StateTime — a Visual Toolset for the Design and Verification of Real-Time Systems.

Jonathan S. Ostroff
Department Of Computer Science, York University[1],
4700 Keele Street, North York Ontario, Canada, M3J 1P3.
Email: jonathan@cs.yorku.ca Tel: 416-736-2100 X77882 Fax: 416-736-5872
Electronic Technical Report Number: CS-ETR-94-07
ftp.cs.yorku.ca:/pub/TECH-REPORTS/General-CS/CS-ETR-94-07/text.ps2.Z

Abstract: StateTime is a prototype toolset that uses visual specifications and temporal logic for design and verification. The toolset is especially useful for designing real-time safety critical systems. A design methodology is described that can be used together with the tools. The methodology and the tools are illustrated with a nontrivial example taken from the actual requirements document of the Candu nuclear reactor. The specification of part of the reactor shutdown mechanism is shown to be incorrect. A revised design is proposed and verified.

StateTime consists of three tools: BUILD, VERIFY and DEVELOP. The BUILD tool allows the designer to model the system using a graphical language based on timed transition models (TTMs). Concurrency, nondeterminism, hierarchy, synchronous interaction, time bounds and integer data variables are supported.

The VERIFY tool automatically checks finite state systems for various properties specified in real-time linear temporal logic (RTTL). Safety properties such as freedom from deadlock and mutual exclusion, liveness properties such as termination and accessibility, and bounded response times can be specified and verified. DEVELOP supports a semi-automated disciplined approach to software development for infinite state systems.

Keywords: formal methods, verification, CASE tools, real-time temporal logic, timed transition models.

1. This work is supported in part by the National Science and Engineering Council of Canada.

Table of Contents

1.0 Overview 3

1.1 CASE tools for design automation 3
1.2 Design Methodology....6
1.3 StateTime Terminology....10

2.0 Example 1: The real-time dining philosophers 12

2.1 Building a model of the college....14
2.1.1 Root activities, ordinary activities and basic activities 15
2.1.2 AND-composition, XOR composition and Activity Variables....16
2.1.3 States, computations, state-formulas and RTTL formulas....17
2.1.4 Local Events, Shared Events and Transitions....18
2.1.5 Spontaneous events 20
2.1.6 Grouped Events 20
2.1.7 Data variables, Assignments and Guards 22
2.2 Using BUILD to validate the model by executing it 22
2.3 Specify the RTTL plant requirements....24
2.4 Design a controller — the porter 24
2.4.1 Control Interface Specifications 25
2.4.2 Designing the controller 25
2.5 VERIFY — submit the design for verification 25
2.6 DEVELOP — synthesizing controllers 28

3.0 Example 2: The delayed reactor trip (DRT) problem. 29

3.1 Modelling the plant....31
3.2 The software controller....35
3.3 DRT requirements expressed in Temporal Logic....36
3.4 Using the VERIFY tool....39

4.0 Conclusions 39

5.0 Acknowledgments 41

6.0 REFERENCES 42

1.0 Overview

Computers are increasingly used to monitor and control safety critical systems. Real-time software controls aircraft, shuts down nuclear power reactors in emergencies, keeps telephone networks running, and monitors hospital patients. The use of computers in such systems offers considerable benefits, but also poses serious risks to life and the environment [24].

There is general consensus in the software and control systems literature that real-time systems are difficult to model, specify and design. In addition, experience has shown that software components of systems are problematic, perhaps even more so than mechanical or other hardware components. Software is complex; consider the documentation needed for even simple modules. Software is not usually robust as small errors have major consequences, and software is notoriously difficult to test; the number of test cases that must be checked becomes unmanageably large even in small systems [27].

Formal methods [8] have been proposed for verifying the correctness of safety critical systems. It is clear, however, that the application of formal methods to realistic systems will require suitable automatic tools for model execution (simulation), code generation and verification.

This paper presents a methodology and a toolset (*StateTime*) for designing (possibly nondeterministic concurrent) real-time systems using a formal method. The formal method uses timed transition models (*TTMs*) for representing complex systems, and real-time temporal logic (*RTTL*) for specifying their requirements. The methodology is a set of recommended design steps for applying the formal methods to the design of actual systems. The StateTime tool automates many of the design steps including modelling complex systems, executing the model, and the verification of their correctness.

TTMs are computational models that specify concurrent processes, non-deterministic behaviour, communication between modules, structured programs, and real-time constraints. The transitions of a (conceptual) discrete external clock, that ticks infinitely, is interleaved with other system transitions to model the progress of time.

The temporal logic RTTL is a useful specification language for expressing a variety of properties including: freedom from deadlock, mutual exclusion of critical regions, liveness properties (e.g. access to critical regions and process fairness), and real-time response.

The underlying formal method (TTMs/RTTL) used in the sequel for verifying system correctness has been presented elsewhere [19,18,17,24]. This paper will therefore concentrate on the StateTime toolset and the proposed design methodology.

The rest of this overview will survey some of the tools reported in the literature and compare them to StateTime.

1.1 CASE tools for design automation

Two directions have been pursued in recent years to deal with the problems of real-time software design [6]:

1. Commercial tools have been developed incorporating structured methods for specifying real-time systems requirements [32,9,33,29]. These tools are in actual use, and have been successful on the whole in removing ambiguities in the requirements. However, these methods are at best semi-formal. They lack a precise semantics and rigorous verification methods (e.g. through model checking, proof calculi or algebraic bisimulations). Two notable tools are ObjectTime and Statemate. The main feature of ObjectTime [28] is its use of classes and inheritance allowing for the re-use of processes, data structures and communication protocols between processes. The Statemate tool [7] is unique amongst commercial tools as it has a formal semantics (statecharts). The semantics is used for simulation, code generation, and the verification of deadlock freedom and reachability analysis. A more detailed comparison between Statemate and StateTime is provided later in this section.

2. In academia, formal, mathematically precise methods have been proposed for the design of real-time systems [24]. Many practically-oriented software engineers will probably consider that formal methods cannot be applied to realistic real-time systems. However, software safety has become a vital public policy issue. It has been proposed that safety-critical software be certified by licensed software engineers, in part by the use of formal validation methods [5]. These academic methods will therefore have to be taken out of the realm of theory into practice [8].

In response to the need for practical verification of structured systems, tools have been developed recently that combine visual representation methods with (semi) automatic verification of real-time systems. For example, the modechart tool [10,13] uses statechart like visual formalisms and real-time logic (RTL). The ExSpect tool in [31] uses Petri Nets for visualization.

The tool reported in [10] was found to be a "significant advance in the state-of-the-art of specification and verification" despite some of its limitations (it does not allow for modelling data values, has a limited repertoire of specification formulas, and does not allow proofs of a combination of modechart and logical RTL formulas). The authors of [10] state that many existing CASE tools are ill-thought out and inadequately tested; research in interface design is much needed.

The following general features are needed for such tools:

- Hide the complexity of the formal method from the designer where feasible. For example, use a visual language to model the system. Concurrency, non-determinism, communication between modules, real-time constraints, and hierarchies should be presented in an appealing graphical fashion. Top down (decomposition of an abstract system into subsystems) and bottom up development (composing subsystems into larger systems) must be supported.
- Allow the user to execute the model. This will allow the designer to validate the model as it is developed. Animated simulation is also useful.
- Automate verification methods, whenever this can be done, e.g. implement finite state model checking or synthesis procedures. Semi-automated procedures should be implemented where total automation cannot be achieved, for example in the case of very large or infinite state systems.

- Generate code into an appropriate real-time language such as Ada or Occam.
- Provide a well thought out user interface.
- The toolset should be portable to as many computing platforms as possible in order to encourage its use and acceptance.

Statemate is one of the few commercially available tools that meets most of the above goals (although it is costly to purchase). It is based on a formal model (statecharts), allows for dynamic execution in which triggers can be provided interactively on the fly, and which can formally verify various properties including reachability of conditions, deadlock, nondeterminism, usage of elements and racing.

The StateTime tool discussed in this paper is a prototype and thus does not compare to Statemate in some important respects. The most important deficiency in StateTime is that presently only integer types are available for data variables, whereas Statemate has the full range of types available in normal programming languages. A future object-oriented version of StateTime is under development that will make available to the user all the basic Smalltalk[2] classes.

In other respects StateTime is more expressive than Statemate. While retaining the hierarchical and concurrent constructs of statecharts, State*Time* has facilities for expressing and verifying certain kinds of real-time properties that Statemate is unable to deal with directly.

Statemate has time-out and scheduled *transitions*. These transitions allow for an exact delay of a specified period after which the transition occurs. By contrast, StateTime has a much richer hierarchy of timing properties that can constrain the behaviour of a transition $\tau[l, u]$ (with lower time bound l clock ticks and upper time bound u). In order of increasing stringency of timing these transition are: spontaneous, just and timed transitions.

Spontaneous transitions may occur at any point in time (after they are enabled via a "guard"), or they may never occur. An example is a transition that represents the event of a device failure. In the sequel, spontaneous transitions are indicated by the fact that their upper time bound is infinity (∞).

Just transitions must eventually occur if they are continually enabled. For example, a process that is continuously enabled to enter its critical region should eventually be allowed in. The external clock must always eventually tick (i.e. *tick* transitions must be taken infinitely). Justice is qualitative in the sense that although a just transition must occur, no finite bound on the time of occurrence is given.

Timed transitions must occur within an interval specified by a lower and an upper time bound. For example, sending a message may take between 5 and 8 ticks of the clock. This type of nondeterminism, that is caused by inexact knowledge of precise times, is commonly found in many real-time systems.

Statemate cannot distinguish between just or spontaneous transitions (a Statemate transition must be generated somewhere in the system for it to be

2. The StateTime BUILD tool is already implemented in ParcPlace's Smalltalk V4.1. The object-oriented nature of Smalltalk has been useful as a prototyping language for BUILD as the requirements continue to change rapidly. Smalltalk provides suitable support for the pictorial nature of BUILD, and the tool is automatically available on a wide variety of platforms (e.g. Sun, Dec, HP, IBM Apple Macintosh and Windows) without any need to port the software.

sensed). Nor is it able to express timed transitions in a direct manner. For example, to represent a transition $\tau[2, 5]$ in Statemate, two time-outs (one for the lower and one for the upper time bound) and some intermediate actions and states are needed.

There is also a fundamental between StateTime and Statemate with respect to program verification. The Statemate reachability test can check whether there is *a computation* from the initial state to a specified condition. By contrast, StateTime generates the complete reachability graph, and can therefore check that a certain property holds in *all computations* (also called *trajectories*).

Statemate cannot check timing properties directly. An example of a timing property is *real-time response*, which in temporal logic is written $f_1 \Rightarrow \Diamond_{[l, u]} f_2$. This property asserts that every time the condition f_1 becomes true, then the condition f_2 must eventually become true in all subsequent computations at a point in time that is between l and u ticks inclusive from f_1 (such temporal properties are explained in more detail in the sequel).

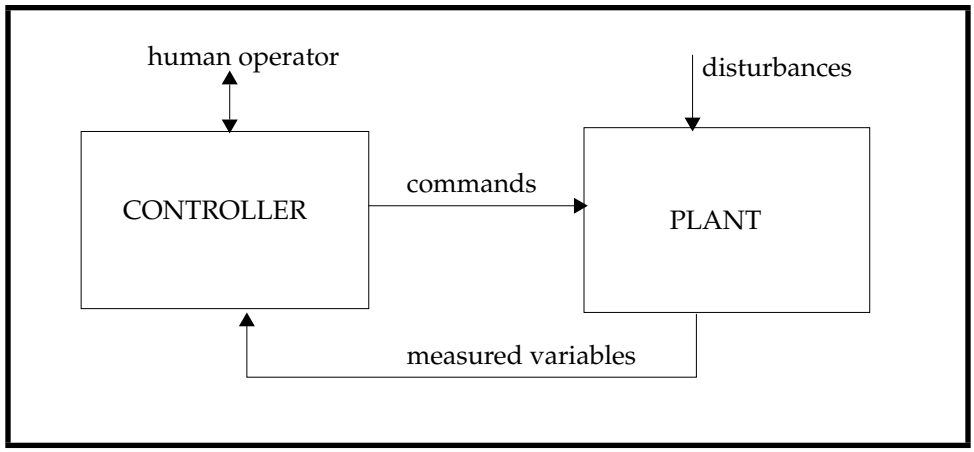
To check a real-time response property in Statemate, a *watchdog* must be appended to the system to detect the various conditions, and the response property must be reformulated as a reachability problem. A watchdog is an observer that has access to all the system variables without affecting them. However, using a watchdog increases combinatorially the size of the state space that must be checked.

In contrast, StateTime checks a small but important subset of temporal logic properties (such as deadlock, safety, liveness and real-time response) without the need of a watchdog. Properties not within the core set, however, will also require the use of a watchdog as in Statemate.

1.2 Design Methodology

The system under design (SUD) is usually divided into two parts, the *plant* (often called the "environment) and the *controller* (often called the "system"). The plant and controller interact with each other concurrently as shown in Figure 1,

FIGURE 1. The closed loop system: SUD = PLANT || OPERATOR



and we write

SUD = plant || controller

The *plant* is that part of SUD that is a given. The designer cannot change the plant, although the plant may have sensors and actuators through which its state can be measured and manipulated. For example, in the reactor shutdown system treated in the sequel, the plant is the nuclear reactor including the sensors for power and pressure, and the shutdown relay with its actuator to open and close the relay.

The environment may also involve processes such as the atmosphere and weather that cause disturbances to the plant. Such disturbances (the uncertainties due to weather, equipment failure and other unknowns) will often be modelled by spontaneous transitions in the plant. In the case of the reactor, disturbances are modelled by the pressure or power going beyond safe levels. Part of the job of the controller will be to operate the plant safely in the face of disturbances.

The *controller* is that part of the SUD that is to be designed. In the case of the reactor shutdown system, it is a computer program implemented on a microprocessor, that continually monitors the pressure and reactor variables of the plant, and issues appropriate commands to the shutdown relay to open or close depending on the measurements. In the original reactor design, the controller was implemented in hardware using comparators, timers and other such devices.

In the sequel, the term *model* is used to denote how a system (e.g. the plant) *actually* behaves. The term *requirement* is used to denote how a system *should* behave (the actual behaviour may not match the required behaviour).

The term *specification* is used for formal languages that represent models or requirements; the context in which "specification" is used will indicate in which sense it is being used. Two examples of specification languages are used in this paper: TTMs and RTTL. Timed transition models (TTMs) are visual languages (states and events) usually used for modelling systems (plants or controllers). Real time temporal logic (RTTL) is a high level abstract language usually used for stating requirements. It is possible to use RTTL for modelling systems and TTMs for stating requirements, but this is not their usual function. The notions of TTMs and RTTL will be explained in detail in the sequel.

The reactor might fail to shutdown (in the absence of the controller) when the power and pressure are above critical values — this type of behaviour is "illegal"; nevertheless such illegal behaviour must be represented in the model of the plant. A requirement of the reactor is that when the power and pressure are above critical values, then the reactor should shutdown within a certain number of ticks of the clock (i.e. the requirements specify the "legal" behaviour of the plant). It is the job of the controller to ensure that the requirements are satisfied, i.e. the closed loop system of plant and controller (SUD) must only execute legal behaviour.

The requirements specify *what* the controller is to do *to the plant*, not how the controller itself is to operate. For this reason, the requirements will refer primarily to entities (variables, states and events) in the plant, and not to entities in the controller. An example of a requirement is: "whenever the *power* reaches an acceptable level then the *relay* should be *closed* within two ticks of the clock". Note that "power", "relay" and "closed" are plant (not controller) entities.

The job of the designer is: given a model of the *plant* and given a set of *requirements* for correct behavior, design a *controller* so that SUD = plant || controller satisfies its requirements. The verification problem is: given the *requirements* and SUD, check that SUD satisfies the requirements.

