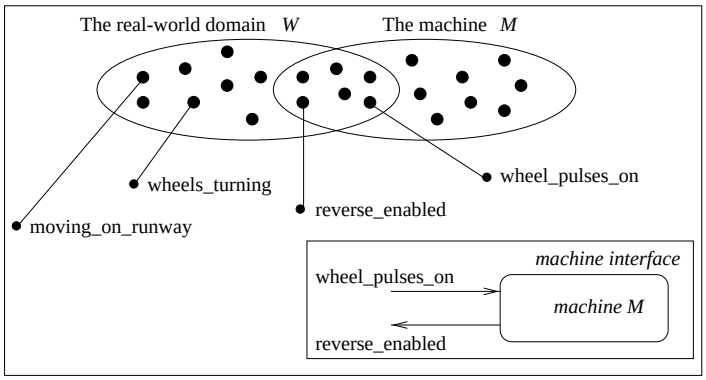

Figure 1. Airplane overshooting the runway

Fig. 1 shows the phenomena that we are concerned with. The requirement *R* was

$$R : \mathrm{reverse\_enabled} \equiv \mathrm{moving\_on\_runway}$$

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

$$W.desc : \begin{cases} \mathrm{wheel\_pulses\_on} \equiv \mathrm{wheels\_turning} \\ \mathrm{wheels\_turning} \equiv \mathrm{moving\_on\_runway} \end{cases} \tag{4}$$

So they derived the specification

$$M.spec : \ \mathrm{reverse\_enabled} \equiv \mathrm{wheel\_pulses\_on}$$

For the above description of the real-world domain, (2) is indeed a theorem. Unfortunately, the developers did not understand the real-world domain correctly. The first property listed in (4) was indeed a correct description of the real-world domain. But, the second property 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

$$W.desc = \left\{ \begin{array}{l} \text{wheel\_pulses\_on} \equiv \text{wheels\_turning} \\ (\text{wheels\_turning}) \vee (\text{aquaplaning}) \equiv \\ \quad \text{moving\_on\_runway} \end{array} \right. \quad (5)$$

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

3. Using Logic for Software Description

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 [19].

- *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 (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 also describe how a program is composed from other programs. Program text is an example of a constructive description. For example, the text for the module in Fig. 2 describes how the module body 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 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 [11].

A refutable description describes some domain, saying something about it that can – in principle – 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).

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.

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, 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. In the full report [17], we illustrated this type of knowledge with a simple example that shows how logic may be used to:

- make informal descriptions precise,
- calculate properties (by proving theorems), and
- understand the role of counterexamples.

We have shown how logic can be used for describing requirements, specifications, and programs. We also explained that logic can be used as a descriptive calculus throughout the software life-cycle including design, implementation and documentation. The logical calculational

format can be used in various phases of the software lifecycle, 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 [5] 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 logic and discrete mathematics courses for computer science students are using the calculational approach, based on the text [5]. 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. A Case Study – Cooling Tank

In the full report [17], we described how calculational logic can be used in all phases of software design. In this section, we present a case study that illustrates the use of logical methods and tools through all phases of software design from requirements to implementations.

The case study involves the use of conditional expressions such as

$$\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2$$

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 $b \mid^{e_1}_{e_2}$ (see [17] for further details). Logic **E** as described in [5] provides the two axioms

$$b \rightarrow (b \mid^{e_1}_{e_2} = e_1) \qquad\qquad \neg\, b \rightarrow (b \mid^{e_1}_{e_2} = e_2)$$

for conditional expressions. We will need more powerful theorems to simplify calculation. We therefore refer the reader to the Appendix of [17], in which further theorems of conditional expressions are listed. This Appendix also provides a proof of the theorem (6) which is an illustration of the utility of Logic **E** for developing new theory. Under the assumption that $p \rightarrow b$ is a theorem,

$$(p \rightarrow E[z := b \mid^{e_1}_{e_2}]) \equiv (p \rightarrow E[z := e_1]) \qquad (6)$$

(6) 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 (see the appendix in [17]) the following is a theorem:

$$[x' = x + (x \leq 1 \mid^{9}_{y}) - (x \geq 1 \mid^{1}_{z})] \equiv \left[\begin{array}{c} (x = 0) \rightarrow (x' = x + 9 - z) \\ \wedge (x = 1) \rightarrow (x' = x + 9 - 1) \\ \wedge (x > 1) \rightarrow (x' = x + y - 1) \end{array}\right]$$

We now present an informal description of the case study.

