April 1992

Proving Properties of Real-Time Distributed Systems: A Comparison of Three Approaches

Patrice Brémond-Grégoire
University of Pennsylvania

Susan B. Davidson
University of Pennsylvania, susan@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-92-20.

Proving Properties of Real-Time Distributed Systems: A Comparison of Three Approaches

Abstract

Three formal methods for specifying properties of real-time systems are reviewed and used in a common example. Two of them offer a graphical representation and the third is an algebraic language. The example is that of an automatic railroad system with sensors to detect the train position and controls for the gate mechanism. Associated with each formalism is a proof methodology which is described and used to prove a safety property about the example. A comparison is made between the three formalisms according to various criteria including the expressiveness, readability, maintainability of the language, support for real-time concepts, method for expressing properties and proof mechanisms.

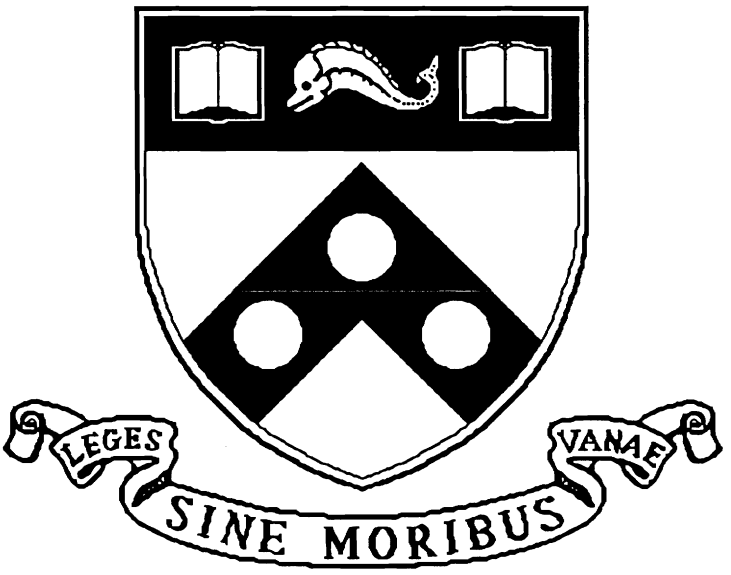
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-92-20.

Proving Properties of Real-Time Distributed Systems: A Comparison of Three Approaches

MS-CIS-92-20
GRASP LAB 306

Patrice Brémond-Grégoire
Susan Davidson
Insup Lee



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

March 1992

Proving Properties of Real-Time Distributed Systems: A Comparison of Three Approaches

Patrice Brémond-Grégoire, Susan Davidson, Insup Lee
Unversity of Pennsylvania

April 1992

Abstract

Three formal methods for specifying properties of real-time systems are reviewed and used in a common example. Two of them offer a graphical representation and the third is an algebraic language. The example is that of an automatic railroad crossing system with sensors to detect the train position and controls for the gate mechanism. Associated with each formalism is a proof methodology which is described and used to prove a safety property about the example. A comparison is made between the three formalisms according to various criteria including the expressiveness, readability, maintainability of the language, support for real-time concepts, method for expressing properties and proof mechanisms.

1 Introduction

As computers are being used more and more often to control critical systems, the need for formal methods to specify the system behavior and formally prove the correctness of the specification becomes more and more acute. Incorrect programs may have far reaching consequences including the loss of property and even human life. This is particularly true in the case of distributed real time systems, since they are used to control such things as transportation systems and weapons. Real-time programs must not only perform the correct function, they must do so in the prescribed time frame. This adds a new dimension to the complexity of the problem.

A solution is to have a tool to allow us to specify systems and a methodology to prove that, once implemented, they will exhibit all required properties, including the temporal ones. This goal is still far away, but progress worth noting has been made in the past few years.

In this paper, we will survey three different formalisms to specify real-time distributed systems. For each one, we briefly describe the syntax and semantics of the language and use it to specify a common example. We then present the mechanisms available to prove properties of the system and use those mechanisms to verify a safety assertion.

The system used as an example, a railroad crossing, must satisfy the following requirements. There are two sensors to detect the position of the train. Each sensor has a reaction time of at most 50 time units. The first sensor is positioned in such a way that the train will take at least 300 units of time to travel from the sensor to the crossing. The second sensor is positioned to be reached as soon as the train exits the crossing. The gates take a maximum of 50 units of time from when they start moving down until it they are fully closed. The time taken by the gates to move up to a full open position is not specified. In addition, there is a mechanism (outside of the scope of the specified system) that prevents a new train from approaching the crossing until the previous one has exited for at least 100 units of time. Finally, a correct system must satisfy a safety property that the train cannot be in the crossing if the gates are not down.

Before diving into the details of the three methods, we define the criteria that will be used to compare them. We look at the expressive power of the language, as well as its readability, maintainability and reusability of specifications. We compare their notion of time and their support for synchronization and communication. We examine what type of property can be formally expressed in each system. We look at such aspects of the proof system as applicability, completeness, complexity and implementability.

The first system we study, Hierarchical Multi-State machines is an extension of finite state automata. The distributed aspect of a system is captured by multiple states being active simultaneously, as opposed to finite state machines which have only one state active. The behavior and real-time aspects are expressed by a sophisticated set of controls on the state transitions. The proof system consists of a set of rules that can be used to manipulate and rewrite these transitions. In this methodology, safety properties are represented by failure states. The proof strategy consists of attempting to build the set of all possible transitions that leading from an initial state directly to a failure state. The absence of such a transition proves the unreachability of the failure state.

The second system, modechart, is a graphical way of representing systems as a structured set of modes, a concept very similar to states. Distribution is specified by parallel modes where several submodes are active simultaneously. Functionality is specified by transitions between modes, and by actions. Actions are a means to formally state the modification of state variables and to informally indicate the effects of the system on its environment. Real-time aspects of a system are stipulated by associating delays and deadline to the various transitions and actions. Tied to modechart is Real-Time Logic, a model theoretic approach to specify properties of a real-time system. The proof method-

ology for modechart consists of developing a computation graph, which is similar to a computation tree except that folding occurs. The vertices of the graph represent all the possible combinations of modes the system can reach in any execution. The edges are labelled with timing information that capture, in a very special way, the minimum and maximum timing constraint between points. Safety properties are proven if the graph does not exhibit any path to an unsafe configuration.

In the third formalism, Calculus for Communicating Shared Resources or CCSR, systems are specified as algebraic expressions. The functionality is given by the specification of sequence of actions which are in fact sets of simultaneous events. The distributed aspect is addressed by the stipulation of parallel behaviors via an operator. Each event in the system is presumed to take a constant, minimum observable, amount of time, this takes care of the real time aspect. CCSR is a much more powerful language than the other two in that it allows us to model not only true parallelism, but also interleaving with priority based preemption. The proof methodology of CCSR consists of algebraic manipulations of the expression according to a set of axioms that preserve equivalence. A system is said to be correct if it can be shown to be equivalent to another, simpler, system which is known to be correct.

For now, let us see on what grounds we will compare these three systems.

2 Comparison criteria

In this section, we define the various criteria that we will be using later to compare the three methods. We separate them in three groups: general criteria, real time properties and the proof mechanism.

2.1 General criteria

This classification includes those properties of a formal language that are not particularly specific of real-time systems.

Expressiveness: The capability of a language to precisely express the ideas and concepts as desired by the author. It should be possible, without too much difficulty, to describe exactly and unambiguously a solution to our small example. Long winded discussion should not be required to enounce simple concepts. Most importantly all of the aspects of the system should be specified using the apparatus of the formal method. Side notes and informal English sentences should not be required except to help the human reader understand the motivations of the author.

Readability: It is important that during and until long after the design of a system, one can look at the expression of the design and easily understand it. The notations

should be clear, concise and intuitive.

Maintainability: The only systems that do not evolve are those that are not used. The ease of maintenance is one of the most important characteristic of any development system. It is important to be able to design a system where the interactions between the various parts are very localized and/or identifiable. The total construction should not have the intricacy of a card castle that will collapse at the first attempt to move one of its pieces.

Modularity: The ability to define independent modules that interact in formally identified ways. Tools that facilitate modularity are better suited for large system development because independent modules can be assigned to different persons or teams.

Abstraction: the capability to abstract out the details of the implementation of some subsystems and look at them as black boxes with specified input and output is important for multiple reasons. They allow the specification of systems by successive refinements, while keeping the same formal expression at all levels. The resulting systems are usually easier to understand because of the limited amount of information one has to keep in mind while looking at one level of the system. Finally and maybe most importantly, abstraction is an effective way of achieving code reusability. Proven subsystems can be identified and included in many independent designs.

Property assertions: An important aspect of proving correctness of a particular system is the definition of what correct means. In this paper, we will adhere to the idea that a correct system is one that satisfy a set of properties that we call requirements. In order to be provable, properties must be formally (i.e. precisely and unambiguously) expressed. In keeping with Alpern and Schneider [AS85], we define a safety property as a property such that failure to satisfy the property can always be detected in a finite amount of time. Conversely, failure to satisfy a liveness property cannot be detected by looking at a partial execution.

2.2 Real-time concepts

In this section, we will specialize and look more particularly at the support for concepts that are very common in real-time systems. Some of theses concepts are not unique to real-time systems, but they are nevertheless very useful when studying them.

System state: A configuration of the system totally determines all of its possible future behaviors. It is often important to identify such points as milestones when analyzing the behaviors of a system. When a given state has been thoroughly analyzed, it does not need to be re-examined when encountered again later.

Notion of time: Time is obviously a key ingredient of real-time systems but there are numerous ways to represent and manipulate it. It can be continuous or discrete. When discrete, the granularity is important. It can be global, the whole system evolves

to one synchronous clock, or distributed, different parts can evolve independently and synchronize only when they need to communicate. Another important characteristic of real-time formalisms is the kind of time expression one can define. Can one refer to the exact time of occurrence of any event arbitrarily far in the past? What kind of time expression are supported?

Delays and deadlines: These are ubiquitous concepts in real-time systems. The ease with which one can specify maximums and minimums for time intervals and recovery mechanisms is an interesting point of comparison between real-time development systems.

Synchronization: The ability to specify that two events happen at exactly the same time. This too is an important aspect of real-time systems and must be supported by any real-time formal development method.

2.3 The proof system

This discussion is devided into two parts. Before attempting to formally prove anything, one must have a formal enunciation of the object of the proof. Therefore we first look at how properties can be stated, and then at how they can be proven.

Specification of properties: Here we are interested in the semantics domain of the property definition language (what can be expressed) as well as the syntax. The property specification language can be the same as (or a subset of) the system definition language or it can be a completely different language altogether. When the language is different, the questions of readability, maintainability etc. should be posed all over again. The specification of safety and liveness properties are very different in nature and it is interesting to note whether the language supports them both. Finally, functional versus operational style is another differenciating factor.