Computer failures are not always a result of coding errors, but are either a result of omissions in the requirements or mishandled plant and environmental conditions [15]. What is needed is a design methodology for dealing with safety critical systems. It is essential to provide a model of the plant and to write down the requirements before proceeding to the design of the controller. The designer will therefore typically proceed as follows:

1. *Model of the plant*. An abstract model of the plant must be constructed. A balance must be sought between abstracting out unnecessary features, while at the same time providing enough information to indicate illegal behaviour of the plant in "open loop" (in the absence of the controller). In the reactor example, the behaviour of the relay (closing and opening), and the way in which pressure and power variables change must be modelled.

2. *Specify requirements*. The plant (without a controller) may obviously behave in an inappropriate fashion (e.g. the relay may not open when the pressure is high). The required plant behaviour must therefore be specified as formulas of the temporal logic RTTL. (Since Step 1 enumerated the relevant entities of the plant and their interactions, these entities are now available for writing the requirements.) The plant model (Step 1) represents what *is*, whereas the requirements (Step 2) asserts what *ought* to be.

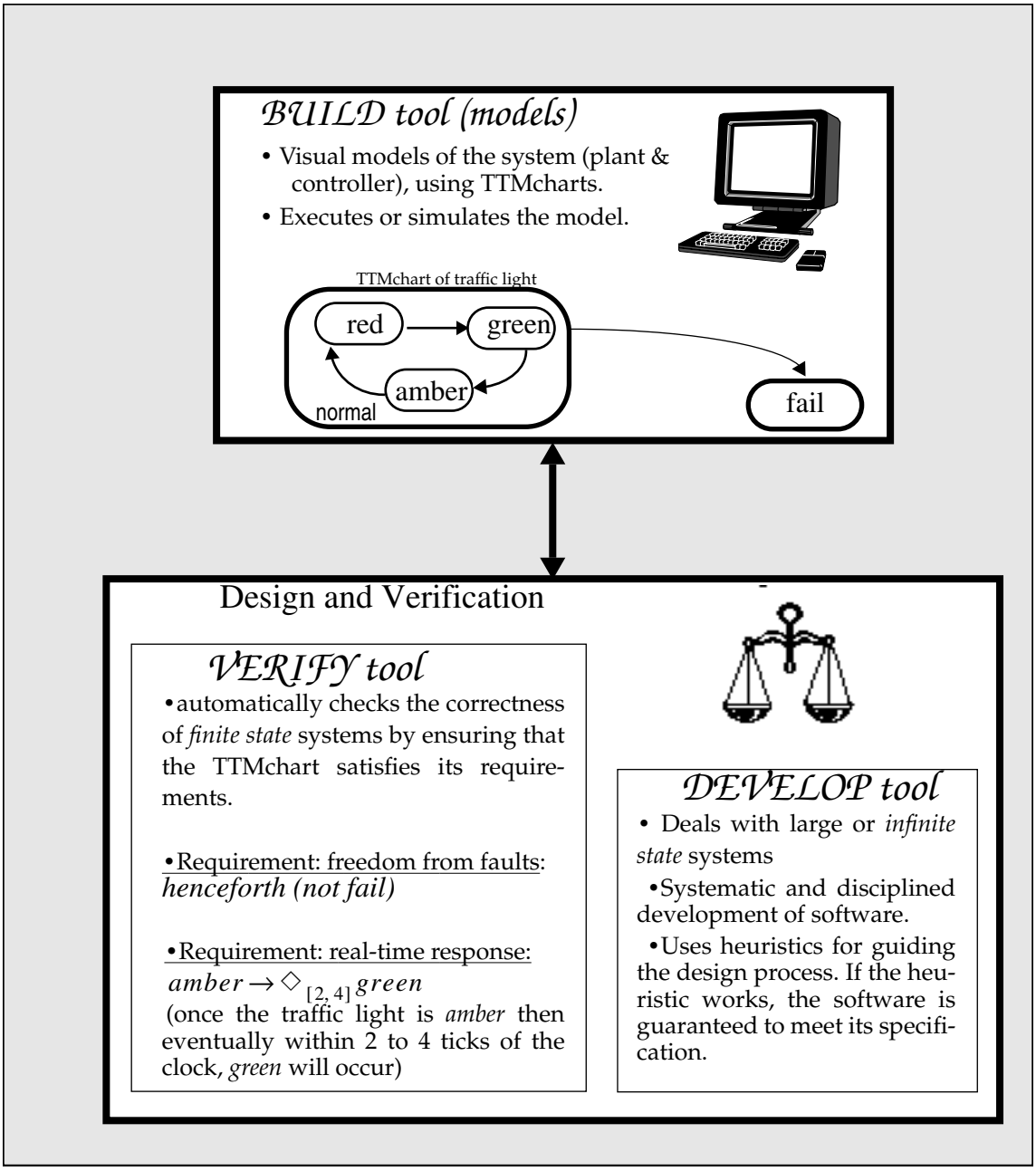
3. *Design a controller*, which when composed with the plant will satisfy the requirements (of Step 2). An example of a requirement for the reactor was already presented above: "whenever the power reaches an acceptable level then the relay should be closed within two ticks of the clock". The desired controller might be a program implemented on a microprocessor that periodically scans the power, compares it with an acceptable level, and on the basis of the comparison sends a suitable command to the relay, and then returns to the scanning phase. The controller could also be implemented directly in hardware using comparators, gates and timers. TTMs may be used to model the controller, irrespective of whether the controller is software or hardware (or a combination).

4. V*erify that the design is correct*. Check that SUD satisfies all the requirements enumerated in Step 2. Better still, use a disciplined method for obtaining the controller so that the proof of correctness is constructed hand in hand with the controller. If the development of the controller can be fully automated, then we have a synthesis procedure.

StateTime supports each of the above steps. StateTime currently consists of three integrated components BUILD, VERIFY and DEVELOP, as illustrated in Figure 2.

- BUILD — is a front end for constructing visual models (TTMcharts) of systems (plants and controllers) in a structured fashion, executing them, and refining them until a satisfactory design has been achieved. The iterative modelling/executing/refining cycle is very important as it helps to validate the model, i.e. to ensure that the model correctly reflects the actual devices of the real system. TTMcharts use the visual features (hierarchy and concurrency) of statecharts, and the communication and process composition features of CSP. A TTMchart can be executed at any point in the cycle even

FIGURE 2. StateTime = BUILD + VERIFY + DEVELOP



before it is finally fixed. TTMcharts can be composed together in parallel (AND composition) or sequentially (XOR composition). BUILD is written in ParcPlace's Objectworks/Smalltalk.

- VERIFY — is a model checker for real-time temporal logic. It computes the global state reachability graph of any finite state TTM, and then checks that the TTM satisfies the requirements (specified as formulas of RTTL). The naive interleaved model for concurrent processes cannot be used to construct the reachability graph because: (a) some states may no longer be reachable due to real-time constraints, and (b) since time is a monotonically increasing vari-

able the naively constructed reachability graph will always be infinite state. Special procedures developed for the TTM/RTTL framework must therefore be used. VERIFY is written in Quintus Prolog.

- DEVELOP — If the model is infinite state, then although a fully automated verification check cannot be performed, it may still be possible to provide semi-automated verification heuristics. If the heuristics succeed, then the model is guaranteed to satisfy its requirements. The heuristics also facilitate systematic development of the controllers from their specifications. DEVELOP uses the constraint logic programming language PrologIII [3]. Most of the reasoning can be performed by checking that certain constraints over the model variables are satisfied, without the need to perform complicated temporal logic manipulations.

StateTime thus helps the designer to obtain a thorough understanding of complex systems. Partial models can be executed to see how the system, as specified, would behave if implemented. The designer can then determine if the model truly describes what is required. Because there is a precise underlying formal framework, the model can be verified for correctness.

The VERIFY tool has been documented elsewhere [18,25], and has been used by a team at NASA to study fault-tolerant Transputer communication for flight controllers [4]. This paper will therefore focus on the design methodology, and how the BUILD and VERIFY tools support the methodology.

The theory behind the DEVELOP tool has been reported elsewhere [21,20,23]. This paper will briefly describe where this tool will be useful. A future paper is planned which will document the use of this tool in detail.

Two examples will be used to illustrate the methodology: an extended version of the real-time dining philosophers (Section 2.0), and the delayed reactor trip problem which is taken from the actual requirements documents of the Candu nuclear reactor [14]. The reactor problem describes how a watchdog can be used to verify arbitrary RTTL properties. The discussion of the delayed trip reactor presented in Section 3.0 is taken from [26].

1.3 StateTime Terminology

Here is a brief review of the terms that are used in the sequel:

1. The **TTM/RTTL** framework — the underlying mathematical theory that the StateTime toolset is based on. TTMs are Timed Transition Models. RTTL is Real-Time Temporal Logic. TTMs are mathematical models of interacting distributed processes. RTTL is a rigorous specification language for stating how the models ought to behave. Without an underlying mathematical theory, the Statetime toolset would not be able to execute systems and verify their correctness.

2. **TTMchart** — a visual representation of a system that can easily be converted into a timed transition model (TTM). Graphical notions are provided for representing states (called *activities*), transitions, concurrent processes, hierarchy,

timing and program statements (assignments). A TTM is a mathematical entity. A TTMchart is a concrete visual representation of that mathematical entity.

The terms "TTM", "TTMchart", and "chart" are often used interchangeably where it is clear from the context what is meant.

A TTMchart is one type of visual front end to TTMs. Other front ends (e.g. Petri Nets) could also have been used. The TTM computational model is general enough to support statecharts, Petri Nets, CSP and other programming languages (e.g. Ada or Occam) [16,17].

3. The **StateTime** toolset consists of three tools:
   The **BUILD** tool — a tool that provides automated support for drawing and executing TTMcharts.
   The **VERIFY** tool — a tool that automatically verifies that a *finite state* TTM satisfies an RTTL requirement.
   The **DEVELOP** tool — a tool that semi-automates the verification of *infinite state* TTMs. It also helps the designer to synthesize TTMs from RTTL specifications.

State-formulas are boolean valued expressions in the system variables. RTTL formulas are constructed from state-formulas together with special temporal logic operators such as $\Box$ (*henceforth*) and $\Diamond$ (*eventually*). Let $f, f_1, f_2 \ldots$ stand for state-formulas in the table below. The table documents some of the RTTL properties that the VERIFY tool can check.

How the property is read	Property	Definition of the property
Invariance: f_1 entails henceforth f_2.	$f_1 \Rightarrow \Box f_2$	In any reachable state s in which the state-formula f_1 holds, the formula f_2 must also hold in s and in all following states.
Real-time response: f_1 entails eventually f_2 within l to u ticks	$f_1 \Rightarrow \Diamond_{[l,u]} f_2$	In any reachable state s in which f_1 holds, f_2 must also hold in some following state s' which is at least l ticks but no more than u ticks after s.
Unless or *waiting-for:* f_1 entails f_2 waiting for f_3.	$f_1 \Rightarrow (f_2 \mathcal{W} f_3)$	If f_1 holds in any reachable state s, then in s and all following states the formula f_2 holds continuously or until the next occurrence of f_3.

For the real-time response property, the verifier is given f_1, f_2 and it returns the time bounds l and u. If it returns $u = \infty$, then the property is false.

2.0 Example 1: The real-time dining philosophers

In this section we use our methodology for designing a controller for the dining philosophers. This example can also be used to illustrate the StateTime tools, especially its facilities for representing sequential composition, concurrency, hierarchy, time bounds, data variables and synchronous interaction. In the next section, we apply the methodology to the shutdown logic taken from the actual requirements document of a nuclear reactor.

The first step in the design process is to discover and describe as much as possible about the problem domain. Informal descriptions of the system to be developed must be translated into suitable TTMcharts. The problem of the dining philosophers illustrates instances of system deadlock and process starvation resulting from the interaction of distributed processes as they attempt to access shared resources. We add to these standard problems real-time (bounded) response, and the use of data variables for controlling the behaviour of a system.

The well-known tale of the dining philosophers is due to Edsger W. Dijkstra, as retold by C.A.R Hoare [11, p75]. In ancient times, a wealthy philanthropist endowed a College to accommodate three eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking. There was also a common dining room, furnished with a circular table, surrounded by three chairs, each labelled with the name of the philosopher who was to sit on it. The names of the philosophers were philo_x where x is {a,b or c} disposed in this order anticlockwise around the table. To the left of each philosopher was laid a golden fork, and in the center of the table was placed a large bowl of spaghetti that was constantly replenished.

A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his designated chair, and picked up his left fork and plunged it into the spaghetti (it would be bad manners to pick up the right fork first, hence this was never done). The philosopher then picked up the fork to his right to help carry the spaghetti to his mouth. When he was finished eating, he would put down both forks, and continue thinking. Of course, a fork could be used by only one philosopher at a time. If the other philosopher wanted it, he just had to wait for it. The forks are called forkn where n is {1,2 or 3}.

In the above folktale, the forks represent shared resources, such as a printer or disk drive that can only be used by one process at a time. The philosophers represent the various processes in a system, e.g. the users of a computer system. The users cannot all access a shared resource at the same time. Hence there can be various problems of deadlock, and accessibility.

The top-level TTMchart in the hierarchy is college, which is the parallel composition of philosophers and forks:

```
college = philosophers || forks
philosophers = philo_a || philo_b || philo_c
forks = fork1 || fork2 || fork3
```

In the case of the dining philosophers, the plant is the college. The plant can behave in unsatisfactory ways; for example, it is possible for it to deadlock. Therefore, there is a need for a controller, which in this case will be called the porter. The job of the porter will be to control the seating arrangements of the college so that deadlock and other such problems are avoided.

The following design steps will be followed using the StateTime toolset:

1. Use the BUILD tool to model the college (the plant) as a TTMchart. See Section 2.1.

2. Validate the model of the plant by executing its chart. BUILD uses the formal semantics of TTMs to help the designer check that the model indeed captures the essential behaviour of the plant. The designer may have to iteratively return to the previous step so as to bring the model in conformance with the real world. Simulating the model may also help the designer discover unsatisfactory plant behaviour. See Section 2.2.

3. Once the designer is satisfied with the model of the plant, the next step is to write down the RTTL requirements of how the plant ought to behave. At this point, the plant will not satisfy its requirements, e.g. it may deadlock. See Section 2.3.

The TTMchart college and the RTTL requirements together provide for a complete description of the problem to be solved: "develop a controller (the porter) that when composed with the plant (the college) satisfies the RTTL requirements". The design may thus continue as follows:

4. The porter (the controller) must satisfy its RTTL specifications. The designer must provide the "interface specifications", i.e. which variables of the plant are observable by the controller, and which plant events are controllable. The controller is itself a TTMchart constructed with the help of BUILD. The controller may only access those plant variables that are observable, and control the plant only through those plant events that are controllable. See Section 2.4.

5. BUILD can then be used to compose the porter in parallel with the college to obtain: newcollege = college || porter.

6. Finally, the VERIFY tool is invoked to check that newcollege satisfies the RTTL requirements provided in Step 3 above. If the requirements are not satisfied, then the porter must be modified until all the requirements are met. To help with this step, VERIFY returns diagnostic information as to where the specification failed (e.g. in which state the specification failed to hold true). See Section 2.5.

The execution and simulation facility of BUILD is useful throughout the design cycle. Simulation may be useful in detecting problems in the plant as well as in the controller.

The temporal logic language allows requirements to be expressed in an abstract fashion for specifying the dynamic behaviour of real-time systems. The special temporal operators allow for natural and succinct expression of frequently occurring system properties. The desired behaviour is described while avoiding references to the method or the details of the implementation of the controller.