4.1. A Cooling Tank Description

> "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. 2. The body of the module executes *set_alarm; fill_tank* once every cycle.

```
Module controller
Inputs
     level: LEVEL
       -- tank input, where type LEVEL={0..10}
Outputs
     alarm: BOOLEAN
       -- initially false, raises tank alarm
     in: LEVEL
       -- setpoint for tank input value
Body
     every 20 seconds do
       set_alarm; fill_tank
     end
Private routines used in Body
     set_alarm is
       do alarm := (level<=1) end

     fill_tank is do
          if     level=0 then in := 9
          elseif level=1 then in := 8
          else   in := 0
       end
end controller
```

Figure 2. Faulty code for the cooling tank

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

Our rational software design method requires that we first divide the system of interest into the real-world domain W and the controller 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. 3 illustrates the phenomena of the real-world domain, including phenomena shared with the controller (*in*, *level* and *alarm*). The water outflow *out* is not a shared phenomenon as the controller cannot measure it. 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 informal descriptions.

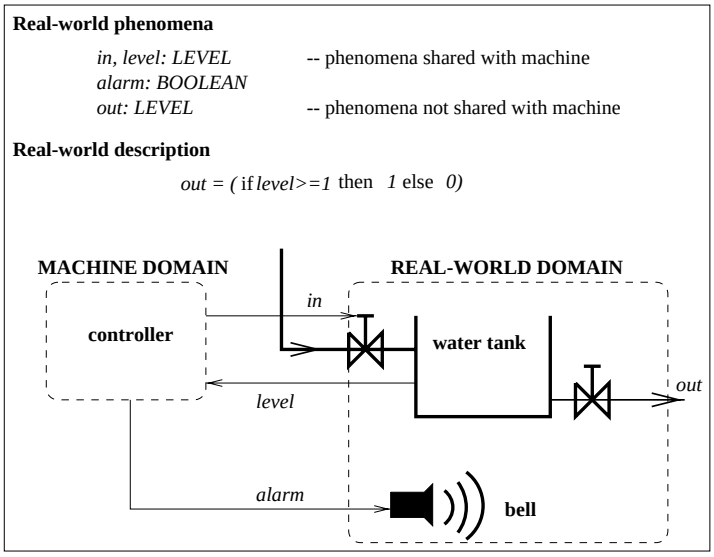

Figure 3. Rough sketch of the cooling tank

Having identified the phenomena of interest, the next step is to write the requirements for the cooling tank. We assume that the controller 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.

$$R \mathrel{\widehat{=}} (R_1 \wedge R_2 \wedge R_3) : \left\{ \begin{array}{ll} R1: & (1 \leq \mathit{level}' \leq 9) \\ R2: & \mathit{level}' = (\mathit{level} \leq 1 \mid^{9}_{\mathit{level}-\mathit{out}}) \\ R3: & (\mathit{alarm} \equiv \mathit{level} \leq 1) \end{array} \right. \tag{7}$$

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 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 design method is to describe the properties characterizing the real-world domain.

$$W.\mathit{desc} \mathrel{\widehat{=}} W.d_1 \wedge W.d_2 : \left\{ \begin{array}{ll} W.d_1: & \mathit{level}' = (\mathit{level} + \mathit{in} - \mathit{out}) \\ W.d_2: & \mathit{out} = (\mathit{level} \geq 1 \mid^{1}_{0}) \end{array} \right. \tag{8}$$

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

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 controller may therefore *not* refer to it. Here is a first attempt at the controller specification:

$$\left[\begin{array}{l} \mathit{in} = (\text{if } \mathit{level} = 0 \text{ then } 9 \\ \quad \text{elseif } \mathit{level} = 1 \text{ then } 8 \\ \quad \text{elsif } \mathit{level} > 1 \text{ then } 0) \end{array} \right] \wedge (\mathit{alarm} \equiv \mathit{level} \leq 1) \tag{9}$$