The proof mechanism: The most important quality of a proof system is soundness; every provable fact must be true. Without soundness, there is no proof system. Another important aspect is completeness, is every true fact provable. Convergence, the assurance that steady progress can be made towards proving a true fact is an important point in automating proof systems. Finally, the practicality of developing correct proofs manually and developing automatic or semi automatic proof tools is a key to the usefulness of any formal method.

3 Hierarchical Multi-State Machines (HMS)

HMS machines were introduced by A. Gabrielian and M.K. Franklin in 1988 as a "formal theory of state-based executable specification of complex real-time systems" [GF88]; in

this paper, they formally describe several extensions to finite-state automata:

- multiple simultaneously active states,
- multiple concurrent transitions, both deterministic and non-deterministic,
- transition control (enable or disable) based on state and temporal considerations,
- capability for states to hold and pass information in the form of tokens,
- hierarchies.

In a subsequent paper [FG89] the same authors take a subset of the above full HMS for which they develop a proof system. Among the significant features that were excluded from this second paper were the hierarchies and the capabilities for states to hold and pass information in the form of tokens.

Since [FG89] contains a complete formal description of HMS machines, they are presented here in a more informal manner, moving freely between syntax and semantics to get more directly to the intuition behind the concepts.

3.1 Description of HMS

An HMS machine is composed of a set of states, a set of deterministic transitions and a set of non-deterministic transitions. The machine evolves in discrete time. At any given instant in time, a subset of the states are said to be marked (or true) and a subset of the transitions are said to be enabled. All the deterministic transitions and an arbitrary (possibly empty) subset of the non-deterministic transitions that are enabled at an instant will actually fire (and be marked) at the next instant, thus usually (but not necessarily) changing the set of marked states. Note one major difference with finite state automata: because of the possibly to control transitions based on the transitions that fired at a given point in time, it is possible that the set of active states do not change between two successive instants without the machine being stuck.

The time is measured as follows: 0 is always the present. Past instants are denoted by negative integers, relative to the present: smaller and smaller number denote more and more distant past. The control of the machine can be made dependent of its history, i.e. the states that were active and/or the transitions that fired in arbitrarily distant, but always relative, past can be made to influence the set of enabled transitions. Time expressions are always constant and hence the *relevant* history of any given machine is bounded.

A transition, traditionally called γ, is of the form

$$\mathrm{id} : \mathrm{PRIMS}(\gamma)\mathrm{CNTRLS}(\gamma) \rightarrow \mathrm{CNSQS}(\gamma).$$

Where id is the label of the transition. PRIMS(γ) is a set of states called primaries, CNTRLS(γ) is a set of controls. For a transition to be enabled, all the states in PRIMS(γ) must be marked, and all the controls must be satisfied (see below). When a transition fires, all the states in CNSQS(γ), i.e. the consequences, become marked and all the states of PRIMS(γ) that are not in any of the CNSQS set of any transition that fire simultaneously become un-marked.

A control, c is an expression of the form (x, t) or ($\neg$x, t) where x can be either a state or the label of a transition and t is a time expression $\langle t_1, t_2 \rangle$, $[t_1, t_2]$ or $\langle t_1, t_2 \rangle!$ where $t_1 \leq t_2 \leq 0$. When $t_1 = t_2$, t is used as an abbreviation for both $\langle t, t \rangle$ and $[t, t]$, and t! is used for $\langle t, t \rangle!$.

A control c is said to be satisfied if and only if one of the following rules applies. (By convention, x can be either a state or the label of a transition and the bounds t_1 and t_2 are always included.)

- c is $(x, \langle t_1, t_2 \rangle)$ and x was marked sometime between t_1 and t_2.
- c is $(x, [t_1, t_2])$ and x was marked at all times between t_1 and t_2.
- c is $(x, \langle t_1, t_2 \rangle!)$, x was not marked before t_1 and became marked sometime between t_1 and t_2.
- c is $(\neg x, \langle t_1, t_2 \rangle)$ and x there was a time between t_1 and t_2 were x was not marked.
- c is $(\neg x, [t_1, t_2])$ and x was not marked at any time between t_1 and t_2.
- c is $(\neg x, \langle t_1, t_2 \rangle!)$, x was marked before t_1 and became not marked sometime between t_1 and t_2.

Interestingly enough, the initial state (or rather set of states) of an HMS machine is not part of the specification of the machine. Rather one studies the behavior of an HMS machine as a function of a set of states and transitions that are marked at the beginning of the (recorded) history.

Similar to S-invariant in Petri nets [Rei85], 1-invariant sets of states in an HMS machine are sets of states such that if one and only one state of the set is marked at time 0, one and only one will be marked at any time during the execution of the HMS machine. 1-Invariant sets play a significant role in the proof of properties of HMS machines. More work needs to be done to define methods to discover and prove them. One obvious case is when all the transitions in and out of a set of state form a single cycle.

A safety property is specified as $\Box \rightarrow l_1, l_2, \cdots l_n$ where each l_i is either a state or the negation of a state. The safety property is satisfied if, at the present time and at every

instant in the future, there is an i such that either l_i is a state and that state is marked or l_i is the negation of a state and that state is not marked.

Safety properties can be expressed in HMS machines by adding new states to represent the undesired situations, and adding transitions leading to the unsafe states. These transitions can be easily derived from the formal safety property: they have an empty set of primaries and one control of the form $(\neg l_i, 0)$ for each l_i. Safety verifications are thus transformed into reachability problems.

A deadline is formally expressed as $\Box(l \rightarrow (l_1 \vee l_2 \vee \cdots \vee l_n)$ before d) which means that within d units of time after l is satisfied (marked if l is a state or unmarked if l is the negation of a state) one of the l_i will be satisfied. Again this can be easily expressed in an HMS machine by a "missed deadline" state. A transition to that state that has no primary, a control of the form (l, -d) plus one control of the form $(\neg l_i, [-d, 0])$ for each l_i.

Let us now look at the specification of our railway crossing using HMS.

3.2 Railroad Crossing using HMS

The railroad crossing system is shown in figure 1. The conventions for this graphical representation are as follows. Boxes represent states. Thick arrows link the primaries to the consequences of transitions. Transitions without primaries originate from a cross-bar. Non-deterministic transitions are characterized by a star near the arrow head. Thin lines represent controls, they go from a state, or a transition, to a small oval on the transition they control. Timing expressions are written next to a circled T located on the control they apply to. We call the unshaded states functional states because they are part of the specification of the functionality, while shaded states represent missed deadlines and/or unsafe conditions.

Figure 1 expresses the following. When a train is nowhere to be seen, the Before Crossing (BC) state is marked. Non-deterministically, a train will show up, that is the sate Near Crossing (NC) will be marked and the state BC will be un-marked; this is expressed by the transition t_1. Any time after 300 units, the train will be In Cross (IC) and this is expressed by the transition t_2. After an unspecified amount of time, the train will have past the crossing as expressed by the transition t_3 and the state Past Cross (PC). Finally, not earlier than 100 units after a train has past the crossing will another train be able to approach the crossing, this is expressed by the transition t_4.

One of the conditions for the correct evolution of the system is that one and only one of the four states (BC, NC, IC, PC) be marked in the original markings of the HMS machine, otherwise, trains could vanish non-deterministically. Unfortunately, there is no way to express this constraint in the HMS formalism. Nevertheless it is easy to see that this set of states does form a 1-invariant of the system.

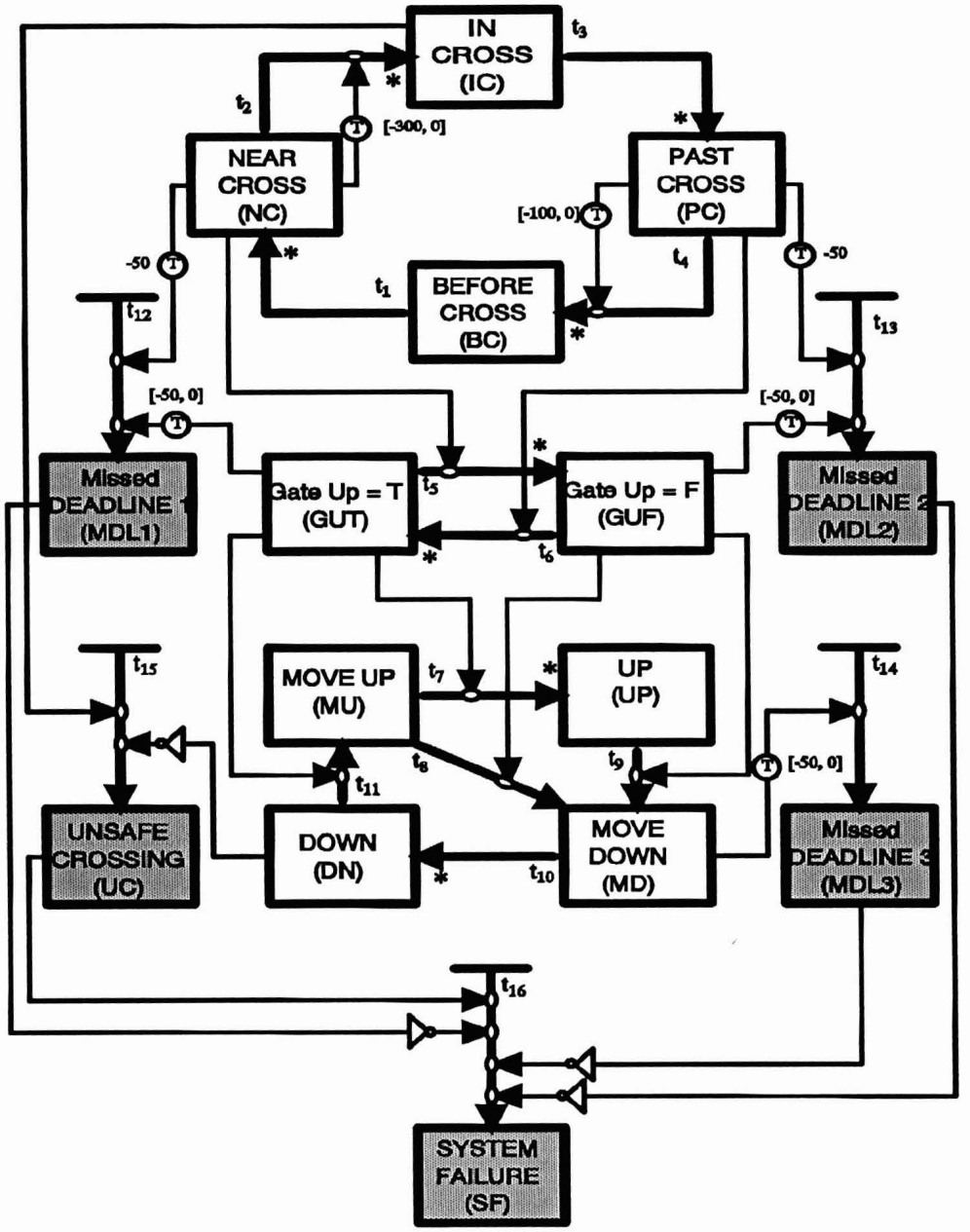


Figure 1: The railroad crossing using Hierarchical Multi State machines

The pair of states Gate Up = T (GUT) and Gate Up = F (GUF) form a control of the system, remembering whether the gate can be up (GUT) or should be down (GUF). Transition t_5 indicates that , at any time while the train is near crossing (NC is marked), the control may switch from GUT to GUF. Transition t_6 indicates that, at any time while the train is past crossing (PC is marked), the control may switch back from GUF to GUT. GUT and GUF form a second 1-invariant of the system.