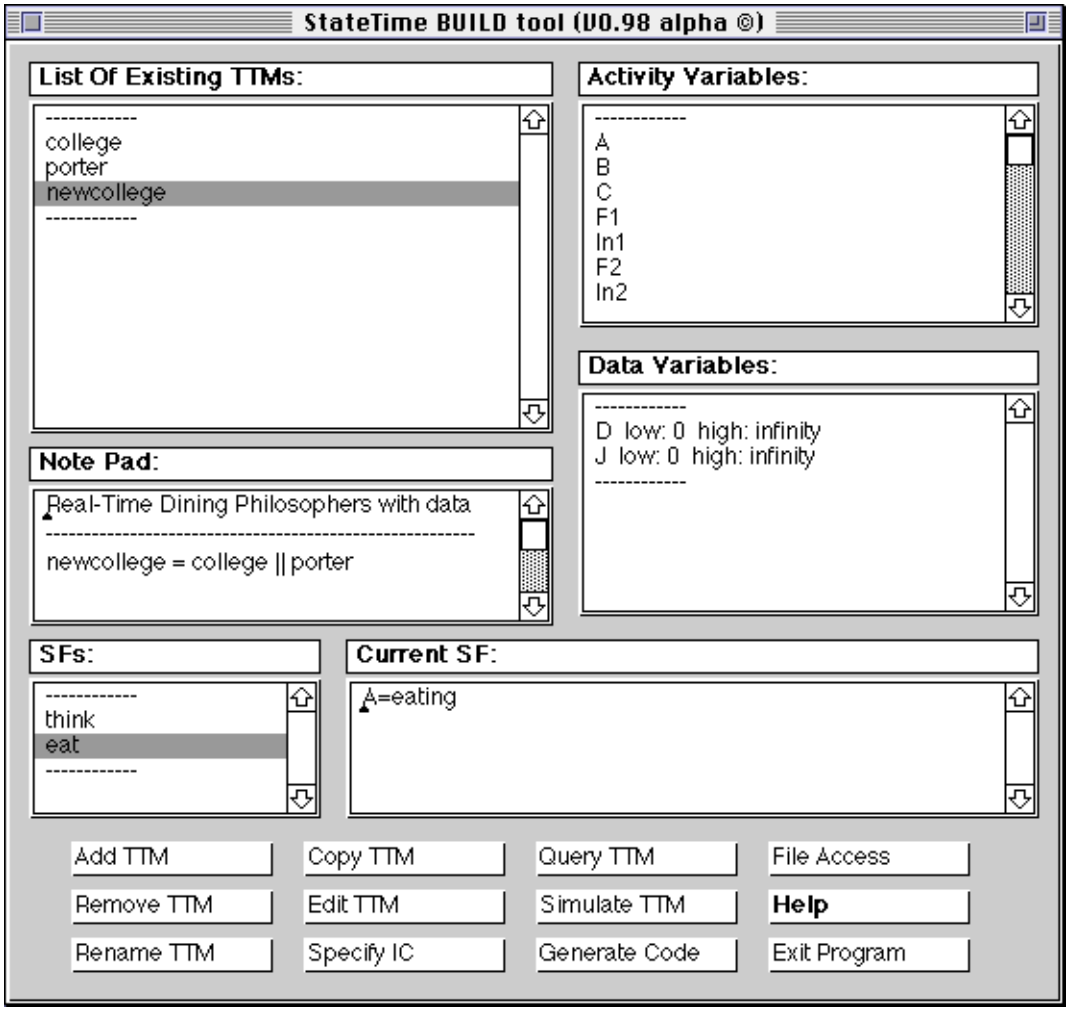
The TTMchart description of the college captures the detailed operational information of the philosophers and forks. Both TTMcharts and RTTL may be viewed as generators of executions or computations of the system, thus providing alternative characterizations of the system. Hence, RTTL could also have been used to describe the plant. However, the already existing operational nature of the

plant is more easily expressed as a TTMchart. By contrast, the as yet to be implemented controller tasks are more easily described by the less operational and more abstract RTTL requirements.

In the next subsection, the BUILD tool is used to construct a pictorial model of the college as a TTMchart. The meaning of these TTMcharts will be explained informally by example. For a complete formal treatment of TTM semantics see [19,18].

2.1 Building a model of the college

Figure 3. Main Window of the BUILD tool — list of TTMcharts and chart variables



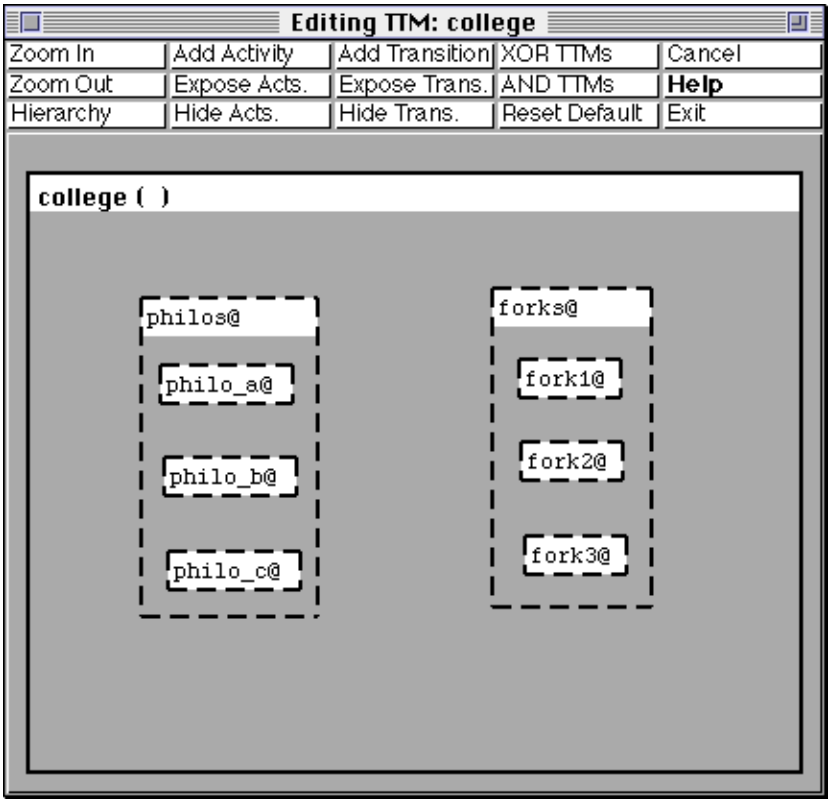
The BUILD tool is used to draw and execute TTMcharts. When the tool is first invoked, the main window shown in Figure 3 is displayed showing a list of all the available TTMcharts. Various operations on charts can be invoked by clicking the buttons towards the bottom of the main window. For example, the FileAccess operation is used to load in previously stored TTMcharts, or to store new ones to disk.

The CopyTTM and AddTTM operations may be invoked to generate new charts. The CopyTTM operation allows the designer to reuse an already existing chart (e.g.

once philo_a is created, it may be used as a template for creating the other philosophers). The AddTTM operation is used for creating a new chart.

The college chart is created by invoking the AddTTM operation. The user is prompted for the name of the chart (i.e. college), after which the user invokes the EditTTM operation, which produces an edit view of the college chart, as shown in Figure 4.

Figure 4. Edit view of college = philosophers || fork



Initially the edit view is empty. If a bottom-up approach is followed, the philosophers and forks charts would already have been created, and the AND-TTMs operation in the edit view may then be invoked to compose philosopher and forks in parallel with each other to obtain college = philosophers || forks.

Alternatively, a top-down approach may be used in which the philosopher and fork charts have not yet been created. The Add-Activity operation in Figure 4 may be invoked to insert the new sub-charts philosophers and forks. The user may then ZoomIn to these sub-charts to refine them further.

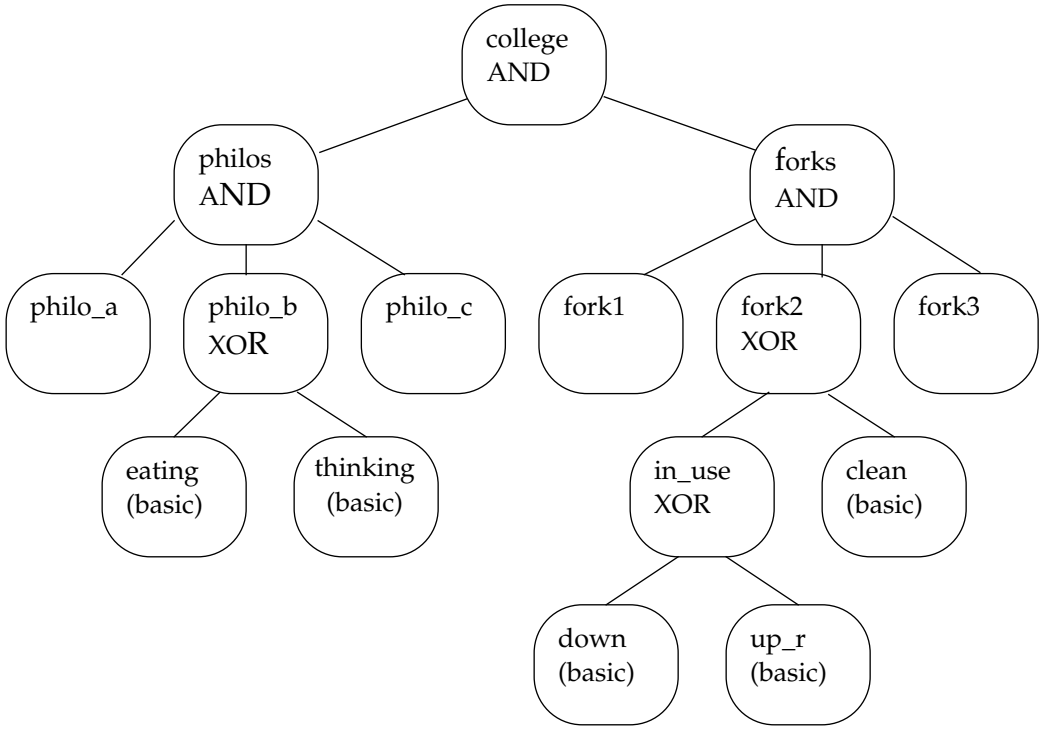
2.1.1 Root activities, ordinary activities and basic activities

All available TTMcharts are listed in the main window (Figure 3). These charts at their highest level are also called *root activities* that can be further refined into ordinary *activities*, and those activities can themselves be further refined into sub-activities. Activities with no further internal structure are called *basic activities*.

At any level of refinement, a chart can be imported from the chart list in the main window either by AND-composition (parallel composition) or by XOR-composition (sequential composition). The imported chart then becomes a sub-activity at that level.

The activity hierarchy of the primary chart college is shown in Figure 5. The user may navigate within the chart using the ZoomIn, ZoomOut and Hierarchy buttons of the edit view.

Figure 5. Part of the activity hierarchy for the college TTM



Zooming in to philosophers produces the edit view shown in Figure 6, in which philo_a and philo_b are shown in expanded format, whereas philo_c@ is collapsed. The @ sign indicates that philo_c has further internal structure, which can be explored either by expanding it in its place (if there is room) or by zooming in. Zooming in to forks and then into fork1 produces the edit view shown in Figure 7.

2.1.2 AND-composition, XOR composition and Activity Variables

AND-composition of activities is indicated by dotted rectangles. An activity which is AND-decomposed into several sub-activities, is simultaneously in all its sub-activities. Thus college has six separate simultaneously executing threads (three philosopher threads and three fork threads) as shown in the edit view of Figure 4.

XOR composition is indicated by rectangles with solid lines. An activity A which is XOR (exclusive-OR) decomposed into sub-activities A1 and A2 has the following behaviour: to be in A is to be in either A1 or A2, but not in both. For example the fork1 activity shown in Figure 7 is XOR-decomposed into sub-activities in_use and clean.

At any level of the hierarchy in the edit view, the topmost activity name and its corresponding *activity variable* is shown on a white background. For example, the activity fork1 in Figure 7 has activity variable F1.

The *type* of an activity variable is the range of sub-activities that it decomposes into, e.g. type(F1) = {in_use,clean} because the activity fork1 is XOR-decomposed

into in_use and clean. The activity in_use (with its corresponding activity variable In1) is further XOR-decomposed into the sub-activities down, up_l and up_r, so that type(In1) = {down,up_l,up_r}.

To assert that the current execution point is somewhere within the activity in_use, we write (F1=in_use) . To assert that the system is in the basic activity down of the in_use chart, we write

$$(\mathsf{F1}=\mathsf{in_use}) \wedge (\mathsf{In1}=\mathsf{down}) \qquad \text{(EQ 1)}$$

(EQ 1) is a *state-formula*, i.e. it is a boolean valued expression in the chart (activity) variables. The activity variables are thus used in state-formulas for describing activity properties.

Every activity that is XOR-decomposed into sub-activities must have a corresponding activity variable. However, activities such as college which are AND-decomposed into sub-activities do not need activity variables (see Figure 4), as to be in college is automatically to be in philosophers and forks simultaneously. These in turn are AND-decomposed into the individual forks and philosophers. The forks and philosophers (which are further XOR-decomposed into other sub-activities) have activity variables (F1,F2,F3 for the forks and A,B,C for the philosophers), which are sufficient for expressing all relevant activity properties of the college chart.

Every activity that is XOR-decomposed into sub-activities, must have one of these sub-activities designated as the *default activity*. When there is a transition from activity A1 to A2, the destination A2 is assumed to start executing in its default activity. The default activities are indicated in bold print. For example, **in_use** is the default of fork1, and **down** is the default of in_use (see Figure 7).

2.1.3 States, computations, state-formulas and RTTL formulas

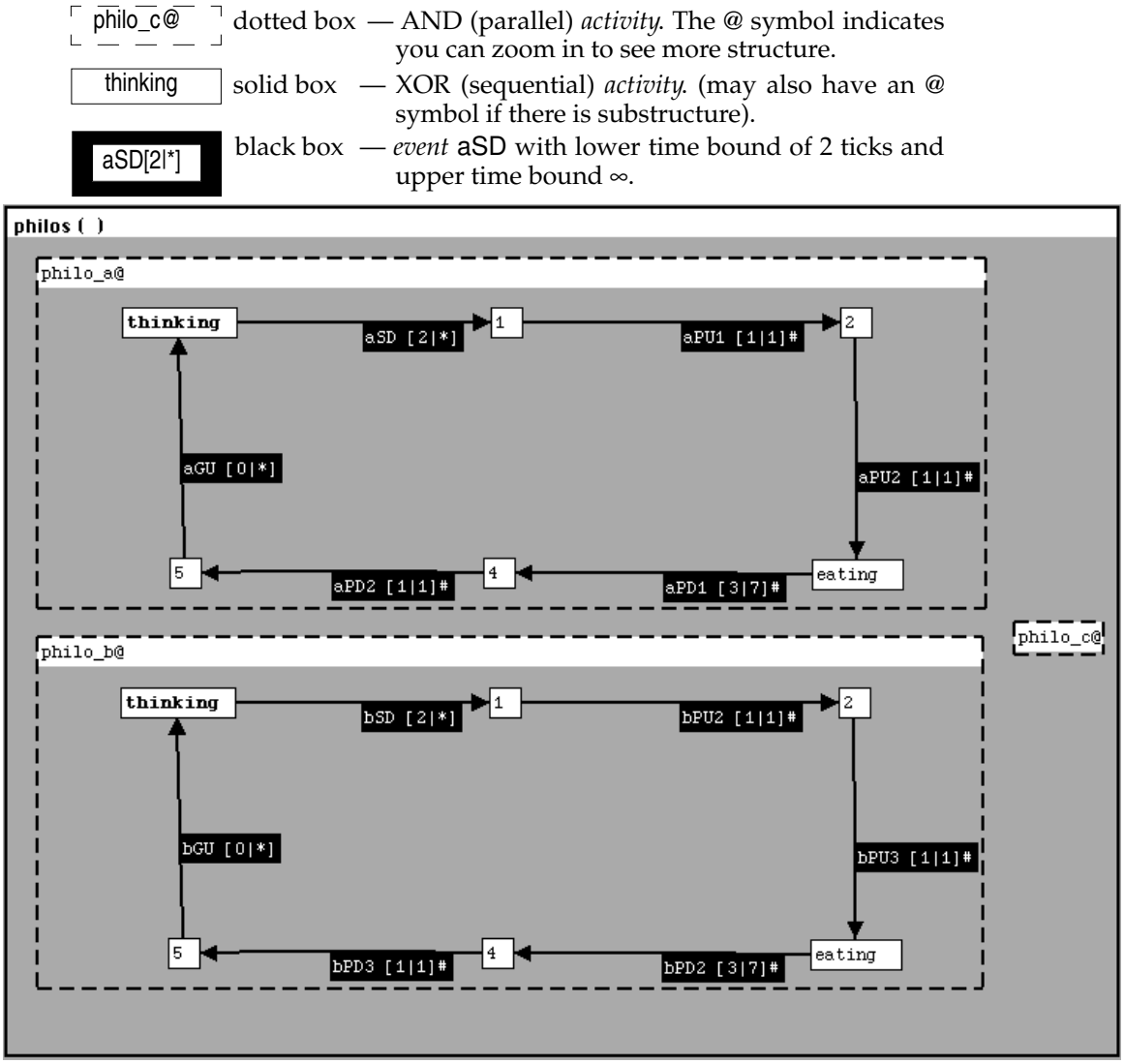
What is called a "state" for statecharts is called an "activity" in TTMcharts. The word "state" is reserved in TTMs for the global state of the chart, i.e. the assignment of values to all the variables in a given TTMchart. A state-formula such as (EQ 1) can be evaluated to true or false in such a global state s_i. A *computation* (or trajectory) of a TTMchart is a sequence of states:

$$s_0 \xrightarrow{\text{aSD}} s_1 \xrightarrow{\text{bSD}} s_2 \xrightarrow{\text{cSD}} s_3 \xrightarrow{tick} s_4 \xrightarrow{\text{aPU1}} s_5 \xrightarrow{\text{bPU2}} s_6 \xrightarrow{\text{cPU3}} s_7 \xrightarrow{tick} s_8 \longrightarrow \cdots$$

A state-formula can be interpreted as either true or false in a state. For example, the state-formula $((\mathtt{D} \geq 3) \wedge (\mathtt{A} \neq \mathtt{B})) \rightarrow (\mathtt{F1} = \mathtt{clean})$ evaluates to *true* in the state <A:thinking, B:eating, C:eating, F1:clean, F2:clean, F3:clean, In1:down, In2:down, In3:down, D:4, t:10, ε:out>.