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

4.2. Verifying Correctness

The controller module in Fig. 2 implements the specification of (9). 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 controller specification is wrong, as can be seen by a counterexample. Consider a state at the beginning of a cycle in which $\mathit{level} = 1$. According to the above specification, $\mathit{in} = 8$. Thus $\mathit{out} = 1$ by $W.d_2$. Hence, by $W.d_1$,

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

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 (2) was performed.

A correct specification for the controller is:

$$M.spec \mathrel{\widehat{=}} M.s_1 \wedge M.s_2 : \begin{cases} M.s_1 : & in = (level \leq 1 \mid_0^9 \\ M.s_2 : & alarm \equiv (level \leq 1) \end{cases} \quad (10)$$

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 holds if we can show the validity of

$$\forall\, level : LEVEL \mid W.desc \wedge M.spec \rightarrow R \quad (11)$$

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 (8) and the controller its specification, then the requirements will be satisfied. By Logic **E**, this is the same as proving that

$$(0 \leq level \leq 10) \rightarrow (W.desc \wedge M.spec \rightarrow R)$$

Gathering all the information together, we must prove:

$$\begin{array}{l} W.d_0 : (0 \leq level \leq 10) \\ W.d_1 : level' = (level + in - out) \\ W.d_2 : out = (level \geq 1 \mid_0^1) \\ M.s_1 : in = (level \leq 1 \mid_0^9) \\ M.s_2 : (alarm \equiv level \leq 1) \\ \hline R_1 : (1 \leq level' \leq 9) \\ R_2 : level' = (level \leq 1 \mid_{level-out}^9) \\ R_3 : (alarm \equiv level \leq 1) \end{array}$$

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

$$M.s_2 \rightarrow R_3$$

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, which also uses the assumptions $W.d_2$ and $M.s_2$, can be found in [17] and yields:

$$W.d_2 \wedge M.s_1 \wedge W.d_1 \rightarrow R_2 \quad (12)$$

The proof of (12) is long (in fact, longer than we had hoped). The proof length is due to the need to do case analysis. It was precisely this case analysis that provided a counterexample to the naive specification (9).

As we originally anticipated, R_1 can be derived from R_2. This is shown in Fig. 4.

$$\begin{array}{ll} & R_2 \\ = & < \text{definition of } R_2;\ \text{IF} - \text{transform};\ \text{assumption } W.d_2 > \\ & [level \leq 1 \wedge level' = 9] \vee \\ & [level > 1 \wedge (level' = level - (level \geq 1 \mid_0^1))] \\ = & < (10.14\text{b}) \text{ with } (level > 1) \rightarrow (level \geq 1) > \\ & ((level \leq 1) \wedge (level' = 9)) \vee [(level > 1) \wedge (level' = level - 1)] \\ \Rightarrow & < \text{arithmetic} : level' = 9 \rightarrow R_1 \text{ and monotonicity} > \\ & R_1 \vee [(level > 1) \wedge (level' = level - 1)] \\ = & < true \text{ is the identity of conjunction} > \\ & R_1 \vee [(level > 1) \wedge true \wedge (level' = level - 1)] \\ = & < \text{assumption } (0 \leq level \leq 10) \text{ and theorem equivalence} > \\ & R_1 \vee [(level > 1) \wedge (0 \leq level \leq 10) \wedge (level' = level - 1)] \\ = & < \text{math} : (level > 1) \wedge (0 \leq level \leq 10) = (2 \leq level \leq 10) > \\ & R_1 \vee [(2 \leq level \leq 10) \wedge (level' = level - 1)] \\ = & < \text{Leibniz substitution with } level = level' + 1 > \\ & R_1 \vee [(2 \leq level' + 1 \leq 10) \wedge (level' = level - 1)] \\ \Rightarrow & < \text{weakening theorem } p \wedge q \rightarrow p \text{ and monotonicity} > \\ & R_1 \vee (2 \leq level' + 1 \leq 10) \\ = & < \text{arithmetic simplification} > \\ & R_1 \vee (1 \leq level' \leq 9) \\ = & < \text{definition of } R_1 \text{ and } (p \vee p) = p > \\ & R_1 \end{array}$$