Finally, the last four functional states describe the operation of the gate. Down is marked when the gate is down. When GUT is marked, the gate will unconditionally start moving up (transition t_{11}). If GUF becomes marked while the gate is moving up, it will immediately reverse direction and start moving down (MD) as dictated by transition t_8. After and undetermined amount of time, and if the state GUT is still marked, the gate will be in the up (UP) position (transition t_7). From that state, if GUF becomes marked, the gate will unconditionally start moving down (MD) per transition t_9. Finally, transition t_{10} specifies that, after an undetermined amount of time moving down, the gate will be in the down position.

At this point it is interesting to note that the four states UP, MD, DN and MU do form a 1-invariant of the system. This is true because the transitions t_7 and t_8 have conflicting controls, and cannot be simultaneously enabled. This, in turn, stems from the fact that GUT and GUF form a 1-invariant of the system. This small example illustrates the difficulty of discovering and proving 1-invariant.

In order for this system to work, some deadlines must be met. GUT must be marked within 50 units of time of NC being marked. GUF must be marked within 50 units of time of PC being marked. MD must not be marked for more than 50 time units, i.e. the gate must not take more than 50 units to come down.

The unsafe crossing state (UC) is reached if a train is In Crossing (IC) and the gate is not down ($\neg$DN). The safety property of the system states that there cannot be any unsafe crossing unless a deadline has been missed.

3.3 The Proof System for HMS

Before talking about the proof system, we need to specify the notion of equivalence for HMS machines. Two HMS machines are said to be equivalent if and only if they have the same set of states and the set of all possible executions (i.e. sequences of sets of marked states) is the same for both machines. In other words, the exact transitions that fire at each step are unimportant as long as their short term (e.g. next state marking) and long term (e.g. transition enabling) effects are the same.

The proof system for HMS consists of a large number of rewrite rules on the transitions. For the sake of space, we will only present those rules that will be used in the proof of the example, but they are fairly typical and give a good idea of the system.

Delay change: A delay can be extended to the beginning of times if the control in which it appears specifies a state that cannot be reached (i.e. it is in no transition consequents) or specifies a negated state that cannot be left (i.e. it is in no transition primaries). Extending the delay may exhibit conflicting controls.

Case split: consists of replacing a transition that has a control requiring the entrance of a state s during a given interval —e.g. $c = (s, \langle t_1, t_2 \rangle !)$— by a set of transitions, corresponding to each transition that leads to s, with additional controls stating that the corresponding transition (leading to s) has fired during the interval and that the state s was not marked before entering the interval. Intuitively, for the state s to be entered during the interval, it must not be marked before the interval and a transition leading to it must fire during the interval. As the name indicate, case split is useful to decompose a transition into several scenarios.

Control Addition: if a transition g has a control that requires that another transition γ' fire during an interval t, then all the controls and all the primaries of γ' must have been true during an interval that is derived from t by sliding back in time by one time unit, these new controls can be added to γ. This is a key rule because it allows to move back in time and explore the possible histories leading to a particular transition being enabled.

Transition deletion: transitions can be deleted if they cannot be enabled. There are several cases of this: if they have conflicting controls, if they have a control that requires leaving a state s and there is no transition out of s, or if they would violate a 1-invariant of the system.

There is a set of 9 rules along the lines of the ones shown above that can be proven to be complete in the sense that if a transition cannot enabled in any legitimate execution of an HMS machine, it can be eliminated by applying the rewrite rules. The proof is constructive and is based on the fact that the relevant history is limited since all the time expressions are constants, and that all possible scenarios of transitions firing in that interval can be embodied into a finite set of transitions. Each transition that cannot be enabled will exhibit conflicting controls and hence can be deleted.

The rules that we have seen so far do preserve correctness i.e. an HMS machine obtained by applying those rules is truly equivalent to the original HMS machine. This is useful when attempting to simplify a design and there is a need to prove that the resulting machine is equivalent to the original one. However, since safety problems are transformed into reachability problems, reachability preserving transformations are just as useful as equivalence preserving transformations. Delay sharpening is such a transformation.

The idea behind Delay Sharpening it is that if a state is reachable then there is a transition that is enabled and leads to it. This transition (say γ) may have a number

of controls and a number of primaries; for g to become enabled, at least one of them has to be the last to become true. We can investigate all the possibilities of one of the controls and/or primaries becoming true at the instant prior to the transition firing. This is done by splitting the transition into a set of transitions, one corresponding to each possibility of one control or one primary changing to become true just before the state under scrutiny being reached. This method does not preserve equivalence since an enabled transition does not necessarily fire, but it does preserve reachability since a transition must fire for a state to be reached.

Let us examine how this work on the example.

3.4 Proving correctness of the railway crossing

Proving correctness of the railway crossing means proving the unreachability of the state SF (System Failure.) The only transition that leads to the SF state is t_{16}:

$$\emptyset\{(\neg\text{MDL1}, 0)(\neg\text{MDL2}, 0)(\neg\text{MDL3}, 0)(\text{UC}, 0)\} \rightarrow \{\text{SF}\}$$

Using the delay sharpening rule we obtain four transitions:

$$\emptyset\{(\neg\text{MDL1}, 0!)(\neg\text{MDL2}, 0)(\neg\text{MDL3}, 0)(\text{UC}, 0)\} \rightarrow \{\text{SF}\}$$

$$\emptyset\{(\neg\text{MDL1}, 0)(\neg\text{MDL2}, 0!)(\neg\text{MDL3}, 0)(\text{UC}, 0)\} \rightarrow \{\text{SF}\}$$

$$\emptyset\{(\neg\text{MDL1}, 0)(\neg\text{MDL2}, 0)(\neg\text{MDL3}, 0!)(\text{UC}, 0)\} \rightarrow \{\text{SF}\}$$

$$\emptyset\{(\neg\text{MDL1}, 0)(\neg\text{MDL2}, 0)(\neg\text{MDL3}, 0)(\text{UC}, 0!)\} \rightarrow \{\text{SF}\}$$

The first three transitions can be eliminated by applying the transition deletion rule since there is no transition leading out of a missed deadline state; for the same reason, we can apply three times delay change to the fourth transition to obtain:

$$\emptyset\{(\neg\text{MDL1}, [-\infty, 0])(\neg\text{MDL2}, [-\infty, 0])(\neg\text{MDL3}, [-\infty, 0])(\text{UC}, 0!)\} \rightarrow \{\text{SF}\}$$

By applying case split, and since t_{15} is the only transition that leads to UC, we obtain:

$$\emptyset\{(\neg\text{MDL1}, [-\infty, 0])(\neg\text{MDL2}, [-\infty, 0])(\neg\text{MDL3}, [-\infty, 0])(\neg\text{UC}, -1)(t_{15}, 0)\} \rightarrow \{\text{SF}\}$$

Since there is no transition leading out of UC, we can apply delay change and replace $(\neg\text{UC}, -1)$ by $(\neg\text{UC}, [-\infty, -1])$. Then using control addition with the controls that makes t_{15} fire, we obtain:

$$\emptyset\{(\neg\text{MDL1}, [-\infty, 0])(\neg\text{MDL2}, [-\infty, 0])(\neg\text{MDL3}, [-\infty, 0])$$
$$(\neg\text{UC}, [-\infty, -1])(t_{15}, 0)(\neg\text{DN}, -1)(\text{IC}, -1)\} \rightarrow \{\text{SF}\}$$

Delay sharpening will now give us two transitions:

$$\emptyset\{(\neg\text{MDL1},[-\infty,0])(\neg\text{MDL2},[-\infty,0])(\neg\text{MDL3},[-\infty,0])(\neg\text{UC},[-\infty,-1])(t_{15},0)(\neg\text{DN},-1!)(\text{IC},-1)\} \rightarrow \{\text{SF}\}$$

and

$$\emptyset\{(\neg\text{MDL1},[-\infty,0])(\neg\text{MDL2},[-\infty,0])(\neg\text{MDL3},[-\infty,0])(\neg\text{UC},[-\infty,-1])(t_{15},0)(\neg\text{DN},-1)(\text{IC},-1!)\} \rightarrow \{\text{SF}\}$$

At this point, the proof tree branches out. For the sake of space, we are going to explore the simplest path and that will give us a good idea of how the proof goes. Case split applied to $(\neg\text{DN},-1!)$ will add the controls $(\text{DN},-2)$ and $(t_{11},-1)$ i.e. in order to move out of DN at time -1, DN must have been marked at -2 and t_{11} must have fired at -1. The controls for t2 imply (GUT, -2) which we add. At this point, the transition looks like this:

$$\emptyset\{(\neg\text{MDL1},[-\infty,0])(\neg\text{MDL2},[-\infty,0])(\neg\text{MDL3},[-\infty,0])(\neg\text{UC},[-\infty,-1])(t_{15},0)(\neg\text{DN},-1!)(\text{IC},-1)(t_{11},-1)(\text{DN},-2)(\text{GUT},-2)\} \rightarrow \{\text{SF}\}$$

Again we apply delay sharpening to the pair (DN, -2) (GUT, -2) and follow one of the branches:

$$\emptyset\{(\neg\text{MDL1},[-\infty,0])(\neg\text{MDL2},[-\infty,0])(\neg\text{MDL3},[-\infty,0])(\neg\text{UC},[-\infty,-1])(t_{15},0)(\neg\text{DN},-1!)(\text{IC},-1)(t_{11},-1)(\text{DN},-2)(\text{GUT},-2!)\} \rightarrow \{\text{SF}\}$$

We now apply case split, and since t_6 is the only transition that leads to GUT, we can add the controls $(t_6,-2)$ $(\neg\text{GUT},-3)$, then we add the controls that makes t_6 fire: (PC, -3). At this point, it is easy to see that, given (IC, -1), either IC or NC has to be marked at -3, which violates the 1-invariant of IC, PC, BC, NC.

The other branches eventually resolve in more or less the same way. The total proof takes 104 steps, the proof tree has 8 main leaves and depth ranging from 19 to 35.

There are a number of remarks that can be made about this proof. One is that it is fairly complex for a system that is rather simple. In particular, in all the applications of case split we had to apply, only one transition was involved with the consequence that the proof tree was not branching at this point; should there have been more transitions, the tree would have grown exponentially. This is not surprising since state reachability is an NP-complete problem.

There are a great deal of rules in the proof system, and choosing the right one is not necessarily an easy task.