As discussed in Section 1.3, RTTL formulas are constructed from state-formulas and temporal logic operators. State-formulas are evaluated in a single state of a computation, whereas RTTL formulas assert properties that hold over many states of the computation. For example, the RTTL formula given by $(\varepsilon = \mathsf{aSD} \wedge \mathsf{F1} = \mathsf{down}) \Rightarrow \Diamond(\mathsf{F1} = \mathsf{up_l} \vee \mathsf{F1} = \mathsf{up_r})$ asserts that if the transition aSD is taken in a state of the computation satisfying the state-formula (F1 = down) , then eventually a state of the computation is reached in which fork1 has been picked up. The distinguished *transition variable* ε has as its type the set of all transitions in the chart. The transition variable is used in state-formulas to refer to transition occurrences. Thus the RTTL formula $\Box\Diamond(\varepsilon = tick)$ asserts that there are an infinite number of clock ticks in the computation.

Figure 6. Edit view of philosophers = philo_a || philo_b || philo_c



2.1.4 Local Events, Shared Events and Transitions

An *event* in a TTMchart is indicated by drawing an arrow from a source activity to a destination activity. Multi-level events are allowed (the source and destination activities need not be at the same level in the hierarchy), provided the source and destination are not in parallel with each other.

Events are declared to be either *local* or *shared*. An event which is declared local, is hidden from all other events in the chart, and does not synchronize with them. A shared event synchronizes with all other shared events with the same name. The set of all shared events of the same name, taken together as a simultaneous interaction or rendezvous, is called a *shared transition*.

An *event* is an edge in the TTMchart. A *transition* is the corresponding theoretical entity in a timed transition model (TTM). A TTM is a 3-tuple consisting of (V, I, T) where V is the set of all variables (e.g. activity and data variables), I is the initial condition, and T is the set of transitions. Such a TTM can be considered as a generator of computations.

The transition takes into account the fact that two or more events may synchronize with each other. Such a shared transition may have any finite number of component events. If any one of the components is disabled then the composite is also disabled from being taken. Some examples will now be provided of local and shared events. The corresponding transitions together with their enabling conditions and transformation functions will also be described.

Let A1 and A2 be two activities of a chart that are in parallel with each other (e.g. philo_a and fork_1). Let $E_1\,[l_1, u_1]$ and $E_2\,[l_2, u_2]$ be events with sources in A1 and A2 respectively. Let E_1 and E_2 both be declared shared[3]. Then E_1 and E_2 taken together form the shared transition $E\,[l, u]$ which may only be taken if both E_1 and E_2 are enabled. The transition $E\,[l, u]$ must respect the time bounds of all of its components, i.e. $l = max(l_1, l_2)$ and $u = min(u_1, u_2)$.

The college events have the meanings indicated in the table below:

Local or Shared	**Event Label** (also the transition name)	**Event Meaning** x ∈ {a,b,c} and n ∈ {1,2,3}
Local events (these events also happen to be spontaneous)	xSD	philosopher x sits down
	xGU	philosopher x gets up
Shared events	xPUn	philosopher x picks up fork n
	xPDn	philosopher x puts down fork n

The two events aPD1[3,7]# in philo_a and aPD1[0,∞]# in fork_1 are declared shared. The # symbol after the event name indicates its shared status. These two events together form a shared transition also denoted aPD1 involving simultaneous action on the part of the philo_a and fork_1: the first fork is put down and simultaneously philosopher a stops eating.

Every transition has an enabling condition, transformation function and time bounds. This information is displayed by invoking Query-TTM from the main window (Figure 3). For the transition aPD1 the information displayed is:

Transition name	**Enabling Condition**	**Transformation Function**	**Lower Bound**	**Upper Bound**
aPD1	(A=eating) ∧ (F1=in_use ∧ In1=up_r)	[A:4,In1:down]	3	7

The enabling condition is the conjunction of $(\mathsf{A = eating})$, which is the activity constraints due to philo_a, and $(\mathsf{F1{=}in_use} \wedge \mathsf{In1{=}up_r})$ which is the constraint caused by fork1. Similarly, the transformation function indicates that both philo_a and fork1 experience a change in their activity variables, when the transition is taken.

A state s is a *moment of enablement* of transition aPD1 when its enabling condition becomes true (i.e. philo_a is executing in eating and fork_1 in up_r). From that moment, aPD1 will not occur until 3 clock ticks have been taken. Thereafter, provided aPD1 is still *enabled*, aPD1 must be taken in a subsequent state s_i before the 8th tick of the clock (from its moment of enablement s). When aPD1 is taken in s_i

3. E_1 and E_2 should both be called E, as it is intended that they synchronize with each other. For clarity, they are here given different labels, i.e. E_1 and E_2. The detail window of an event may be used to declare an event to be local or shared (see Figure 9 for an example of a detail window).

to its successor state s_{i+1}, the successor state is the same as s_i, except for those variables changed by the transformation function (i.e. A is assigned 4 and In1 is assigned the value down). Transitions are *atomic*. Thus, no other transitions (including tick), can occur between s_i and s_{i+1}. Any number of transitions (and at least 3 tick transitions) may be taken between s and s_i.

A transition can only be taken if it is enabled. There are three ways in which it may become disabled before being taken: (i) because of nondeterminism (e.g. in Figure 7, the event aPU1 may preempt cPU1); (ii) a transition may have a guard as will be explained later, and the guard may become disabled by the occurrence of some other transition that changes the value of the guard; (iii) the transition aPD1 may have component events one of which is disabled from participating in the transition.

2.1.5 Spontaneous events

The xSD[2,∞] and xGU[0,∞] transitions are *spontaneous* because their infinite upper time bounds indicate that they may be taken at any moment or never. This means that the philosophers are never forced to sit down or get up.

Spontaneous events are useful for modelling device failures and the like in plants. Initially, all events can start out as spontaneous events. As more timing information is gathered, either from the physics of the plant devices, or by empirical testing, so the TTMchart can be updated to reflect the new information. This frees the designer from having to commit to actual execution times prematurely.

It was not necessary to give separate names aSD, bSD and cSD for the event in which a philosopher is seated. Each event could have been given the same label SD. Since these events are all local, BUILD distinguishes them from each other (in the query view they would be called SD_1,SD_2 and SD_3 respectively). This type of approach is helpful when building a philosopher template chart, which can be copied and reused with minimal changes.

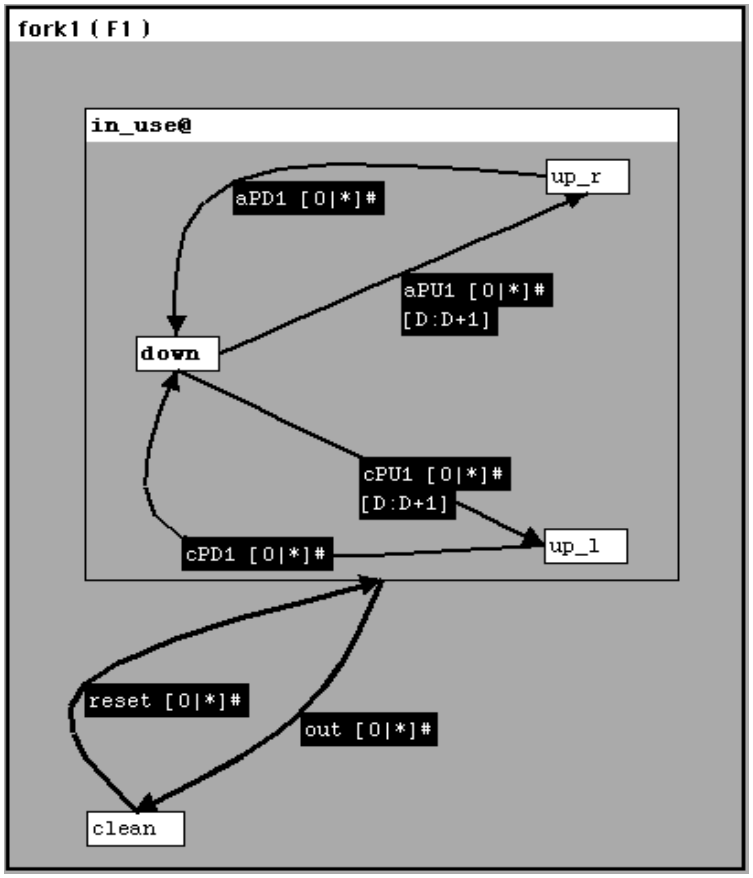
2.1.6 Grouped Events

The edit view of fork1 which is XOR decomposed into the basic activity clean and the structured activity in_use@ in Figure 7, illustrates the notion of a *grouped event*. Instead of showing three separate out events from each subactivity down, up_r and up_l to the activity clean, we instead cluster these three activities into the super activity in_use, and replace the 3 edges with a single out event whose source is the outer contour of the TTM in_use. Grouping is useful for hierarchically structuring charts and prevents the proliferation of event edges.

The out transition models a waiter who is able to grab a fork away from a philosopher, no matter what activity the fork is in. The waiter acts in this rude fashion because he wishes to keep the forks clean so as to preserve their golden shine. The event out# in each fork is shared. Either the waiter takes all three forks out of service simultaneously, or takes none.

The default behaviour of a grouped event can be changed both at its source activity and destination. The default behaviour of out at source is that it is enabled no matter where execution in subactivity in_use is (see Figure 7). The source of the out transition may be restricted (e.g. it may be taken only from the down activity). Further restrictions can be made should the activity down itself have further subactivities (and so on recursively).

Figure 7. Edit view of fork1 (with activity variable F1)



When the event reset occurs, its destination is shown as the outer border of the activity in_use, which means to the default **down** of in_use. It is possible to change the default destination of out, and to select some other subactivity of in_use as the destination. If that subactivity has structure, then its default becomes the new destination unless it too is changed (and so on recursively down the hierarchy).

If an event E is displayed in an edit view with a left angle bracket <E, then the source behaviour has been changed from the default. The right angle bracket indicates that the destination has been changed from its default. The table below summarizes the visual indicators used in BUILD:

Type	Symbol	Meaning
Structured activity: @	act@	Activity act has internal substructure. Otherwise the activity is basic.
Shared event: #	evt#	The event evt is a shared event. Otherwise the event is local.
The default behaviour of a grouped event is changed: < and >	<evt	The source of evt is changed
	evt>	The destination of evt is changed
	<evt>	both source and destination are changed.

Each event has an associated detail window with all pertinent information, including time bounds, guards and transformation functions. The detail window is also used to declare whether the event is shared, and to change the default

behaviour at source and destination (see Figure 9 for an example of a detail window).

2.1.7 Data variables, Assignments and Guards

In addition to the special transition variable (ε) and the activity variables (e.g. A, F1, In1), a chart may also have any finite number of *data variables*. The current version of BUILD allows data variables to have an integer type with all the usual operators (+, -, *, =, <,>) as well as mod (%) and integer division (/). A data variable is declared in the main window (see Figure 3).

The data variable D for the college counts the cumulative number of times that all forks have been used. It is incremented by one every time a fork is picked up. This accounts for the assignment [D:D+1] attached to the transformation function of the event aPU1 and cPU1 (in Figure 7). In this case, D is a *shared variable* that may be accessed by all events in the chart.

It is possible for an event to have simultaneous assignments as in $[V_1{:}e_1, V_2{:}e_2, \ldots, V_i{:}e_i]$, where each variable V_i is assigned an integer expression e_i. Events in the reactor example have simultaneous assignments (see Figure 17).

Events may have *guards*, which are state-formulas in the activity and data variables. For example, the event aSD in Figure 9 has a guard $(J < 2 \wedge D < 10)$. The enabling condition of a transition corresponding to an event with a guard is just the conjunction of the guard and the activity constraints. For example, the enabling condition of aSD is $(J < 2 \wedge D < 10) \wedge (Portr = seat_philos \wedge Sa = 0)$. The variables J, D are data variables, and Portr, Sa are the activity variables of porter and seat_a respectively (see Figure 9).

Since shared variables are allowed its is possible for unsafe race conditions to occur. It will be up to the verifier to check that the shared variables are used in a safe manner by checking the model for the relevant safety and liveness properties.

TTMcharts extend the notion of a finite state machine in four ways: hierarchy (subactivities and grouped events), concurrency (AND-composition), interaction (shared events and variables), and timing (lower and upper time bounds).