Figure 4. Proof deriving R_1 from R_2

From this, we can obtain

$$(W.d_0 \wedge W.d_2) \rightarrow (R_2 \rightarrow R_1)$$

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

4.3. Tool Support

The cooling tank example can be checked automatically with the help of PVS (Fig. 5). The PVS descriptions of the real-world domain, requirements, and controller specification for the cooling tank are shown in the figure. The conjecture *system_correctness* (end of Fig. 5) is proved automatically when submitted to the PVS prover.

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 [18], but there are

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

%Designations: "level_f" is final value of "level"
level, level_f, inn, out: VAR LEVEL
alarm: VAR bool

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

%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))

%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)
```

Figure 5. PVS theory for the cooling tank

now a variety of tools available that have been used in selected industrial applications.

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

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 [14]).

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 [16].

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 setting as a function of time.

The StateTime [14], STeP [13] and Hytech tools [1] 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.

Students can be introduced to the use of automated tools, such as PVS or the B-Tool [10], in the later stages of their undergraduate education, and particularly 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.

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 such as the calculational logic, which the students can use for designing small systems or small components, and as a supplement to the automated tools when difficulties arise in proof.

5. 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 suggest a possible curriculum sequence that makes use of calculational methods, from introductory un-

dergraduate courses, through upper-year real-time and reactive software engineering courses.

- The logic text by Gries and Schneider [5] 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 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.

- A third year software design course, taught using Eiffel and the Business Object Notation [15], can introduce formal methods for sequential and object oriented software design. A grounding in calculational logic is vital in writing specifications, and in performing consistency checks and other analyses.

- A fourth year course can introduce the formal methods of reactive systems (e.g., using STeP [13], SPIN [8] or SMV [2]). Suitable textbooks are available for each of these courses, but more need to be written, emphasizing the use of mathematical methods in design.

A variety of applications of formal methods to industrial systems have been reported. 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 [12] and the Ariane 5 heavy launcher [9], which illustrate the need for professional standards in all aspects of 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. 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 [22].

In software development, the practitioner has to subordinate everything to the over-riding 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.

References

[1] R. Alur, T. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[2] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[3] N. Dean and M. Hinchey (eds.). *Teaching and Learning Formal Methods*. Academic Press, 1996.

[4] R. Glass. The software research crisis. *IEEE Software*, 11(6):42–47, 1994.

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

[6] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[7] M. Hinchey and J. Bowen. *Applications of Formal Methods*. Addison-Wesley, 1995.

[8] G. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[9] J.-M. Jezequel and B. Meyer. Design by contract: the lessons of the Ariane. *IEEE Computer*, 30(1):129–130, 1997.

[10] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[11] M. Jackson. *Software Requirements and Specifications*. Addison-Wesley, 1995.

[12] N. Leveson and C. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[13] Z. Manna. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford University, 1994.

[14] Z. Manna and A. Pneuli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

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

[16] J. Ostroff. A visual toolset for the design of real-time discrete event systems. *IEEE Trans. on Control Systems Technology*, 5(3):320–337, 1997.

[17] J. Ostroff and R. Paige. The logic of software design. Technical Report TR-98-04, York University, 4700 Keele St., Toronto, Ontario M3J 1P3, Canada, July 1998.

[18] S. Owre, J. Rushby, N. Shankar, and F. Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Software Engineering*, 21(2):107–125, 1995.

[19] D. Parnas. Mathematical descriptions and specification of software. In *Proc. IFIP World Congress 1994, Vol. I*, 1994.

[20] D. Parnas and P. Clements. A rational design process: How and why to fake it. *IEEE Trans. Software Engineering*, 12(2):251–257, 1986.

[21] D. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Trans. Software Engineering*, 20(12):948–976, 1994.

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