The proof is heavily based on proving the violation of 1-invariant. In this specific case, proving 1-invariant was trivial given the simple cyclic structure of the transitions in and out of the states. In general, however, proving 1-invariant may be a difficult task in itself.

4 Modechart

Modechart [JLM88] is a graphical language for the formal specification of the behavior of real-time systems. Modechart is derived from Harel's statecharts [Har87, Har88]. As the names indicate, statecharts deals with states and modechart deals with modes. Although somewhat similar in their syntax, (statecharts uses rounded angles while modechart uses square angles) the two languages are very different in their semantics. Statecharts, on the one hand, was designed as an extension to finite state automata and state transitions are taken according to external inputs; the semantics of a statecharts is the set of acceptable sequences of inputs. Modechart, on the other hand, is intended for the formal description of real-time systems, and the mode transitions are the consequences of event occurrences and timing constraints; the semantics of a modechart is a set of set of events with their time of occurrence .

In modechart, a system is defined as a hierarchy of modes. Each mode is represented as a box and is given a name. The most basic modes are called primary modes and have no internal structure. In figure 2, modes M1, M4, M5, M7 M8, M10 and M11 are primary modes. There are two kinds of structured modes (modes that contain other modes): serial and parallel. The hierarchy of modes is presented by the geometrical inclusion of boxes within boxes.

Parallel modes are identified by the word *parallel* in the upper left corner. When a system enters a parallel mode, all the children are entered simultaneously. When the system exits a parallel mode, all the children are exited as well. There are no transitions between children of a parallel mode.

Serial modes are identified by the word *serial* in the upper left corner. When a system is in a serial mode, it is in exactly one of the children. As the system exits a child of a serial mode, it must simultaneously enter another child or exit the serial mode altogether. Transitions between children of a serial mode are allowed, but not required because it is possible for different transitions to land on different modes of the same serial mode, as, for example mode M3 of figure 2. A serial mode may have a distinguished (initial) mode which is the mode which is entered by default when the serial mode is entered via a transition that does not land on one of its descendents (e.g. a transition that lands on the child of siblings in a parallel mode). Initial modes are identified by an arrow, for example in figure 2, M1 and M7 are initial modes.

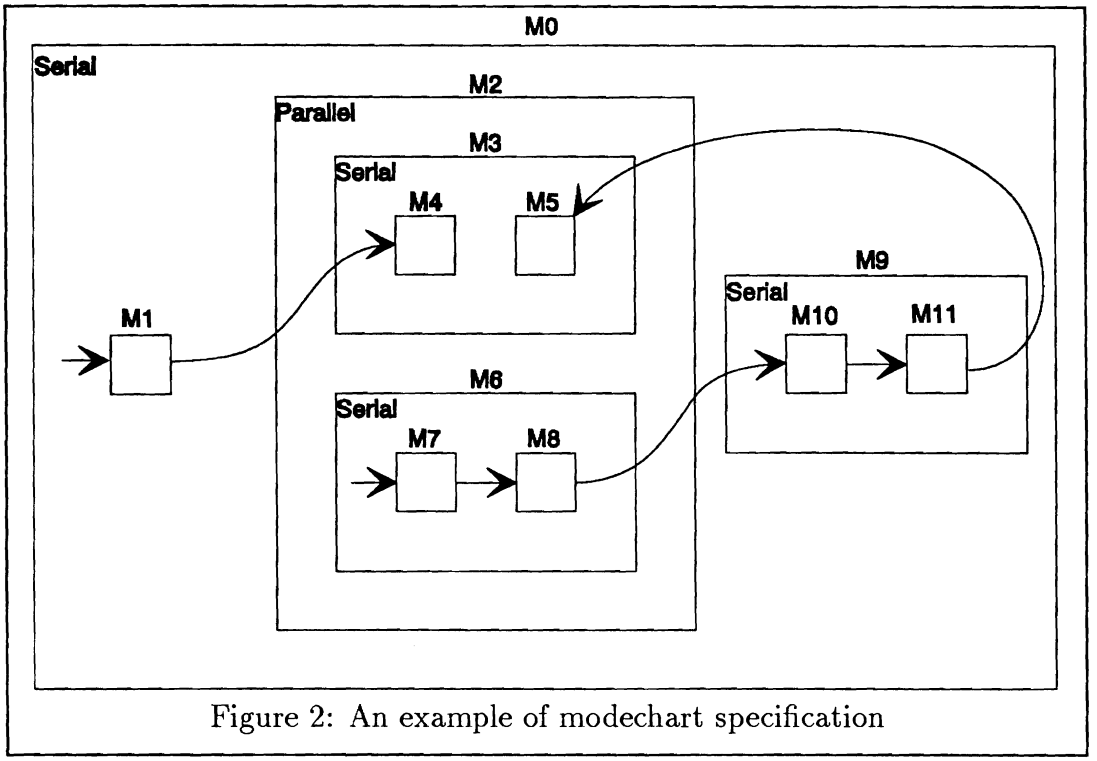


Figure 2: An example of modechart specification

Modechart includes the notion of well-formedness. In order to be well formed, a modechart must have: modes are properly nested; parallel modes with no initial state, serial modes with at most one initial state, and the smallest mode that includes both the source and the destination of a transition must be serial. In addition, serial modes must comply with the Unambiguous Designation of Initial Mode (UDIM) condition. This condition requires that, whenever a serial mode is entered, the child mode to be entered be unambiguously identified either specifically by the transition, or by the presence of an initial mode. In figure 2, for example, M3 complies with the UDIM condition because it can only be entered explicitly by either the transition originating from M1 or the transition originating from M11. This would not be the case if, for example, there was a transition from M10 to M6 or a descendent of M6 because M3 would have to be entered simultaneously to M6, but its initial mode would not be uniquely identified.

Modechart includes the notion of action which allows a system to influence its environment. While transitions are instantaneous, actions take a non zero, but finite amount of system resources. Although the term is not formally defined, the intent is that actions can be used to pace transitions and avoid infinitely many transitions occurring in a finite amount of time. At most one action is associated with each mode. It is not clear why this restriction is made, nor whether or not an action associated with a parent node is automatically associated with the children or vice versa. In any case, multiple actions

can be associated with the same mode by creating parallel children, but this mechanism can become cumbersome if the mode is not a primary mode.

The action associated with a mode is started upon mode entry. Since actions are not instantaneous, the question of what happens when an action is not completed at the time a transition out of the mode is to be executed needs to be addressed. Three interpretations are possible: the transition can be suspended until the action is completed, the action can be aborted or the action can be terminated in the next mode. It is possible in modechart to express the first interpretation by expanding the mode into a serial mode with two children, one where the action is in progress and one where the action is completed, and disable the transition while the system is in the first child. However there is no syntax available to differentiate between the second and third interpretations. In our example, we will adhere to the second interpretation and assume that the action ends concurrently with the mode exit. This avoids the anomaly of having an action that never ends.

Modechart also includes the notion of state variables, of type boolean. Values are assigned to these variables by actions.

Events play an important role in modechart. There are several categories of events:

- External events are identified by the Greek letter Ω followed by the event name in uppercase letters.

- The start and end of actions are events identified by the name of the action preceded by $\uparrow$ and $\downarrow$ respectively.

- An event is associated to the change of the value of a variable, not to the assignment of a value. e.g. (S:=T) represents the event of variable S changing from False to True and (S:=F) represents the event of variable S changing from True to False.

- An event is associated with each transition being taken. (M1-M2) represents the occurrence of a transition from mode M1 to mode M2.

- Finally, an event is associated with the entry and exit of each mode, these events are denoted (M:=T) for the entry and (M:=F) for the exit of mode M.

A triggering condition is associated to each transition. A triggering condition is of the form: $c_1 \vee c_2 \vee \cdots \vee c_k$ where each disjunct c_k is either a time condition of the form (r, d), in which case the transition is taken anytime between r and d time units after entering the mode, or a conjunction of the form $p_1 \wedge p_2 \wedge \cdots \wedge p_m$, in which case the transition occurs as soon as all the conjuncts are true. Each conjunct can be either a state variable being True (S), a state variable being False ($\overline{S}$), the system being in one of several states ($\text{in}\{M1, M2, \cdots Mk\}$), or the occurrence of an event E.

The separation of timing expressions from triggering conditions is meant to simplify the generation of computation graphs as we shall see, but it does seem somewhat unnatural. It is however possible to achieve the result of composing triggering conditions and delays by creating additional modes.

The semantics for modechart can be expressed in Real-Time Logics (RTL) [JMS 88]. RTL extends natural number arithmetic without multiplication with an occurrence function, denoted @, which represents the relationship between the events and their time of occurrence. The occurrence function is a mapping: $@ : E \times Z^+ \rightarrow N$, where $@(e,i) = t$ indicates that the event e occurred at time t for the i-th time. The @ functions is monotone on its second argument, which means that occurrence i+1 of an event cannot occur before occurrence i of that event. RTL includes existential and universal quantification of integer variables, but there is no event variable. As an example, the RTL formula:

$$\forall i : (@(\downarrow A, i) \leq @(\uparrow A, i) + 20) \wedge (\exists j : @(\uparrow B, j) \leq @(\downarrow A, i) + 50.$$

expresses the fact that each occurrence of action A takes at most 20 time units and that it is followed within 50 time units by an occurrence of action B.

Now we can proceed to express our railroad crossing example using modechart.

4.1 The railroad crossing example using modechart

The railroad crossing system is made of two main modes in parallel: the Monitor which, as its name indicates, keeps track of the train's position, and the Gate-Controller which controls the operation of the gate. Initially, the train is Before Crossing, and the gate is up, the Monitor and the Gate Controller are in the corresponding modes (BC and UP respectively), as indicated by the arrows. The complete modechart specification is shown in figure 3.

The Monitor works as follows. After and undetermined amount of time, as specified by the timing constraint $(0, \infty)$, the train approaches the crossing and the system moves from mode BC to mode NC (Near Crossing). Upon entering the NC mode, the Action A is executed which sets the state variable GATEUP to False; this action is completed within 50 time units (deadline 50). After at least 300 time units, the train enters the crossing and the monitor moves to the mode IC (In Crossing). Any time after that, the train leaves the crossing, and the monitor moves to the mode PC (Past Crossing). Upon entering PC, Action B is started which will set the state variable GATEUP to True within 50 time units. Anytime after a delay of 100 time units, meant to simulate inter-train separation, the monitor returns to the mode BC.