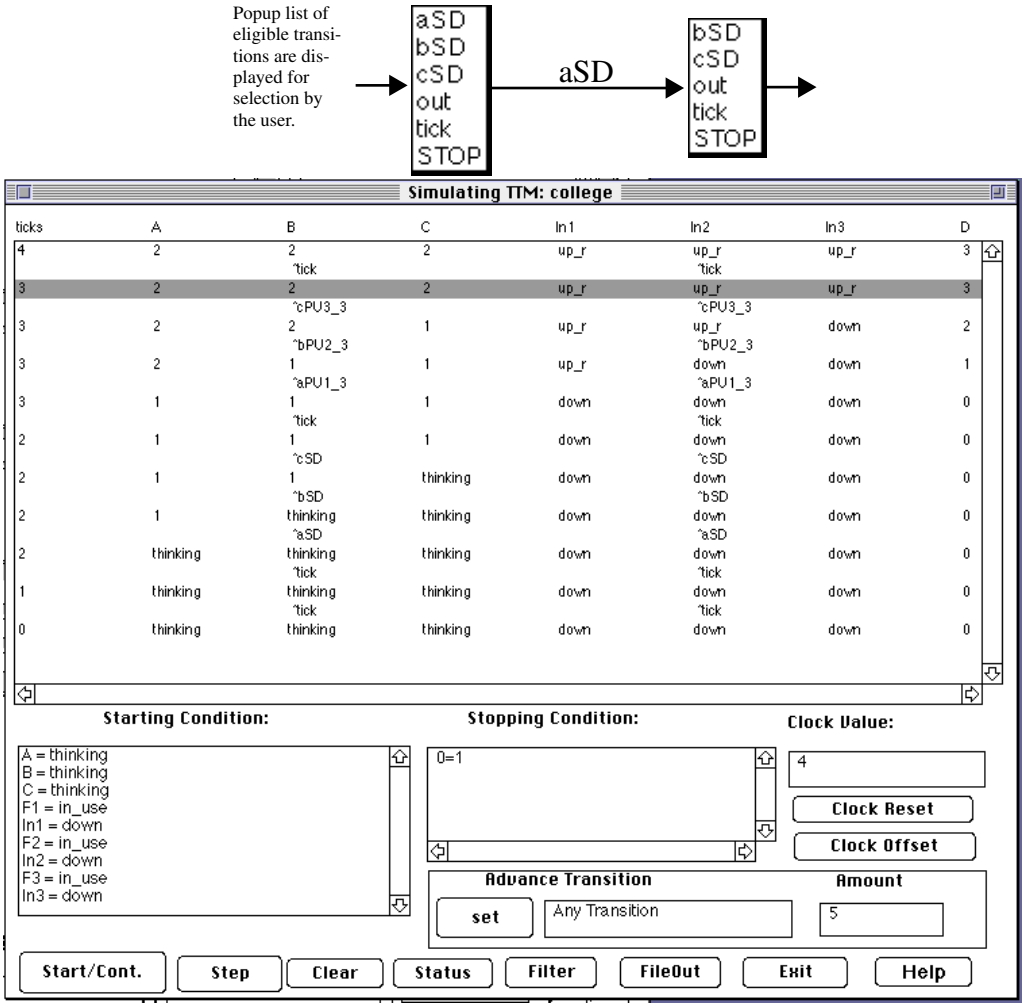
The data variable D in college can be incremented unboundedly which means that college has an infinite number of reachable states. The finite state verifier can therefore not be used to check any property. However, the execution facility may be used to simulate computations. Once the porter is composed with the college (see Section 2.4), the system becomes finite state and is amenable to verification.

2.2 Using BUILD to validate the model by executing it

In the previous section, BUILD was used to construct the TTMchart college, which is a model of the plant. There is no formal way to prove that a model truly represents the real plant. However, the simulation facility of BUILD can be used to execute various scenarios, thus allowing the designer to explore the behaviour of the model. Simulation can thereby increase the confidence of the designer that the specified model is adequate.

In the main menu of BUILD (see Figure 3), the user can invoke the SimulateTTM operation on the college chart. Figure 8 illustrates the simulation mode. The user sets up the starting conditions, the desired stopping conditions, and then starts the simulation by pressing the Start/Continue button. If the system is deterministic, then a predetermined number of steps (selected by the user) is executed with each

Figure 8. Simulation: a computation of the chart college showing deadlock



intermediate state (value of all activity and data variables) displayed on the screen. If there is more than one eligible transition (the nondeterministic case) then the user is prompted with a pop-up list of all eligible transitions at that step. The user selects which eligible transition is taken and the successor state is displayed. The simulator will only allow those executions that satisfy the semantic requirements for a legal computation (described informally in Section 2.1 and formally in [19]).

Figure 8 displays a computation in which the philosophers all get hungry at the same time. They all sit down, pick up the fork to their left, and then reach out for the other fork — which isn't there. In this situation, partial deadlock occurs (the philosophers cannot progress to the eating activity). A tick of the clock or the out transition may still be taken. Eventually, a point may be reached where all that can happen is a tick of the clock, in which case there is total deadlock.

Not all variables need necessarily be displayed in the simulate window. The user may select which variables to display by invoking the Filter operation. This allows the user to examine reduced behaviours, thereby focussing only on those entities of direct interest at the time of the simulation.

2.3 Specify the RTTL plant requirements

The designer must next specify how the plant ought to behave. For example, there should be no system deadlock, nor should it be possible for any one philosopher to starve forever. The specifications will be written in the real-time temporal logic specification language RTTL.

The following requirements must be *valid*, i.e. true in all legal computations of the TTMchart college:

[R1] Deadlock freedom: $\Box$ (enabled) .

Henceforth, in every state of every computation of the college, there must be at least one event (other than tick) which is enabled. The predicate enabled is a built-in state-formula of the verifier. It is the disjunction of the enabling conditions of all the college transitions (except for the tick transition).

[R2] Limited cumulative forks usage: $\Box$ (D $\leq$ 10) .

Henceforth the cumulative total of forks picked up, as represented by the value of the variable D, must always be 10 or less.

The requirement [R2] can easily be satisfied by preventing the philosophers from picking up any more forks (i.e. the philosophers can be immobilized), after the count of ten is reached. However, we ought to ensure that any philosopher who wants to eat always gets to eat. We therefore further specify that:

[R3] Bounded real-time response: (A = 1) $\Rightarrow \Diamond_{[1, 15]}$ (A = eating) .

Every time a state is reached in any computation of the college, in which philosopher a is in activity 1, then *eventually within* 1 to 15 ticks inclusive from that state, the philosopher must reach a later state in which philosopher a is eating. Thus, once the philosopher sits down at the table, he will eventually get to eat within the stated time bounds. Similar specifications can be provided for the other philosophers.

The requirement specified by (A = thinking) $\Rightarrow \Diamond_{[1, 15]}$ (A = eating) will *not* be valid, because the transition aSD is a spontaneous event. There is no guarantee that a philosopher will ever leave the thinking activity. Therefore, there are computations of the college in which the philosopher may choose to think forever. Consequently, the above specification is invalid.

[R4] Non-interruption: (ε = aSD) $\Rightarrow$ ($\varepsilon \neq$ out $\wedge$ D $\leq$ 12) $\mathcal{W}$ (ε = aGU) ,

asserts that every time the event aSD occurs in any state of a computation, then from that state and onwards, the event out will not occur and the count of forks lifted up will not exceed 12, *unless* the event aGU occurs.

We may think of the critical region of a philosopher as comprising the activities: 1, 2, eating, 4 and 5 (see Figure 6). The requirement [R4] asserts that once a philosopher enters this critical region, he is not interrupted by the porter until he gets up. While execution is in the critical region the fork count will never exceed 12.

2.4 Design a controller — the porter

The porter (the controller) must now be designed so as to ensure that the college (the plant) behaves according to the specified requirements [R1] to [R4]. The important issue is not how the porter behaves in isolation, but how the college

behaves once the porter is in place (the "closed-loop" system"). Hence, the requirements all focus on plant entities (the fork count D, philosopher transitions such as aSD and philosopher activities such as eating).

2.4.1 Control Interface Specifications

In order to design the controller, it is necessary to know the *control interface specifications*, i.e. which entities in the plant can be observed and which entities can be controlled.

For the porter, assume that the variable D (the cumulative fork count) is *observable*, and can also be reset to zero. This means that the porter has "read-only" access to the college data variable D (as well as the ability to reset D). All other plant variables are not accessible to the controller.

The *controllable* plant events for each philosopher x are: xSD, xGU, and out. This means that there is an interlock (or some other such mechanism) through which the porter controls these college events. The controllable events include the seating arrangements for the philosophers and the event needed for taking the forks out of service. The controllable events are modelled by considering them to be shared between the porter and the college. All other plant events are uncontrollable and are therefore local to college. The controller is allowed to have its own local events and data variables.

2.4.2 Designing the controller

The chart for the porter is shown in Figure 9. The porter is composed in parallel with the college to obtain: newcollege = porter || college. The default activity of the porter is seat_philos, which is itself refined into three sub-activities as shown in the figure.

The behaviour at the source activity of the event out is changed from the default as shown in the detail window of Figure 9. The out event is constrained to be taken only when the state-formula

$$(\mathsf{Port} = \mathsf{seat_philos}) \wedge (\mathsf{Sa} = 0 \wedge \mathsf{Sb} = 0 \wedge \mathsf{Sc} = 0)$$

holds. The out event is given an upper time bound of 6 to force its occurrence when the count D reaches 10. In practice, the upper time bound would have to be determined by the physics of the interlock mechanism.

A new local data variable J is used by the sub-activities of seat_philos to ensure that no more than two philosophers are seated at one time. A philosopher may only sit down if there is no more than one philosopher already at the table (J<2), and also it is the case that the count of forks used is less than ten (D<10). This prevents the out event from occurring while the porter is in its critical region, as the out event can only occur after the count reaches 10.

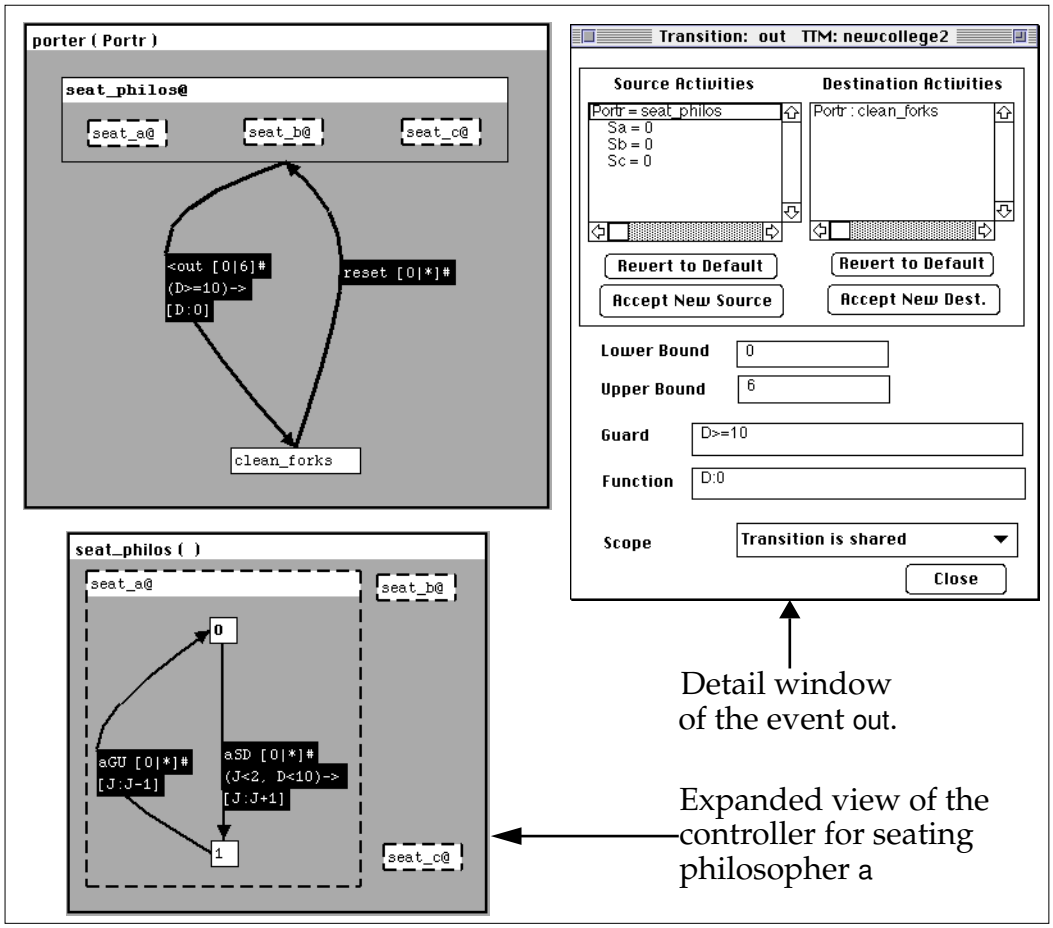
2.5 VERIFY — submit the design for verification

When the verifier is applied to the newcollege of Figure 9, the requirements [R1], [R3] and [R4] are shown to be correct (see the transcript in Figure 10).

Requirement [R2] fails to hold as shown in the following VERIFY transcript:

```
%%% Check [R2] - limited cumulative forks usage
| ?- henceForth(newcollege, initial, dle10, Answer, State).
```

Figure 9. Design of the porter controller



Answer=false
at State= [A:thinking, B:2, C:eating, F1:in_use, In1:up_l, F2:in_use, In2:up_r, F3:in_use, In3:up_r, Portr:seat_philos, Sa:0, Sb:1, Sc:1, **D:11**, J:2]

The value of the fork counter can be greater than 10 (i.e. in the failing state returned by the verifier, D has the value 11). VERIFY also returns further failed states with values of 12 for D. The guard D<10 of the xSD events must be changed to D<8 and the guard on the event out to D>=8 (see Figure 9) in order to meet requirement [R2].

The time performance of the verifier is shown in Table 1 below. The complete check for all requirements takes less than 12 minutes.

Applying Hoare's computation [11, p79] for the potential size of the state-space of newcollege, we have that the number of activities in each philosopher is 6, and the number of activities in each fork is 4. The variable D is unbounded so that the state-space is also unbounded. However, the effect of the porter is to keep an upper limit of ten on D. Since there are three philosophers and forks respectively, the size of the college is $6^3 \times 4^3 \times 10$, or 138,240 states.

Since the events of the porter is a subset of the college events, the newcollege does not have any more states than the original college. Since in nearly every state there are two or more possible events, the number of traces that must be examined will exceed two raised to the power of 138,240, or more than $2.4 \times 10^{41,614}$

TABLE 1. Performance of VERIFY on a Sun 10/20

Operation	Time (in minutes)
Generate Reachability Graph(1890 states and 3487 edges)	5.03
Requirement [R1] — deadlock detection	0.29
Requirement [R2] — detect failure	0.26
Requirement [R3] — real-time response	0.88
Requirement [R4] — no-interruption	4.67
Total	**11.13 minutes**

FIGURE 10. VERIFY transcript for requirements R1, R3, R4 of newcollege2

```
| ?- rg2(newcollege2). %%% Generate reachability graph
yes
| ?- statistics(newcollege2).
  Statistics for TTM:  newcollege
            Unique States =  392
            Total States     = 1890
            Total Edges      = 3487
yes
%%% now check requirement [R1] - freedom from deadlock
| ?- henceForth(newcollege, initial, enabled, Answer, State).
   Answer=true at State=All
yes
%%% Now check [R3] - real-time response (A=1) -> <>(A=eating)
| ?- rtrfRG2(newcollege2, 1, eating, Lo, Hi).
   Succeeded with minimum and maximum bounds of:
   Lo = 1,
   Hi = 11
yes
%%% Check [R4] - no interruption
| ?- unless(newcollege2, 1, 2, 3, Answer, State).
Answer=True at State=All
```

traces. There is no hope that a computer will be able to explore all these possibilities. Hoare therefore concludes that the designer will have to provide a manual proof of correctness.

However, the above complexity computation is a worst case analysis that does not apply in this instance to newcollege. In fact, if all the events are given a lower time bound of zero and an upper time bound of infinity, then there are only 392 reachable states. The actual time bounds have an amplifying effect that increase the size of the reachability graph to 1890 states (some states have to be reduplicated to capture the time constraints). Furthermore, each of the properties [R1] to [R4] can be checked in time *linear* in the size of the reachability graph. The system newcollege can be proven correct by VERIFY in under 12 minutes (and probably significantly faster with more efficient code).

2.6 DEVELOP — synthesizing controllers

The previous section appears to indicate that the correct controller was obtained in one or two steps. This is not the case. In actual fact, design is an iterative procedure, and the initial design was shown to be incorrect by the verifier. When the verifier showed that the initial design was incorrect, it also provided debugging information (e.g. states or computations for which the requirements failed to hold). With the help of this diagnostic information, it was determined that the guards of the porter events were incorrect.

In general, whenever VERIFY discovers a problem there are at least four possible sources of error: (a) the model of the plant may be incorrect, (b) the controller design may be wrong, (c) the interface specifications may be incorrect, or (d) the requirements may be incorrectly stated. VERIFY therefore provides diagnostic information to help the user track down the source of the error. In [22], a modelling error in the college (i.e. the plant) is discussed.

It would be nice if there was some way to systematically develop the controller from the specifications [R1] to [R4], that would get the guards right the first time round.

Such a systematic design procedure is provided in [21]. Procedures are provided, which if they terminate, are guaranteed to provide a satisfying controller. For infinite state systems, there can be no guarantee of termination. Hence, at best this method is semi-automatic and requires human interaction. Nevertheless, with the help of the human designer, systems with an infinite number of states can often be treated effectively.

The procedures require the ability to compute weakest preconditions of transitions. A tool called DEVELOP is available for automating the procedures. DEVELOP is based on the constraint logic notions provided in [20], but has been updated to use the language PrologIII [3].

The design procedure for developing controllers uses proof diagrams and weakest preconditions. A proof diagram is an abstract view of a state reachability graph. It is not confined to finite state systems because a node in the proof diagram is a state-formula that can characterize a possibly infinite set of states. The proof diagram contains the intuition of system executions without the distracting proliferation of states. Most of the reasoning takes place in the ordinary predicate calculus, with temporal or real-time reasoning introduced only where absolutely needed.

The design procedure uses the constraint satisfaction mechanisms of PrologIII to compute weakest preconditions. In the finite state case the design procedures are guaranteed to terminate. However, in the infinite state case, additional heuristics must be applied to obtain a suitable design.

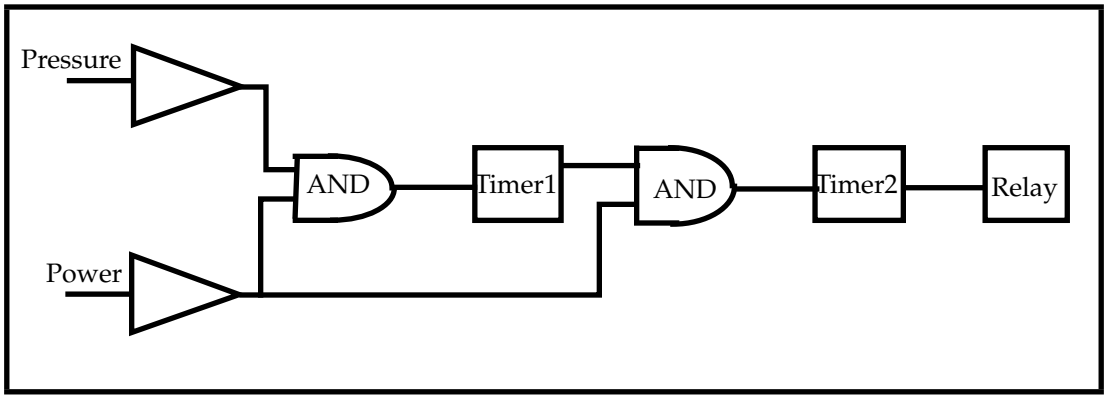
The main advantage of using DEVELOP is that it can be applied to infinite state systems. However, DEVELOP also requires much greater understanding of the TTM/RTTL framework in order for it to be used. In contrast, much of the complexity of the formal methods is hidden from the user in the VERIFY tool. Although VERIFY can only be applied to systems with finite state spaces, it is completely automatic.

3.0 Example 2: The delayed reactor trip (DRT) problem.

The delayed reactor trip (DRT) problem was first described by Lawford in [14]. It is small yet non-trivial to formally verify. Lawford developed behaviour preserving transformations for TTMs with which he was able to discover a flaw in the proposed design. However, his theory cannot be automated as no set of transformations is complete for proving observation equivalence between the actual implementation and its abstract specification. We will analyze the problem from a temporal logic perspective (RTTL), and will attempt to use completely automated verification procedures (VERIFY) to check the correctness of the implementation. The discussion below is taken from [26].

The delayed reactor trip for the CANDU nuclear reactors is currently implemented in hardware using timers, comparators and logic gates as shown in Figure 11. The new DRT system is to be implemented in future on a microproces-

FIGURE 11. Analog implementation of the delay relay trip system DRT (the "controller").



sor system. Digital control systems provide cost savings and flexibility over the hardware implementation. However, the question now is whether the new microprocessor based software controller satisfies the same specifications as the old hardware implementation.

The hardware version of the controller implements the following informal requirements:

[R1] When the power and pressure of the reactor exceed acceptable safety limits, the comparators which feed in to the first AND gate cause Timer1 to start, which times out after 3 seconds and sends a message to one of the inputs of the second AND gate indicating that the time-out has occurred. If after this first time-out the power is still greater than its safety limit, then the relay is tripped (opened), and Timer2 starts. The relay must remain open until Timer2 times out which happens after 2 seconds.

R1 ensures that the relay is opened and remains open for two seconds thus shutting down the nuclear reactor in a timely fashion. If the controller fails to shut down the reactor properly, then catastrophic results might follow including danger to life. Conversely, each time the reactor is improperly shut down, the utility operating the reactor loses money because it must bring additional fossil fuel generating stations on line to meet demand. The next informal specification R2 states:

[R2] If the power reaches an acceptable level then the relay should be closed as soon as possible (thus allowing the reactor to operate once more).

A final requirement that is *implicit* in the hardware specification, but must be *explicitly* stated for the software version is:

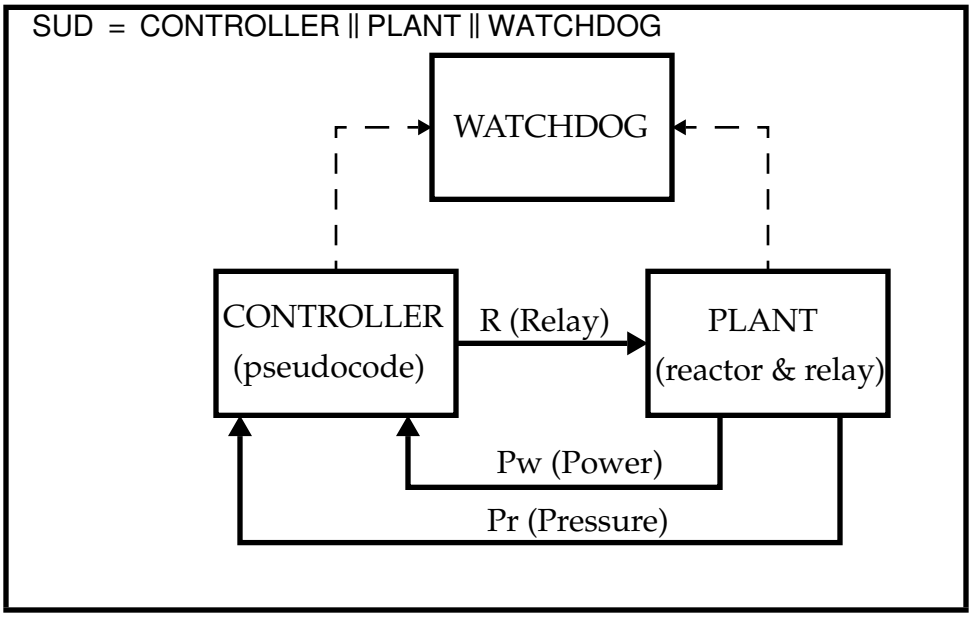
[R3] The controller should never deadlock. (For example, if after the power and pressure have exceeded their critical values, and the system has waited 3 seconds to check the power level again, if the power is below its critical limit, then the system must reset and go back to monitoring its inputs.)

In the actual DRT, there are three identical systems running in parallel with the final decision on when to shut down the reactor implemented on a majority rule basis.

It is possible to try to analyze the complete system of three concurrent microprocessors using the TTM/RTTL approach. However, it is preferable to start by first checking that each individual processor on its own achieves proper control. It is important in general to verify components before proceeding to the larger picture. In addition to "theoretical correctness", this has important practical ramifications. Larger systems have greater state spaces to explore that may be beyond the current limits of automated verification. If a component can be verified to be correct in all its detail, then a reduced order model of it may be used when checking the component in the broader context, thus reducing exponential explosion of states.

The new DRT software controller is to be implemented on a microprocessor system with a cycle time of 100ms. The software controller samples the inputs and passes through a block of control code every 0.1 seconds. It is assumed that the input signals have been properly filtered and that the sampling rate is sufficiently fast to ensure adequate control.

FIGURE 12. Relationship between controller, plant and watchdog of SUD



The proposed pseudocode taken from the DRT requirements documents is shown in Figure 13. The code mimics the original analog implementation by using integer variables c1 and c2 in place of Timer1 and Timer2 respectively. The program also makes use of the variables *Pr* (Pressure), *Pw* (Power) and *R* (Relay) for the sampled inputs (Pressure and Power) and output (Relay) of the controller.

The DRT system under design (SUD) consists of the parallel composition of three components, i.e.

SUD = CONTROLLER || PLANT || WATCHDOG

The relationship between plant and controller is shown in Figure 12. The CONTROLLER is the abovementioned pseudocode for the microprocessor displayed in Figure 13. The PLANT is the environment in which the controller operates, i.e. the nuclear reactor which generates the power and pressure, and the relay that opens or closes depending on the value of the relay variable *R* set by the controller. The WATCHDOG is a non-invasive observer of the plant and controller variables (i.e. it has access to all the system variables but does not in any way change or control them). The WATCHDOG is used for verification only and is not part of the actual implementation (this will be explained further in the sequel).

3.1 Modelling the plant

We follow here the design methodology proposed in Section 1.2. The first step uses the BUILD tool to model the plant.

The power and pressure variables are assumed to be filtered. This will be modelled by allowing them to be updated every two ticks of the clock, where one tick of the clock is 100ms. The model of the plant is shown in Figure 14.

The update activity of Figure 14 is described in more detail in Figure 15, from which it follows that update = power_update || pressure_update . The default subactivity of power_update is **0**, i.e. each time power_update is invoked it is assumed to start in subactivity **0**, from which the power variable *Pw* can be assigned the integer value 0 (meaning the power is within an acceptable range) or 1 (meaning that the power is too high).

The event powerHi[0,0] goes from subactivity **0** to subactivity 1, at the same time assigning the integer value 1 to the variable *Pw*. The lower and upper time bounds are both zero indicating that powerHi is taken before the next tick of the clock. Similarly, the pressure_update activity describes how the pressure variable *Pr* is updated.

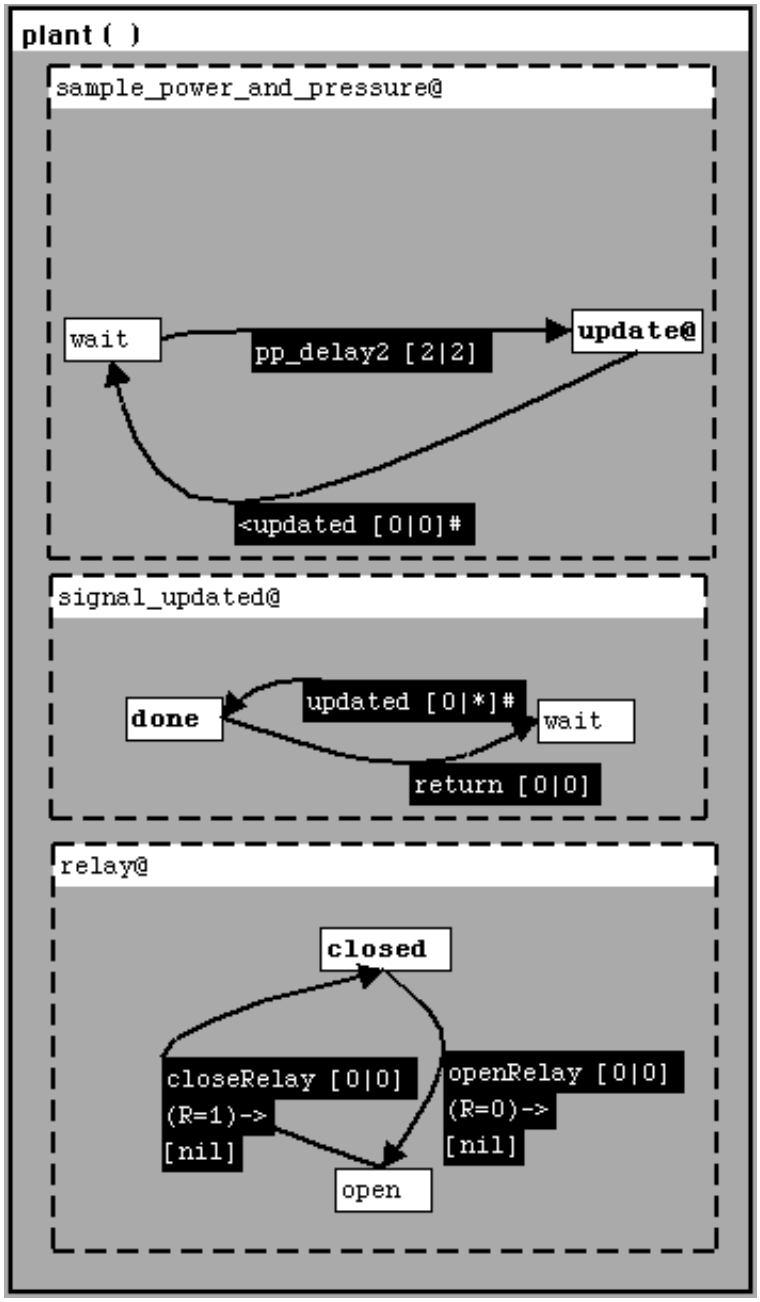
The update activity (of Figure 15) is used to build the super activity labelled sample_power_and_pressure as shown in Figure 14. To be in the XOR activity sample_power_and_pressure is to be either in wait or update but not both simultaneously.

The event updated[0,0] exits from the outer contour of the structured activity update. The default meaning is that no matter where in the structured activity update the two threads of control currently resides, the event updated[0,0] is eligible to occur. However, a special detail view of updated[0,0] can be invoked in which the default behaviour can be changed. In this case, the event is changed so that it is eligible to occur only when both power_update and pressure_update are in their subactivity 1 as shown in Figure 16 (hence the event updated[0,0] occurs immediately

FIGURE 13. Pseudocode for the DRT taken from Lawford's report [14].

```
If Pressure ≥ DSP then
     If Power ≥ PT then
          If counter c1 is reset then
               If counter c2 is reset then
                    increment c1                ]Transition : μ1
               Else
                    If counter c2 has timed out then
                         reset c2               ]Transition : γ
                    Else
                         increment c2           ]Transition : μ2
                         open Relay             ⌋
                    Endif
               Endif
          Else
               If counter c1 has timed out then
                    open Relay                  |
                    reset c1                    |Transition : α
                    increment c2                ⌋
               Else
                    increment c1                ]Transition : μ1
               Endif
          Endif
     Endif
Else
     If counter c1 is reset then
          If counter c2 is reset then
               close Relay                      ]Transition : β
          Else
               If counter c2 has timed out then
                    close Relay                 ]Transition : ρ2
                    reset c2                    ⌋
               Else
                    increment c2                ]Transition : μ2
                    open Relay                  ⌋
               Endif
          Endif
     Else
          If counter c1 has timed out then
               reset c1                         ]Transition : ρ1
          Else
               increment c1                     ]Transition : μ1
          Endif
    Endif
Endif
```

FIGURE 14. DTR plant is the AND-composition of the relay, power and pressure updates.



after the variables are both sampled). By virtue of its time bounds, the event pp_delay2 must wait for two clock ticks before it is taken.

The plant is an AND-composition given by

$$\text{plant} = \text{relay} \parallel \text{sample_power_and_pressure} \parallel \text{signal_update}\,.$$

The activity signal_update in Figure 14 is really part of the watchdog (it is shown within the plant for convenience). It is used for writing requirements (see later) that record when there are still two ticks to the next power and pressure update. the activity variable of signal_update is S, with $type(S) = \{\text{done}, \text{wait}\}$. To express

FIGURE 15. The "update" activity is AND-decomposed into the update for the power variable *Pw* and the pressure variable *Pr*.