The Gate Controller works as follows. As soon as the variable GATEUP is set to False, it moves from the mode UP to the mode MD (Move Down). Simultaneously

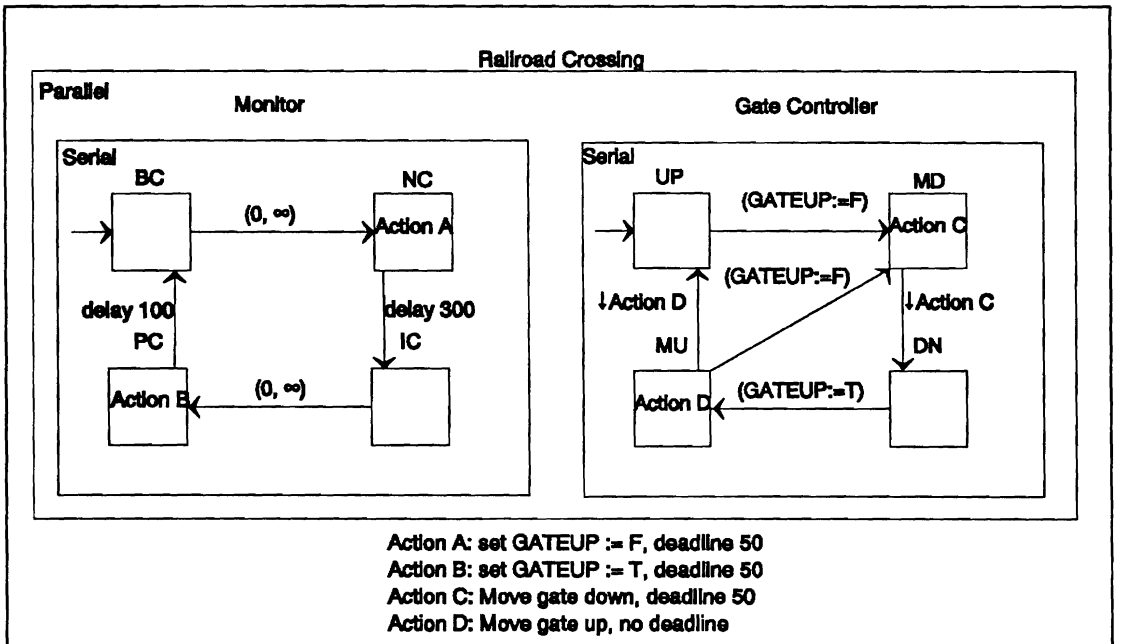


Figure 3: The railroad crossing in Modechart

starts action C which physically moves the gate down and turns on the flashing red lights. Upon completion of this action (i.e. the occurrence of the event ↓Action C), the Controller moves to the mode DN (Down). As soon as the variable GATEUP changes its value to True, the Controller executes a transition (DN-MU). Upon entering the mode MU (Move Up), Action D is started, which moves the gate up and turns out the lights. Upon completion of Action D (for which there is no deadline), the Controller moves back to the mode UP. If GATEUP becomes False while the it is still in mode MU, the Gate Controller would immediately move to mode MD.

It is worth mentioning that there is no way to specify what happens to Action D when the transition (MU-MD) is executed. We assume that it is immediately terminated (↓Action D occurs); however, nothing allows us to specify this part of the behavior.

The final touch to our example is the safety assertion: the system will always be in the mode DN whenever it is in the mode IC. This is expressed in RTL as follows:

$$\forall i \exists j : @((DN := T), j) \leq @((IC := T), i) \wedge @((IC := F), i) \leq @((DN := F), j)$$

For any occurrence i of the monitor entering mode IC, there is an occurrence j of the controller entering mode DN which occurs before the entry of mode IC, and the corresponding exit of mode DN occurs after the corresponding exit of mode IC. In other words, the gate goes down before IC and does not go up until after IC.

4.2 Proof method for Modechart

The methodology for proving properties of modechart specification is based on building a computation graph, and on the notion of a distance between two points of the graph.

The computation graph is very similar to a computation tree in that each node (called point) corresponds to the occurrence of an event and edges indicate direct or indirect causality. The main difference is that folding occurs: when two points are proven to be equivalent, they are merged together.It can be proven that the computation graph for a modechart system is finite. Intuitively, this is due to the absence of true recursion, and the fact that all the time expressions are constant. The maximum relevant history is limited to the maximum non-infinite delay. Since the number of modes is constant and each state variable has exactly two possible values, the number of different possible configurations in that limited history is limited and hence, when folding occurs, the graph is finite. A more formal proof can be found in [JS-88].

In the computation graph, each point is labelled with the state of the system (which modes it is in and the value of each system variable) and the event(s) that occur at this point. The computation tree is a spanning tree over the computation graph. Back edges are created when folding occurs.

The building of the graph is based on the notion of distance between points. The distance between two points A and B, noted distance(A, B), can be either a positive or a negative number. A positive number indicates a delay, the earliest time event B can happen after the occurrence of event A (i.e. A is anterior to B). A negative number indicates deadline, the earliest time B can happen before the occurrence of A (i.e. A is posterior to B). The term distance is unfortunate here because distance(A, B) + distance(B, A) $\leq$ 0, which is surprising as one would expect distance(A, B) = distance(B, A), or, possibly, distance(A, B) = -distance(B, A).

The distance function is calculated by adorning the computation tree with weighted edges corresponding to the time constraints (delay or deadline); we call them timing edges. If there is a minimum delay from event A to event B, then A must precede B in the tree and there is a timing edge from A to B labelled with that delay. If there is a deadline by which B must happen after A, then there is an edge from B to A weighted with minus that deadline. We call the weight of a timing path the sum of the weights of the timing edges along that path. By definition, if there is at least one timing path from A to B then distance (A, B) is the maximum weight of all those paths; otherwise, distance(A, B) is -∞.

A couple of remarks about this adorned tree. Obviously, every edge in the underlying computation tree will be adorned with a time delay of at least 0 since edges in the tree represent causality. Less obvious is the fact that the sum of the edges of an adorned cycle will be negative; otherwise, we would be facing a delay which is longer than the

deadline between the earliest and the latest points in the cycle. This property insures that the distance function is well defined.

The distance function allows us to prune out of the computation graph all the points that cannot be part of a computation due to timing constraints. If the distance path between two siblings A and B in the tree is strictly positive, then point B cannot be reached since point A must happen before it.

Two points of the tree can be folded together if they adhere to the distance equivalence condition. Intuitively, this means that the distance between the relevant events of the histories of A and B and the potential successors of A and B respectively, is either the same, or the difference is irrelevant because all the delays are already expired or one of the deadline has expired and hence the corresponding mode has been exited in both cases. Similarly, points with simultaneous events share the same deadline. The following formal presentation of this condition is somewhat different from that of the article and, hopefully, easier to grasp.

Consider a point A and let $\mathcal{R}(\mathcal{A})$ be the set of all of its predecessors, P, that mark an event that can be simultaneous with the event of A —i.e. distance (P, A) = 0. For each P in $\mathcal{R}$(A) let $\mathcal{E}$(P) be the corresponding event. The function $\mathcal{E}$ is naturally extended to sets so that we can write $\mathcal{E}(\mathcal{R}(\mathrm{A}))$ for the set of events possibly simultaneous to A and preceding A in the computation graph. Similarly, let $\mathcal{M}$(A) be the set of points preceding A in the graph when a mode that is active in A was entered for the last time. Formally, $\mathcal{M}$(A) is the set of points P such that if M is a mode of A, (M:=T) is an event of P and is not an event of any point of the tree between P (exclusive) and A (inclusive). Finally, let us note $\mathcal{S}$(A) the set of potential successors of a point A, i.e. the set of points whose label differ from the label of A by exactly one component, or the occurrence of a new event.

Two points A and B having the same label are said to be distance equivalent iff the following two conditions are true:

1. for each pair of points PA in $\mathcal{M}$(A), SA in $\mathcal{S}$(A) and the corresponding pair PB in $\mathcal{M}$(B) and SB in $\mathcal{S}$(B), let M be the corresponding mode, d the smallest deadline and r the longest delay of any transition out of M:

$$\begin{aligned} &(\mathrm{distance}(\mathrm{PA},\mathrm{SA}) = \mathrm{distance}(\mathrm{PB},\mathrm{SB}) \\ &\quad \wedge \mathrm{distance}(\mathrm{SA},\mathrm{PA}) = \mathrm{distance}(\mathrm{SB},\mathrm{PB})) \\ \vee\ &(\mathrm{d} = \infty \wedge \mathrm{distance}(\mathrm{PA},\mathrm{SA}) \geq \mathrm{r} \wedge \mathrm{distance}(\mathrm{PB},\mathrm{SB}) \geq \mathrm{r}) \\ \vee\ &(\mathrm{d} \neq \infty \wedge \mathrm{distance}(\mathrm{PA},\mathrm{SA}) > \mathrm{d} \wedge \mathrm{distance}(\mathrm{PB},\mathrm{SB}) > \mathrm{d}) \end{aligned}$$

2. $\mathcal{E}(\mathcal{R}(\mathrm{A})) = \mathcal{E}(\mathcal{R}(\mathrm{B}))$ and for each pair of points PA in $\mathcal{R}$(A), SA in $\mathcal{S}$(A) and the

corresponding pair PB in $\mathcal{R}$(B) and SB in $\mathcal{S}$(B) :

$$\begin{aligned} &\big(\text{distance}(\text{PA}, \text{SA}) = \text{distance}(\text{PB}, \text{SB}) = 0 \\ &\quad \wedge \text{distance}(\text{SA}, \text{PA}) = \text{distance}(\text{SB}, \text{PB})\big) \\ \vee \quad &(\text{distance}(\text{PA}, \text{SA}) > 0 \wedge \text{distance}(\text{PB}, \text{SB}) > 0) \end{aligned}$$

Some simple properties of a modechart, can be proven by examining the graph. For example, in the railroad example, we will see that there is no point where the Monitor is in mode IC while the Gate Controller is not in mode DN.

Most properties expressed as RTL formulas require that the graph preserve a relationship between event occurrences. This means that, if we are interested in comparing related occurrences of events e and g, for example @(e, i) and @(g, i+5), then in every cycle of the graph the number of occurrences of e must be the same as the number of occurrences of g. That being the case, the graph can be used to determine maximum and minimum separation between events as well as inclusion and exclusion of intervals.

The set of properties that can be verified using Modechart computation graph is a small subset of the set of properties that can be expressed in RTL. Since RTL is undecidable, the fact is not surprising.

4.3 Proof of safety of the railroad crossing

In order to prove the safety of the railroad crossing, we construct the computation graph shown in figure 4. We have used solid arrows to indicate edges of the computation graph and dotted arrows to indicate timing only edges. To reduce clutter, only the edges that bear a non-zero (timing) weight are labelled with their weight near the arrow head. Each point of the graph is numbered and labeled with the set of modes the system is in (in our case exactly two modes), the value of the unique state variable, GATEUP (abbreviated as GUP on the graph) and the events that mark the start and end of actions. Points that were potential successors of another point but were eliminated due to timing constraints are shown crossed out. The numeric label in front of each point is used only for reference purposes in commenting the proof. To reduce some of the complexity of the graph we have not shown some of the points that would have been immediately crossed out.

Let us look at the graph in more detail. Point 1 is the initial state, hence all the components are underlined. The only transition that can occur from this point is (BC-NC) i.e. a train approaches the crossing, this leads to point 2 and the start of action A. There are two possibilities after point 2: point 3 which is the end of action A, and point 4 where the train shows up in the crossing. Note that the edge from 3 to 2 is labeled

with -50, which indicates the deadline in the execution of the action A, and that the edge from 2 to 4 is labelled with 300, indicating the delay on the transition (NC-IC). These two edges constitute a path from 2 to 4 with a total weight of 250, thus the point 4 is unreachable due to timing constraints and is crossed out.