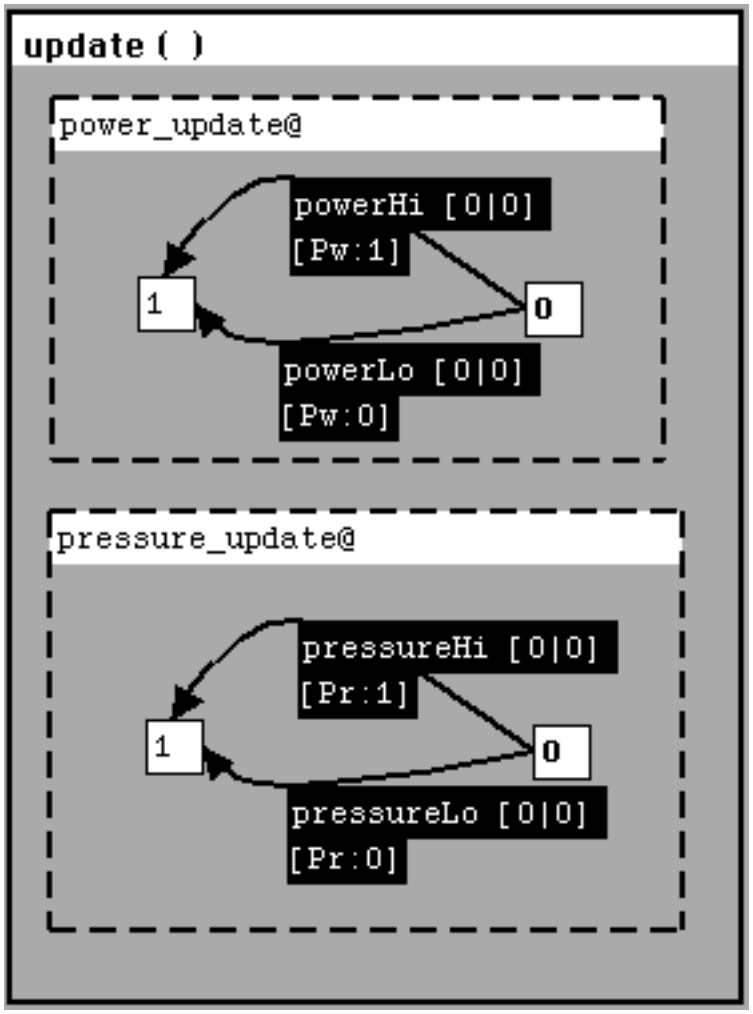
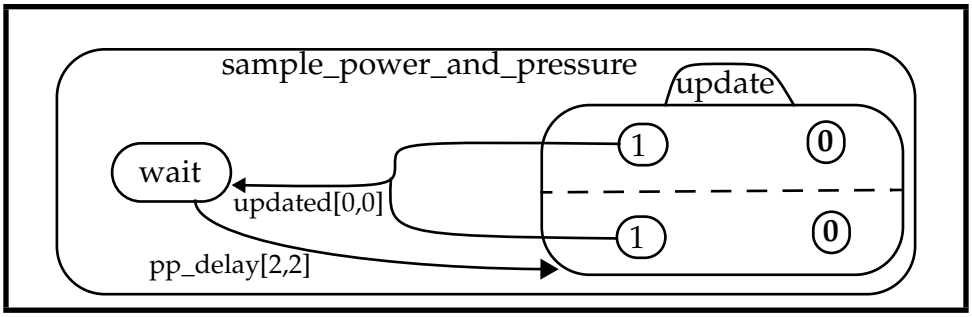


FIGURE 16. Changes from the default behaviour for the grouped event updated



the fact that the plant is in the activity done, we may write $(S = \text{done})$, which is true whenever the next update to the power and pressure variables is exactly in two ticks of the clock.

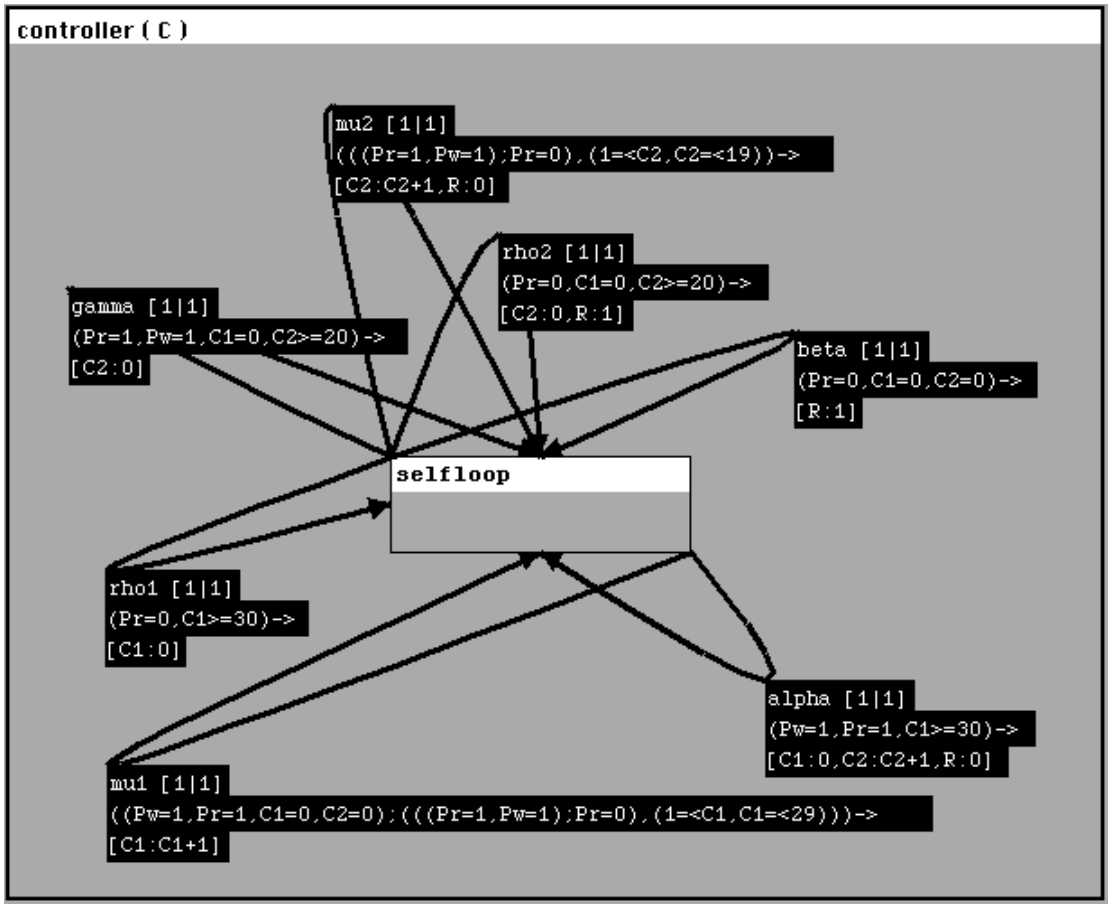
The two component events updated[0,∞]# in the signal_update activity and updated[0,0]# in the concurrent activity sample_power_and_pressure are shared. Hence

they block until they synchronize with each other. The time bounds of the resulting shared transition is [0,0]. The component update event in the activity sample_power_and_pressure may be thought of as a "forcing event" that constrains its spontaneous component update event in signal_update to occur immediately.

3.2 The software controller

Having modelled the plant of Figure 12, the next step is to obtain a TTMchart representation of the controller. The pseudocode of Figure 13 can be represented by the chart controller shown in Figure 17.

FIGURE 17. Faulty controller based on the proposed pseudocode for the microprocessor



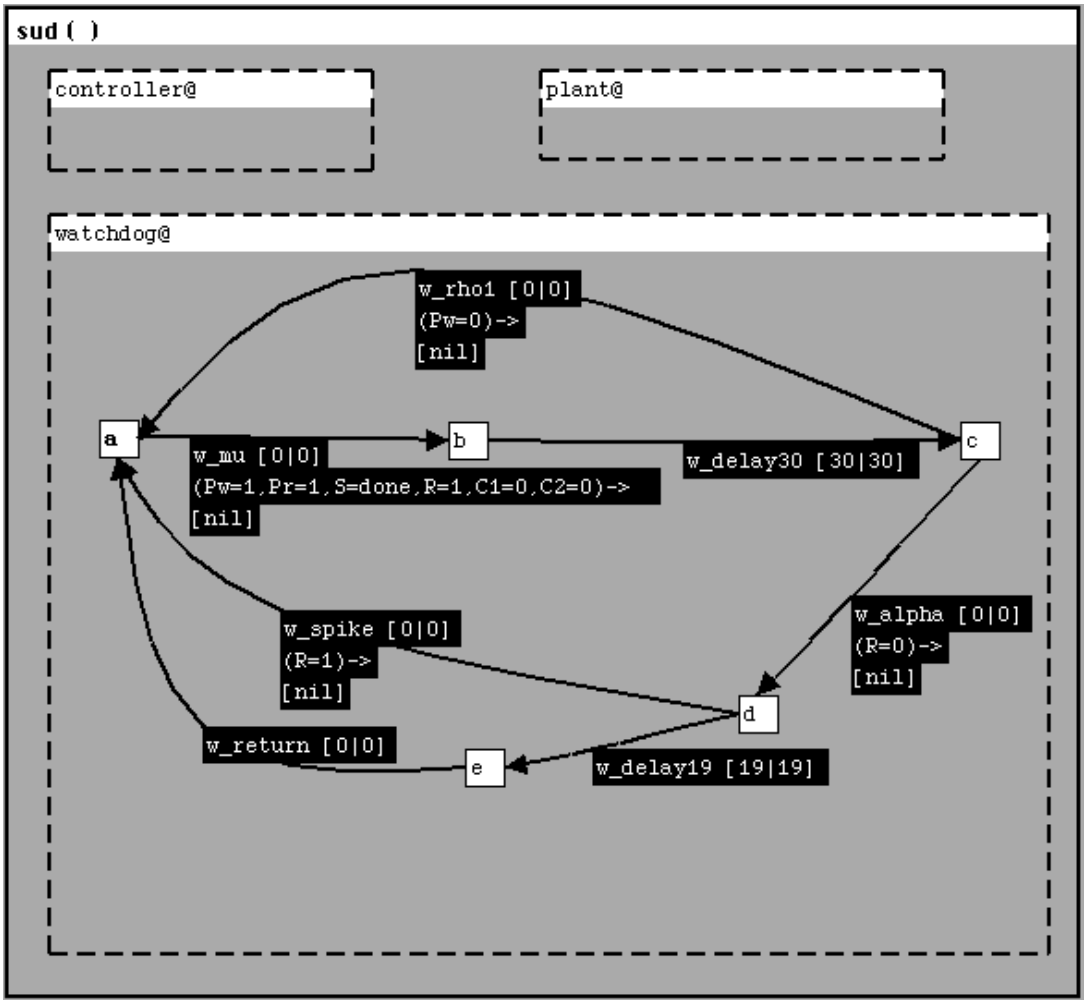
With each pass through the code, the microprocessor picks out one of the labelled blocks of code. The block chosen is the one whose enabling condition is satisfied. The program then loops back to the start and re-evaluates all the enabling conditions in the next cycle. The program structure is that of a large case statement repeatedly executed. Hence each event has a lower and upper time bound of one.

Conditions such as $Pressure \geq DSP$ (pressure exceeds delayed set point) and $Power \geq PT$ (power exceeds the power threshold) can be represented by $(Pr = 1)$ and $(Pw = 1)$ respectively (1 represents beyond the critical threshold and 0 represents normal levels).

In the guards of events a comma stands for conjunction and a semi-colon for disjunction. A transformation function such as [C2:C2+1, R:0] in the event mu2[1,1] of Figure 17 stands for simultaneous assignment (i.e. when the event is taken variable C2 is incremented by one and R is assigned zero).

The final TTMchart sud is obtained by AND-composing plant and controller as shown in Figure 18. The watchdog will be explained in the next subsection that deals with the RTTL specifications.

FIGURE 18. The complete system under design sud = controller || plant || watchdog



3.3 DRT requirements expressed in Temporal Logic

The informal specifications R1, R2 and R3 must now be translated into RTTL specifications. Consider requirement R2 which states that:

> [R2] If the power reaches an acceptable level then the relay should be closed as soon as possible.

At first glance, R2 may be written:

$$(Pw = 0) \Rightarrow \Diamond_{\leq 1} (Relay = closed)$$

i.e. if the power level is normal, then within one tick (100ms) the relay must be closed. The problem with this assertion is that it will not always be true in every computation of the DTR, because even if the power level is acceptable in a given state, the level may become critical in the next state and hence the relay should not be closed. Furthermore, the microprocessor may not be fast enough to detect such instantaneous changes, even presuming that the above assertion is the correct one to enforce. Rather, we must assert that whenever the power is normal for a sufficiently lengthy period of time, then the relay must be closed. R2 should therefore be written

$$\Box_{<2}(Pw = 0) \Rightarrow \Diamond_{\leq 1}(Relay = \text{closed})$$

meaning if any state is reached from which the power is low for at least two ticks of the clock, then eventually in all subsequent computations from that state, the relay must be closed within one tick. The above property can be rewritten more suitably for the verifier as:

$$\text{[R2]:} \quad (Pw = 0 \wedge \text{Init}) \Rightarrow \Diamond_{\leq 1}(Relay = \text{closed}) \qquad \text{(EQ 2)}$$

where $\text{Init} \stackrel{\text{def}}{=} (S = \text{done} \wedge C_1 = 0 = C_2)$. The formula $(S = \text{done})$ replaces the formula $\Box_{<2}(Pw = 0)$. The activity variable S of signal_update has the value done precisely when there are two ticks before the power is sampled. The conjunction $(C_1 = 0 = C_2)$ is required because the microprocessor does not close the relay while it is simulating the timers (Timer1 and Timer2). Hence, the relay is required to close only when the counters are zero.

The requirement R1 can be written as

$$\text{BothHi} \wedge \text{Reset} \Rightarrow \Diamond_{=30}[\text{PowerHi} \rightarrow \Diamond_{\leq 1}(\text{RelayOpen} \wedge \Box_{<20}\text{RelayOpen})] \qquad \text{(EQ 3)}$$

where the various formula names are defined as follows:

$\text{BothHi} \stackrel{\text{def}}{=} (Pw = 1 \wedge Pr = 1 \wedge S = \text{done})$,

$\text{Reset} \stackrel{\text{def}}{=} (R = 1 \wedge C_1 = 0 = C_2)$,

$\text{PowerHi} \stackrel{\text{def}}{=} (Pw = 1 \wedge S = \text{done})$,

and $\text{RelayOpen} \stackrel{\text{def}}{=} (Relay = \text{open})$.

Since (EQ 3) cannot be directly checked by the VERIFY tool, a watchdog (that observes but does not affect the plant or controller) must be constructed as shown in Figure 18. The activity variable of the watchdog is W. If $W = a$, then the watchdog detects when $(\text{BothHi} \wedge \text{Reset})$ holds as can be seen from the guard of w_mu. Then the watchdog delays for 30 ticks (3 seconds) at which point $(W = c)$ holds. At c, the event w_rho1[0,0] is immediately taken if the power is low, and w_alpha[0,0] is taken if the relay is open $(R = 0)$. If w_alpha is taken, then w_spike[0,0] checks that the relay is not opened for 19 ticks of the clock. After the 20th tick, the watchdog should be in activity e. Thus (EQ 3) can be checked by verifying that

$$\text{[R1]:} \quad [W = c \wedge Pw = 1 \wedge S = \text{done}] \Rightarrow \Diamond_{=20}(W = e) \qquad \text{(EQ 4)}$$

which is in a format suitable for the VERIFY tool.