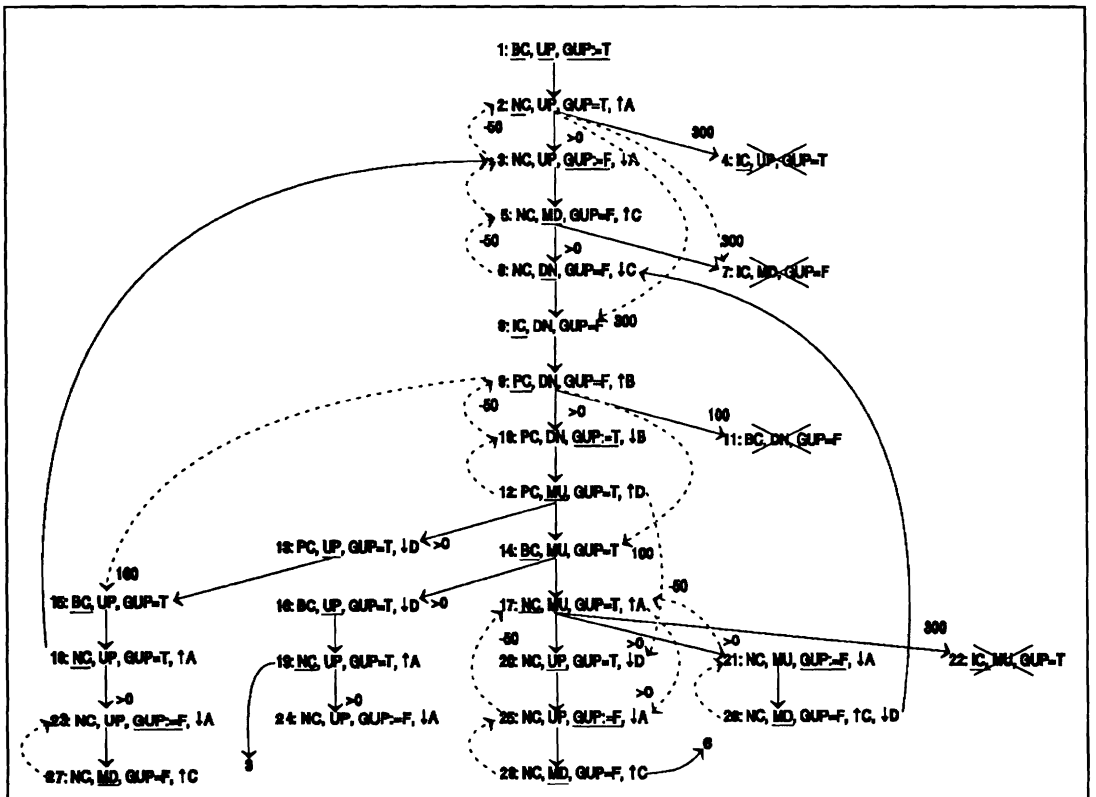


Figure 4: Computation graph for the railroad crossing example

The edge from 2 to 3 is labelled > 0, to indicate that the action A is not instantaneous and hence the events $\uparrow$A and $\downarrow$A cannot be simultaneous. There is an edge from 3 to 5 and an edge from 5 to 3, both edge have a timing weight of 0 which show the constraint that the action C is started immediately when the variable GATEUP changes from True to False at the end of action A.

Again, point 7 is a potential successor of point 5, but is eliminated due to the timing path 6-5-3-2-7 with a total weight of 200. The timing constraints are such that the gate has to be down at least 200 time units before the train enters the crossing.

The only successor to point 6 is point 8: the train enters the crossing and the only successor of point 8 is point 9: the train leaves the crossing. The delay of 100 on the transition (PC-BC) combined with the deadline of 50 on action B, which starts at point 9 and ends at point 10, allow us to eliminate point 11. Point 10 (GATEUP becomes

T) and point 11, start of action D are simultaneous. Two transitions are possible from point 12: (MU-UP), the gate is fully up and (PC-BC) the train has left the scene and the next one may show up at any time. At this point things become a bit complicated and we are going to limit ourselves to the left most branch. Similar arguments can be made for the other branches.

Since the action D is not instantaneous, points 12 and 13 cannot be simultaneous, hence the > 0 label. The only transition out of point 13 is (PC-BD) which has a delay of 100, but that's OK since point 15 has no sibling. The only transition that applies to point 15 is (BC-NC) which leads to point 18.

At this point one would like to apply the distance equivalence condition and fold points 18 and 2. Unfortunately it does not apply: $\mathcal{R}(18) = \{13, 15, 18\}$ and $\mathcal{E}(\mathcal{R}(18)) = \{(\mathrm{MU} - \mathrm{UP}), \downarrow \mathrm{D}, (\mathrm{PC} - \mathrm{BC}), (\mathrm{BC} - \mathrm{NC}), \uparrow \mathrm{A}\}$ while $\mathcal{R}(2) = \{1, 2\}$ and $\mathcal{E}(\mathcal{R}(2)) = \{(\mathrm{BC} - \mathrm{NC}), \uparrow \mathrm{A}\}$.

We go down one more step and compare points 23 and 5. Both nodes have the same label hence the comparison is valid. Both nodes have only one successor: $\mathcal{S}(23) = \{27\}$ and $\mathcal{S}(3) = \{5\}$.

The modes of both points are NC and UP, $\mathcal{M}(23) = \{13, 18\}$ $\mathcal{M}(3) = \{1, 2\}$. It is easy to see on the graph that distance$(13, 27) > 0$, distance$(1, 5) > 0$, distance$(18, 27) > 0$ and distance$(2, 5) > 0$: the first part of the condition is satisfied. $\mathcal{R}(23)$ is $\{23\}$, since point 18 cannot be simultaneous with point 23 and $\mathcal{E}(\mathcal{R}(23))$ is $\{\downarrow \mathrm{A}, (\mathrm{GATEUP} := \mathrm{F})\}$. Similarly $\mathcal{R}(3) = \{3\}$ and $\mathcal{E}(\mathcal{R}(3)) = \{\downarrow \mathrm{A}, (\mathrm{GATEUP} := \mathrm{F})\}$. Again, by looking at the graph we see that distance(23, 27) = distance(3, 5) = distance(27, 23) = distance(5, 3) = 0. Hence the second part of the condition is satisfied. Points 3 and 23 are distance equivalent and can be folded together as shown by the edge 18, 3 on the graph. Similar arguments justify the other back edges of the graph. The graph is complete and it is finite.

At this point it is obvious that all the points labelled with IC are either unreachable (4, 7, 22) or labelled with IC as well, hence our safety property is satisfied.

One observation about this proof: while on this simple example we did not have to look any deeper than one level below the siblings to detect unreachable points (case of the path 20-25-17-22) it is not clear when one should stop, particularly when folding occurs.

5 Calculus for Communicating Shared Resources

The third method for specifying and proving properties of real-time distributed system that we are investigating, CCSR, has been developed by Richard Gerber and Insup Lee at the University of Pennsylvania [GL90a]. Unlike the previous two approaches,

CCSR does not have a graphical representation. A system in CCSR is specified as an algebraic expression. Superficially, CCSR resembles other process algebra like Hoare's CSP [BHR84, Hoa85] and Milner's CCR [Mil89], but it is much more complete in its treatment of inter-process relationships, not simply communication. In particular it allows the definition of resources and allocation of events to resources which permits a formal treatment of interleaving. A formal specification of CCSR in terms of operational semantics can be found in [GL90b]. What follows is more of an intuitive introduction to it.

The atomic thing happening in a system is an event. Unlike modechart, events are not. They take exactly one unit of time. Of course, several events may occur at the same time. A simultaneous occurrence of multiple events is called an action and is written as e1 ... en. Most of the power of CCSR resides in its capability to express conditions under which some actions may or may not happen in a system evolution.

As the name indicates, CCSR deals with resources. Events are executed by resources. Each event in the system can be executed by one and only one resource. In addition, a resource can execute at most one event at any point in time. In the case where no event of a given resource is executed by any process, that resource is considered to idle.

Using resource, true parallelism, in which events execute on different resources, can be distinguished from interleaving, in which events are executed sequentially on the same resource. CCSR does not allow the specification of pools of similar resources allowing, an event to occur as long as one of the resources able to execute it is free.

A process in CCSR is specified as an expression built out of actions, variables and operators. As usual, it is assumed that there is an infinite, countable number of variables, and that a variable stands for any CCSR expression. In the following description of the CCSR operators we use P and Q as variables standing for any legal CCSR expression.

At any point in time the behavior of a CCSR process must be specified. A process unable to perform any action is in a deadlock state. The canonical form of such a process is NIL. On the other hand, one may want to specify that a process may wait, not requiring any event, for a while. This is for example the case when a process is doing internal calculation, or to specify a delay or a behavior taking more than a single unit of time. This is expressed as an empty action and is written $\{\}$ or $\emptyset$. Note the difference between $\emptyset$ and NIL. While $\emptyset$ allows progress to be made and merely marks one unit of time, NIL indicates that nothing further may occur and, should this be the only choice, the system will be totally unable to make any further progress.

The prefix operator (:) is used to specify the passing of time. For example, $\{a\} : P$ indicates that at the very next instant event a must be executed (not necessarily alone, depending on the context), this will take one time unit, then the system will behave as specified in P.

The choice operator, written as +, indicates possible alternatives in the behavior of a process. For example, $(\{a\} : P) + (\{b\} : Q)$ specifies that a may be executed, in which case the system will behave like P, or {b} will happen, in which case the system will behave like Q immediately after. The decision between a and b is influenced by the notions of synchronization and priority that we shall see later. If both actions are equally likely in the context they are in, then the choice is non-deterministic.

Before we talk about the scope operator, we need look at the notion of termination. The termination of a process is specified by including a check mark ($\surd$) as part of an action. The rule for the propagation of termination depends on the operators that compose the expression, and we shall see them as we study the operators.

The scope operator is a multi-purpose operator that is used to specify general timing constraints. It has the form $P\Delta_t^B(C, D, I)$ where P, C(Completion), D(Deadline) and I(Interrupt) are CCSR expression, t is a positive number or infinity and B is either $\{\surd\}$ or $\emptyset$. The semantics is as follows. When P terminates (within t time units, or ever when $t = \infty$) the system behaves as C. If t is finite and P does not terminate in t time units, then, at time t+1, the system starts behaving as D. In addition, I is an alternative choice at every step of the execution of P as well as to the start of D. Informally, t is a deadline for the execution of P, C is the behavior to follow after a successful completion of P, D is the exception behavior in case the deadline expires, and I is a potential interrupt that can occur anytime. Of course it is rare for all these options to be used at the same time, and NIL can be used to disable any of them. The superscript B is used to specify the semantics of D with regard to termination. When $B = \{\surd\}$, the scope operator propagates the termination of P; when $B = \emptyset$, the termination of P always leads to the start of Q and does not propagate.