By using the concept of a watchdog, most properties of interest can be checked in this way. However, there is a cost associated with adding the watchdog, as the size of the reachability graph is increased substantially.

The informal specification R3 requires that the system as a whole (SUD) not deadlock. The verifier is able to directly check that

$$\text{[R3a]:} \quad \Box(\text{enabled}) \tag{EQ 5}$$

which checks that in every reachable state there is at least one event other than tick that is enabled. (If only a clock tick is enabled in a state, then SUD deadlocks in that state as all that it can ever do is tick).

Furthermore, it is advisable to check that the watchdog taken by itself never deadlocks. More specifically it should be the case that

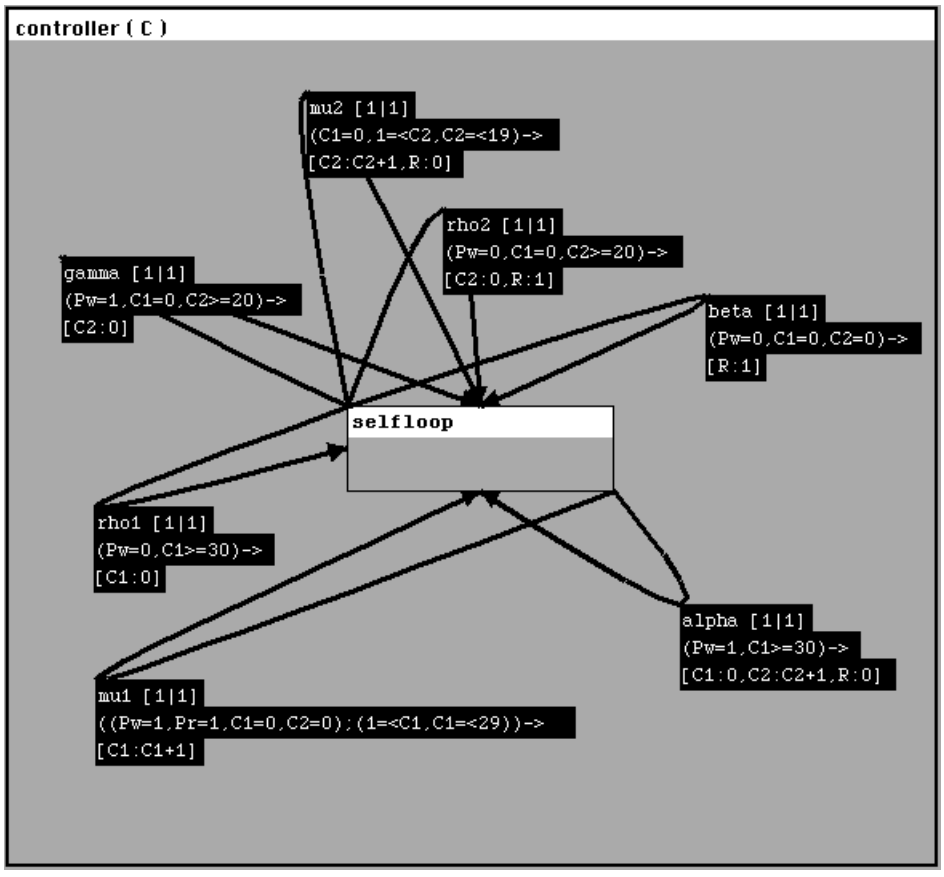
$$\text{[R3b]:} \quad (W \neq a) \Rightarrow \Diamond_{\leq 50}(W = a) \tag{EQ 6}$$

Finally, the relay should not be opened unnecessarily, i.e.

$$(Relay = \text{closed} \wedge Pw = 0) \Rightarrow (Relay = \text{closed}) \mathcal{W} (Pw = 1 \wedge Pr = 1) \tag{EQ 7}$$

The waiting-for property ensures that the relay is kept closed until the critical power and pressure condition is detected.

FIGURE 19. Successful controller



3.4 Using the VERIFY tool

The system (SUD) must now be checked for the requirements R1, R2, and R3. The VERIFY tool takes its input from the BUILD tool. The performance figures are provided in Table 2.

The pseudocode suggested for the microprocessor controller (Figure 17) is shown to be incorrect as the property R1 fails to be satisfied. VERIFY indicates where the failure takes place. Based on this debugging information, corrections can be made to the controller. A revised controller is suggested by Lawford in [14] which is obtained by a set of behaviour preserving transformations. The revised controller is shown in Figure 19. This revised controller satisfies all four properties R1,R2,R3a and R3b.

The incorrect controller has a reachability graph consisting of 89,730 states and edges. The revised controller is much smaller consisting of 17,841 states and edges.

TABLE 2. VERIFY tool performance statistics on a Sun 10/21

Check	Suggested Controller (Figure 17)	Revised Controller (Figure 19)
Reachability graph generation:	96.4 minutes Unique States: 4,448 Unique Histories: 1,103 Total States: 29,369 Total Edges: 60,361	19.2 minutes Unique States: 3,259 Unique Histories: 602 Total States: 5,902 Total Edges: 11,939
[R1] Open Relay	Property *fails* after 39.8 minutes	*Succeeds* after 2.4 minutes
[R2] Close relay		*Succeeds* after 2.7 minutes
[R3a] No Deadlock		*Succeeds* after 1.9 minutes
[R3b] Watchdog health		*Succeeds* after 3.22 minutes
Total Time	2.3 hours to determine that the pseudocode is *incorrect*.	29.4 minutes to check that the revised controller satisfies its specifications.

The current version of VERIFY dumps a compressed version of the reachability graph into a results file on disk. Each time a property is checked, the results file must be compiled in and decompressed by the Prolog verification code. This is very inefficient as the graph should be kept in the RAM memory. The time to consult and decompress the file for R1 is 22 minutes. Hence, with a slight change in the program the actual time to check R1 would be substantially less, viz. 39.8–22=17.2 minutes (rather than 39.8). There are many other inefficiencies in the current code.

4.0 Conclusions

By applying the formal methodology illustrated in this paper we are still by no means guaranteed that the revised controller is correct. However, validation of the design via execution and verification increase our confidence in the correctness of the design.

In addition, RTTL specifications are *incremental*. If after developing a set of requirements, we suddenly realize that the resulting specification is incomplete, the situation can be rectified by adding the missing property to the requirements as additional conjuncts. For example, Table 2 leaves out the requirement (EQ 7).

Once the omission of this requirement was detected, it was checked without regenerating the reachability graph or checking the other requirements again. As we come to understand the design, additional properties can be added to the requirements and checked with ease.

StateTime is a prototype toolset for exploring the interplay between formal methods, suitable visual representations of models and automated verification. It allows for the treatment of both spontaneous events (such as a device failures) as well as timed behaviour. A variety of computational notions such as concurrency, hierarchy, nondeterminism, and process interaction and communication can be represented. StateTime lets the user design parts of safety critical systems in a structured visual fashion, while providing automated formal analysis techniques for a variety of important properties including freedom from deadlock and real-time response.

In this paper an overview of the StateTime toolset has been provided as well as a set of *pragmatics*, i.e. an appropriate methodology for using the tool on safety critical systems. The main idea is to structure the system under design into two parts: the plant and the controller. The plant (together with its interface specification) represents the fixed environment in which context the computer controller must be designed. The plant and its requirements must therefore be represented before design of the controller commences. The closed loop system (plant + controller) must then be checked for correctness.The toolset may however be used in which ever way is convenient to the designer and is not confined to the pragmatics illustrated in this paper.

While graphical tools such as ObjecTime [28] and Observ [30] all have the ability to execute or simulate complex real-time systems, they do not incorporate formal verification methods. Statemate [7] incorporates some verification methods, but these are quite limited as explained in the introduction. None of theses tools provide the complete real-time modelling facilities of StateTime. Statemate and Observ provide a delay statement, but no facility to directly distinguish between spontaneous, just and timed events. ObjecTime cannot deal with “hard” time bounds, but is intended to model “soft” real-time systems.

As a future proposal, what is needed is a blend of the best techniques from the commercially available graphical systems together with appropriate formal verification methods. The following extensions to StateTime would therefore be needed.

The current version of StateTime is limited to integer and enumerated data types. Furthermore, there is no systematic way for re-using TTMs other than copying an already existing one. Both problems need to be addressed to improve the range of systems that the tool can address.

It is possible that object-oriented techniques can be used to solve both problems.

1. Any basic Smalltalk data type should be immediately be available to StateTime. New data types should be constructed as in Smalltalk. This will provide the designer with a rich set of modelling objects.

2. TTMs should also be objects. TTMs can then be re-used or further refined as a sub-class using inheritance. Changing the class will then automatically update all related sub-TTMs. This should provide for a powerful re-use facility. Real-

Time Systems in special domains such as communication protocols or flexible manufacturing systems can then develop special libraries of already verified TTM classes of use in that domain.

The above changes will require a complete re-write of the software. The VERIFY and DESIGN units will have to be re-written in Smalltalk to avoid the problem of translating Smalltalk classes into Prolog. One disadvantage of using Smalltalk is that we cannot achieve as efficient an execution as can be achieved by using C.

An example of the efficiency of using C can be seen in the Spin tool used for communication protocols [12]. Spin does not have any real-time features. To achieve efficiency, its data types are limited to booleans and integers. However, within these constraints it can analyze large state spaces (millions of states) in a fraction of the time of the Prolog based VERIFY tool.

Research will need to be done to ensure that TTM objects are theoretically sound. For example, when the designer creates a sub-class from an already existing super-class, how must the sub-class behave semantically to ensure consistency with the parent class. A class can inherit from another in two ways [1]. The descendant class can add a behaviour by including a new service, or by redefining a parent service. However, care must be taken that a descendant class not treacherously change the semantics of the service so that some trajectories of the parent class do not belong to the descendant class.

Compositional verification techniques must be incorporated into the toolset if larger systems are to be handled. The compositional techniques discussed at the end of [16] have been found useful in this regard, and plans are underway to incorporate these methods into StateTime. New results in temporal logic can also be used to speed up the verification [1]. Another promising technique, currently under further development, is the algebraic reduction of a larger TTM into a smaller reduced order model on the variables of interest [14].

The user interface of the BUILD tool must be improved. The execution facility should be animated and integrated with the charts. A palette of icons (e.g. for activities and edges) will make drawing easier. On the whole, the tool will not allow an illegal chart to be constructed. However, there are still some areas where more syntactic checking would be useful. The verifier diagnostic facility must be linked with the charts to make debugging a design easier.

StateTime is not yet an industrial strength tool. Nevertheless, in its current form it facilitates the design of parts of real systems (e.g. the delay trip reactor), and it has proved a useful basis for analyzing complex safety requirements.

5.0 Acknowledgments

John Dangov and Olaf van Breman wrote the VERIFY code as an undergraduate project. Tim Field implemented BUILD for his M.Sc thesis. Don Laws has debugged parts of both BUILD and VERIFY. He is also extending the BUILD program. Frank Crane of NASA has provided useful feedback on the VERIFY tool. Patrick Dymond (Chair, Department of Computer Science, York University) and Chris Phillips (Departmental Systems Coordinator) have both been extremely helpful in making facilities available for this work to be done.

6.0 REFERENCES

[1] R. Alur and T.A. Henzinger. *A Really Temporal Logic*. Journal of the ACM. Vol 41, No. 1, pp. 181-204, Jan 1994.

[2] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or How to Use Statecharts in Object Oriented Design. *IEEE Transactions on Software Engineering,* 18(1): 9–18, January 1992.

[3] A. Colmeraurer. An Introduction to PrologIII. *Communications of the ACM*, 33(7):69–90, July 1990.

[4] Crane, D.F. and P.J. Hamory. Reactive System Verification Case Study — Fault Tolerant Transputer Communication. *International Conference on Simulation and Hardware Description Language*. Tempe, Arizona, January1994.

[5] P.I. Davis. News from the Committee on Public Policy: Certification of Safety-Critical Software by Licenced Software Engineers. *Computer*, 72–73, December 1992.

[6] W.-P. de Roever. Foundations of Computer Science: Leaving the Ivory Tower. *EATCS Bulletin*, 44, June 1991.

[7] D. Harel, *et al*. Statemate: a Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16:403–414, 1990.

[8] A. Hall. Seven Myths of Formal Methods. *IEEE Software*, pp. 11–19, 1990.

[9] D.J. Hatley, and I.A. Pirbhai. *Strategies for Real-Time System Specification*, Dorset House Publishing Co, New York, 1988.

[10] C. Heitmeyer and B. Labaw. Requirements Specification of Hard Real-Time Systems: Experience with a Language and a Verifier. *Foundations of Real-Time Computing: Formal Specifications and Methods*, Kluwer Academic Publishers, 1991.

[11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

[12] G.J. Holzmann. *Design and Validation of Protocols.* Prentice Hall. 1990.

[13] F. Jahanian and D. Stuart. A Method for Verifying Properties of Modechart Specifications. Proceedings 9th Real-time Systems Symposium, IEEE Computer Society, p. 12-21, 1988.

[14] M. Lawford. Transformational Equivalence of Timed Transition Models. Systems Control Group, Department of Electrical Engineering, University of Toronto, TR# 9202, 1992.

[15] N.G. Leveson and C.S. Turner. *An Investigation of the Therac-25 Accidents*. Computer. July 1993.

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

[17] J.S. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press Limited (distributed by John Wiley and Sons), Advanced Software Development Series, Taunton, England, 1989.

[18] J.S. Ostroff. Deciding Properties of Timed Transition Models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, April 1990.

[19] J.S. Ostroff and W.M. Wonham. A Framework for Real-Time Discrete Event Control. *IEEE Transactions on Automatic Control*, April 1990.

[20] J.S. Ostroff. Constraint Logic Programming for Reasoning about Discrete Event Processes. *The Journal of Logic Programming*, II:3&4, October/November 1991.

[21] J.S. Ostroff. Systematic Development of Real-Time Discrete Event Systems. *Proceedings of the ECC91 European Control Conference*, 522–533, Hermes Press, Paris July 1991.

[22] J.S. Ostroff. StateTime — a Diagrammatic Toolset for the Design and Verification of Real-Time Systems. Technical Report No. TR CS-92-07, Department of Computer Science, York University, Canada. 1992.

[23] J.S. Ostroff. Verification of safety critical systems using TTM/RTTL. *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, LNCS 600, Springer-Verlag, 1992.

[24] J.S. Ostroff. Design of Real-Time Safety Critical Systems. *The Journal of Systems and Software*, 18(1): 33–60, April 1992.

[25] J.S. Ostroff. A Verifier for Real-Time Properties.*The Journal of Real-Time Systems*, Volume 4, pages 5–35, 1992.

[26] J. S. Ostroff. Visual Tools for Verifying Real-Time Systems. *Theories and Experiences in Real-Time Systems: First AMAST Workshop in Real-Time Systems*. University of Iowa, Iowa City, 1993.

[27] D.L. Parnas, A.J.v. Schouwen, and S.P. Kwan. Evaluation Standards for Safety-Critical Software. TR 88-220, Department of Computer Science, Queen's University, May 1988.

[28] B. Selic, *et al.* ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. *CASE'92 Fifth International Workshop on Computer-Aided Software Engineering*, IEEE Computer Society Press, 1992.

[29] Tools Fair: Technical Code Generation. *IEEE Software*, p. 75, May 1992.

[30] S. Tyszberowicz and A. Yehudai. OBSERV — A Prototyping Language and Environment. *ACM Transactions on Software Engineering Methodology*, 1(3):269-309, July 1992.

[31] W.M.P. v.d. Aalst. Timed Coloured Petri Nets and their Application to Logistics. Ph.D Thesis, Eindhoven University of Technology, 1992.

[32] P. Ward, and S. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, New York, 1985.

[33] D.P. Wood, and W.G. Wood. Comparative Evaluations of Four Specification Methods for Real-Time Systems. CMU/SEI-89-TR-36, Software Engineering Institute, Carnegie Mellon University, 1989.