The operators we have seen so far can only specify sequential behaviors. The parallel operator allows us to specify multiple sequences of actions occurring in parallel. A parallel process is of the form $P\ {}_I\|_J\ Q$ where P and Q are the two processes that execute in parallel, I is the set of resources used by P and J is the set of resources used by Q. At every instant, both P and Q must execute one action. P and Q must interleave to execute actions of $I \cap J$ and may communicate when P executes an action of I-J and Q executes an action of J-I.

Note that the specification of the resource sets I and J as part of the parallel statement is useful to formally manipulate CCSR expressions, but does not add to the expressiveness of the language. In fact all the "interesting" behaviors of the parallel construct could be expressed similarly if I and J were defined as a function of P and Q.

Communication and synchronization between processes is specified via the notion of connection sets. A connection set is a set of events such that any action of the system contains either all or none of the events of the set. Obviously, for a connection set to be

well formed, it must not contain two events from the same resource.

The hiding operator, noted "\", allows us to abstract out implementation details by preventing the actual events being executed by a process from being visible by other processes, hidden events cannot be used for synchronization. However, the semantics of the hiding process does not imply freeing of resources and must not violate timing properties. For these reasons and in order to be able to properly apply priorities, (as we shall see), hidden events are replaced by canonical events, noted "τ_j^i", that indicate that the resource j is busy executing an unknown event at priority i.

The τ events are cumbersome to carry around, and are only useful in preventing a parallel process from competing for a resource used by one of them at the microscopic level. The closure operator is an alternative at a macroscopic level: $[P]_J$ indicates that process P fully utilizes every resource of the set J, and therefore none is available to any other process.

Powerful as they may be, the above operators do not allow us to specify infinite behavior. This is done via the fixpoint operator fix(X.P), where X is a variable and P may or may not contain X as a free variable. The semantics is the obvious one of recursion. One can also use recursive definition of processes to achieve the same thing.

One of the most significant features of CCSR is the notion of priority and preemption. Each event has a priority, which is a natural number. The priority of an action is a vector of priorities, one component for each resource used in the action. Priorities are compared resource wise. For a closed system, an action will preempt another one if and only if it has higher priority in every resource. [GL90a] gives a definition of preemption for non-closed systems which is crucial to allow the algebraic manipulation of CCSR expression, but, for the sake of space and since our example does not make use of priorities we will not go into those details.

Let us now apply CCSR to the railroad crossing example.

5.1 Railroad crossing using CCSR

The following δ function will be useful to simplify the expression of the example:

$$\delta_t(A,P) \stackrel{\text{def}}{=} \begin{cases} P & \text{if } t = 0 \\ P + (A : d_\infty(A,P)) & \text{if } t = \infty \\ P + (A : d_{t-1}(A,P)) & \text{otherwise} \end{cases}$$

The δ function expresses a deadline: P will occur at the latest after t time units; the function is extended to $t = \infty$ with the obvious semantics of an infinite deadline. A is the action being executed while waiting.

The Railroad crossing defined using CCSR is shown in figure 5. It is worth noting the terseness of the definition. Let us look at it more carefully.

The definition for the Train works as follows. The idle action is always a choice until the event nc! occurs, i.e. the train appears near the crossing. This event, followed by idling for 299 units of time, totals up as a delay of 300. After that, the idling is again an alternative until ic! occurs. Then more idle until pc! which will be followed by 99 units of time before the process starts again recursively.

$$\begin{aligned}
\text{Train} &\stackrel{\text{def}}{=} \delta_\infty(\tau_1, \{\text{nc!}\} : \tau_1^{299} : \delta_\infty(\tau_1, \{\text{ic!}\} : \delta_\infty(\tau_1, \{\text{pc!}\} : \tau_1^{99} : \text{Train})))\\
\text{Control} &\stackrel{\text{def}}{=} \delta_\infty(\tau_2, \{\text{nc?}\} : \delta_{49}(\tau_2, \{\text{guf!}\} : \delta_\infty(\tau_2, \{\text{pc?}\} : \delta_{49}(\tau_2, \{\text{gut!}\} : \text{Control}))))\\
\text{Gate} &\stackrel{\text{def}}{=} \delta_\infty(\tau_3, \{\text{guf?}\} : \text{fix}(\text{GUF}.\{\text{md!}\} : \delta_{48}(\tau_3, \{\text{dn!}\} : \delta_\infty(\tau_3, \{\text{gut?}\}\\
&\qquad : \{\text{mu!}\} : \text{fix}(\text{MU}.(\emptyset : \text{MU}) + (\{\text{up!}\} : \text{Gate}) + (\{\text{guf?}\} : \text{GUF})))))
\end{aligned}$$

Resources :

Resource1 : nc!, ic!, pc!

Resource2 : nc?, guf!, pc?, gut!

Resource3 : guf?, dn!, gut?, up!, md!, mu!

Connection sets:

Inter-resource:{nc!, nc?}, {pc!, pc?}, {guf!, guf?}, {gut!, gut?}

Local:{ic!}, {up!}, {dn!}

Priorities:

All events have priority 0.

$$\text{Crossing} = (\text{Train}_{\,1} \|_2 \text{ Control})_{\,12} \|_3 \text{ Gate}$$

Figure 5: The railroad crossing using CCSR

The control will also busy-wait doing nothing until nc? occurs. The connection set insures that both events will occur simultaneously, or neither will occur. Within 49 units of time after this point (i.e. a deadline of 50) the event gut! will occur. Then the control will be able to idle until the event pc? occurs which is synchronized with pc!. Within 50 units after that, the event gut! will have occurred, after which the whole process will start over again.

The gate process is a bit more complicated but essentially similar. Idle until the event guf?, which is synchronized with guf!, occurs. The event md! indicates that the gate starts moving down, and within 49 time units the event dn! occurs. At this point

the process will idle until the event gut! occurs. The events gut! and gut? synchronize and are followed by the event mu! which indicates that the gate starts moving up. The gate moves up (as materialized by the fix point of the MU variable) until one of two things happen: either the event guf? is executed and the system will loop back to move the gate down (fix point of GUF) or the event up! will happen and the process will repeat the whole gate behavior.

This example uses only a small subset of the features of CCSR. In particular there is no interleaving, as shown by the resource assignments and priorities are essentially unused.

Before we try to prove anything about this application, let us study the proof system of CCSR.

5.2 Proof system for CCSR

The proof system for CCSR is based on the concept of strong bisimulation [Bol87, Mil89], extended to take priorities into account. Intuitively, two systems are strongly bisimilar if they can mimic each other at every step. In other words, two processes P and Q are strongly bisimilar iff every action that one can perform can be performed by the other process and the two resulting processes are also strongly bisimilar. When the terms bisimilar or bisimulation are used in the remainder of this paper they must be taken as meaning strongly bisimilar and strong bisimulation.

The key property of the (prioritized) bisimulation is that it is a congruence with regard to the CCSR operators. This will, of course, allow us to prove that two sets of operands are bisimilar, and infer that the results of applying the same operator to both sets of operands are bisimilar.

The proof system for CCSR is fully axiomatized. Because a formal presentation of the 20 axioms would require new notation and would be extremely detailed, again we will content ourselves with the intuition behind them.

Four axioms designate NIL as the identity element for choice, and a zero element for the parallel, hide and closure operators. In addition, NIL as the first operand of a scope operator forces the result to be the fourth operand (i.e. the interrupt branch).

The choice operator is idempotent.

Both choice and parallel operator are commutative and associative, although the associativity for parallel requires recalculating the resource sets.

All the operators except prefix are distributive over choice. The reason prefix is not distributive is subtle but interesting. The expression {a}: (P + Q) requires a to be executed, then a choice is made between P and Q the choice may be influenced by the environment; the expression ({a} : P) + ({a} : Q) requires that a choice be made

between the two branches at the time the event a is executed, since {a} appears at the head of both branches, the choice will be totally non-deterministic.

Five of the axioms apply when every operand is a prefix expression and work by examining these first actions.

For the choice operator, if one action preempts the other, the preempted branch can be eliminated.

For the parallel operator, if the two actions are compatible (i.e. they will not require that a resource execute two events) the two actions can be combined (union of the two sets of events with $\surd$ included iff it is present in both actions) as a single prefix and the parallel operator will be applied to the tails. If the actions are not compatible, the whole expression is equivalent to NIL.

The presence or absence of $\surd$ in the first action of the first operand of the scope operator combined to the value of the t parameter may allow us to reduce the operation to a choice or "unroll" the scope for one unit of time.

For the closure operator, a τ is added to the action to saturate free resources and the closure is applied to the tail unless one of the events is outside the set of closed resources, in which case the whole expression is turned into NIL.

For the hide operator, if the connection sets can still be satisfied after application of the operator, then τ can be substituted for the hidden events and the operator is applied to the tail, otherwise the whole expression is equivalent to NIL.

Finally, (immediately) nested hide operators can be absorbed if their resource set is a subset of the resource set of the outer operator.

For expressions with finite behaviors, this set of axioms can be shown to be complete and to converge to a normal form. This is not the case for infinite behaviors and it is not even clear if the equivalence problem is decidable.

5.3 Proof of correctness of the railroad crossing

In order to prove the correctness of a CCSR expression, we need to compare it to another expression that is known to be correct. In our case, we want to prove that, if we eliminate all the details of the implementation, the system will be such that every execution of the ic! event will be preceded by an execution of the dn! event and the next mu! event will follow the pc! event. This is translated in CCSR as:

$$\text{correct1} \stackrel{\text{def}}{=} \{\text{dn!}\} : \{\text{ic!}\} : \{\text{pc!}\} : \{\text{mu!}\} : \text{correct1}$$

Simple enough! Unfortunately, we cannot ignore the real-time aspects of the system and in particular the minimum separations; hence, we will prove that:

$$[\text{Crossing} \setminus \{\text{nc? nc! guf? guf! pc! gut? gut! md!}\}]_{\{123\}} = \text{correct}$$

where "correct" has the form:

$$\text{correct} \stackrel{\text{def}}{=} \emptyset^{\text{p}} : \{\text{dn!}\} : \emptyset^{\text{q}} : \{\text{ic!}\} : \emptyset^{\text{r}} : \{\text{pc!}\} : \emptyset^{\text{s}} : \{\text{mu!}\} : \emptyset^{\text{t}} : \text{correct}$$

with $\text{p}, \text{q}, \text{r}, \text{s}, \text{t} \in \text{N} \cup \{\infty\}$.

In order to make the proof more scrutable, we have decomposed the definitions of the system; it is easy to see that the new definitions are equivalent to the previous ones. In the body of the proof, we will omit the resource set indices of the parallel operator because their value is obvious. We will use $\mathcal{T}$ to stand for $\{\tau_1\, \tau_2\, \tau_3\}$. Finally, we will use the following conventions: the prefix operator has the highest precedence, followed by choice, parallel has the lowest; choice and parallel are left associative while prefix associates to the right.

$$\begin{aligned}
\text{Train} &\stackrel{\text{def}}{=} \delta_\infty(\tau_1, \{\text{nc!}\} : \text{T1}) \\
\text{T1} &\stackrel{\text{def}}{=} \tau_1^{299} : \text{T2} \\
\text{T2} &\stackrel{\text{def}}{=} \delta_\infty(\tau_1, \{\text{ic!}\} : \text{T3}) \\
\text{T3} &\stackrel{\text{def}}{=} \delta_\infty(\tau_1, \{\text{pc!}\} : \text{T4}) \\
\text{T4} &\stackrel{\text{def}}{=} \tau_1^{99} : \text{Train} \\
\\
\text{Cntrl} &\stackrel{\text{def}}{=} \delta_\infty(\tau_2, \{\text{nc?}\} : \text{C1}) \\
\text{C1} &\stackrel{\text{def}}{=} \delta_{49}(\tau_2, \{\text{guf!}\} : \text{C2}) \\
\text{C2} &\stackrel{\text{def}}{=} \delta_\infty(\tau_2, \{\text{pc?}\} : \text{C3}) \\
\text{C3} &\stackrel{\text{def}}{=} \delta_{49}(\tau_2, \{\text{gut!}\} : \text{Cntrl}) \\
\\
\text{Gate} &\stackrel{\text{def}}{=} \delta_\infty(\tau_3, \{\text{guf?}\} : \text{G1}) \\
\text{G1} &\stackrel{\text{def}}{=} \{\text{md!}\} : \delta_{48}(\tau_3, \{\text{dn!}\} : \text{G2}) \\
\text{G2} &\stackrel{\text{def}}{=} \delta_\infty(\tau_3, \{\text{gut?}\} : \{\text{mu!}\} : \text{G3}) \\
\text{G3} &\stackrel{\text{def}}{=} \tau_3 : \text{G3} + \{\text{up!}\} : \text{Gate} + \{\text{guf?}\} : \text{G1}
\end{aligned}$$

We will detail the first steps of the proof, but later, for the sake of space and as we are re-using the same mechanisms, we will skip over most of the details.

If we expand the definition of Crossing and apply the definition of δ we obtain the following expression:

$$\text{Crossing} = \tau_1 : \text{Train} + \{\text{nc!}\} : \text{T1} \parallel \tau_2 : \text{Control} + \{\text{nc!}\} : \text{C1} \parallel \tau_3 : \text{Gate} + \{\text{guf?}\} : \text{G1}$$

By distributing the parallel operator over choice we obtain 8 terms:

$$(\tau_1 : \text{Train} \parallel \tau_2 : \text{Cntrl} \parallel \tau_3 : \text{Gate}) + (\tau_1 : \text{Train} \parallel \{\text{nc?}\} : \text{C1} \parallel \tau_3 : \text{Gate})$$

$$+ \quad (\{\text{nc!}\} : \text{T1} \parallel \{\text{nc?}\} : \text{C1} \parallel \tau_3 : \text{Gate}) + (\{\text{nc!}\} : \text{T1} \parallel \tau_2 : \text{Cntrl} \parallel \tau_3 : \text{Gate})$$
$$+ \quad (\tau_1 : \text{Train} \parallel \tau_2 : \text{Cntrl} \parallel \{\text{guf?}\} : \text{G1}) + (\tau_1 : \text{Train} \parallel \{\text{nc?}\} : \text{C1} \parallel \{\text{guf?}\} : \text{G1})$$
$$+ \quad (\{\text{nc!}\} : \text{T1} \parallel \{\text{nc?}\} : \text{C1} \parallel \{\text{guf?}\} : \text{G1}) + (\{\text{nc!}\} : \text{T1} \parallel \tau_2 : \text{Cntrl} \parallel \{\text{guf?}\} : \text{G1})$$

We take the union of the first actions and "push" the parallel operator down:

$$\begin{aligned} & \mathcal{T} : (\text{Train} \parallel \text{Cntrl} \parallel \text{Gate}) + \{\tau_1\, \text{nc?}\, \tau_3\} : (\text{Train} \parallel \text{C1} \parallel \text{Gate}) \\ + \quad & \{\text{nc!}\, \text{nc?}\, \tau_3\} : (\text{T1} \parallel \text{C1} \parallel \text{Gate}) + \{\text{nc!}\, \tau_2\, \tau_3\} : (\text{T1} \parallel \text{Cntrl} \parallel \text{Gate}) \\ + \quad & \{\tau_1\, \tau_2\, \text{guf?}\} : (\text{Train} \parallel \text{Cntrl} \parallel \text{G1}) + \{\tau_1\, \text{nc?}\, \text{guf?}\} : (\text{Train} \parallel \text{C1} \parallel \text{G1}) \\ + \quad & \{\text{nc!}\, \text{nc?}\, \text{guf?}\} : (\text{T1} \parallel \text{C1} \parallel \text{G1}) + \{\text{nc!}\, \tau_2\, \text{guf?}\} : (\text{T1} \parallel \text{Cntrl} \parallel \text{G1}) \end{aligned}$$

Using the fact that {nc! nc?} and {guf! guf?} are connection sets, we replace by NIL all inconsistent actions, i.e. actions that contain one and only one element of either set:

$$\begin{aligned} \mathcal{T} : (\text{Train} \parallel \text{Cntrl} \parallel \text{Gate}) + \text{NIL} + \{\text{nc!}\, \text{nc?}\, \tau_3\} : (\text{T1} \parallel \text{C1} \parallel \text{Gate}) \\ + \text{NIL} + \text{NIL} + \text{NIL} + \text{NIL} + \text{NIL} \end{aligned}$$

Eliminating NIL from the choice and using the definition of Crossing, we obtain:

$$\text{Crossing} = (\mathcal{T} : \text{Crossing}) + \{\text{nc!}\, \text{nc?}\, \tau_3\} : (\text{T1} \parallel \text{C1} \parallel \text{Gate})$$

Making use of the definition of δ will give us:

$$\text{Crossing} = \delta_\infty(\mathcal{T}, \{\text{nc!}\, \text{nc?}\, \tau_3\} : (\text{T1} \parallel \text{C1} \parallel \text{Gate}))$$

The set of steps: expand δ, distribute parallel over choice, combine first actions, eliminate inconsistent actions and, optionally apply the definition of δ, will be repeatedly used in the remainder of the proof, moving forward through the prefix expressions; we will refer to it as expanding a term.

Expanding (T1 $\parallel$ C1 $\parallel$ Gate) once produces a choice of 50 terms, all of the form:

$$\mathcal{T}^{\mathrm{k}} : \{\tau_1\, \text{guf!}\, \text{guf?}\} : (\tau_1^{298-\mathrm{k}} : \text{T2} \parallel \text{C2} \parallel \text{G1}) \text{ where } 0 \leq \mathrm{k} \leq 49;$$

which we will process generically.

After four more applications of the expansion mechanism we get:

$$\text{Crossing} = \delta_\infty(\mathcal{T}, \{\text{nc!}\, \text{nc?}\, \tau_3\} : \mathcal{T}^{\mathrm{k}} : \{\tau_1\, \text{guf!}\, \text{guf?}\} : \{\tau_1\, \tau_2\, \text{md!}\} : \mathcal{T}^{296-\mathrm{k}-\mathrm{m}} : \text{X1}$$

with $0 \leq \mathrm{k} \leq 49$ and $0 \leq \mathrm{m} \leq 48$ and $\text{X1} \stackrel{\text{def}}{=} \text{T2} \parallel \text{C2} \parallel \text{G2}$.

Similarly expanding X1 in 8 steps produces:

$$\begin{aligned}\text{X1} &= \delta_\infty(\mathcal{T}, \{\text{ic!}\,\tau_2\,\tau_3\} : \delta_\infty(\mathcal{T}, \{\text{pc!}\,\text{pc?}\,\tau_3\} : \mathcal{T}^{\text{n}} : \{\tau_1\,\text{gut!}\,\text{gut?}\} : \{\tau_1\,\tau_2\,\text{mu!}\} \\ &\quad : (\mathcal{T}^{\text{p}} : \{\tau_1\,\tau_2\,\text{up!}\} : \mathcal{T}^{96-\text{n}-\text{p}} : \text{Crossing} + \mathcal{T}^{96-\text{n}} : \text{X2})));\end{aligned}$$

where $0 \leq \text{n} \leq 49$ and $0 \leq \text{p} \leq 96 - \text{n}$ and $\text{X2} \stackrel{\text{def}}{=} (\text{Train} \parallel \text{Cntrl} \parallel \text{G3})$.

Expanding X2 and substituting X1 for its definition (T2 || C2 || G2):

$$\begin{aligned}\text{X2} = \delta_\infty(&\mathcal{T}, \\ &\{\tau_1\,\tau_2\,\text{up!}\} : \text{Crossing} \\ &+\{nc!\; nc?\; \tau_3\} : \\ &\quad (\mathcal{T}^q : \{\tau_1\;\tau_2\;up!\} : \mathcal{T}^r : \{\tau_1\;guf!\;guf?\} : \{\tau_1\;\tau_2\;md!\} : \mathcal{T}^s \\ &\qquad : \{\tau_1\,\tau_2\,\text{dn!}\} : \mathcal{T}^{295-\text{q}-\text{r}-\text{s}} : \text{X1} \\ &\quad + \\ &\quad \mathcal{T}^{\text{t}} : \{\tau_1\,\text{gut!}\,\text{gut?}\} : \{\tau_1\,\tau_2\,\text{mu!}\} : \mathcal{T}^{\text{u}} : \{\tau_1\,\tau_2\,\text{dn!}\} : \mathcal{T}^{296-\text{t}-\text{u}} : \text{X1}) \\ &+\{nc!\; nc?\; up!\} : \mathcal{T}^v : \{\tau_1\;gut!\;gut?\} : \{\tau_1\;\tau_2\;mu!\} : \mathcal{T}^w \\ &\qquad : \{\tau_1\,\tau_2\,\text{dn!}\} : \mathcal{T}^{296-\text{v}-\text{w}} : \text{X1})\end{aligned}$$

with $0 \leq \text{q} \leq 48$; $0 \leq \text{r} \leq 48 - \text{q}$; $0 \leq \text{s} \leq 48$; $0 \leq \text{t} \leq 49$; $0 \leq \text{u} \leq 48$; $0 \leq \text{v} \leq 49$; $0 \leq \text{w} \leq 48$.

At this point, we define:

$$\begin{aligned}\text{Crossing}' &\stackrel{\text{def}}{=} [\text{Crossing} \setminus \{\text{nc? nc! guf? guf! pc? gut? gut! md!}\}]_{\{123\}}, \\ \text{X1}' &\stackrel{\text{def}}{=} [\text{X1} \setminus \{\text{nc? nc! guf? guf! pc? gut? gut! md!}\}]_{\{123\}}\end{aligned}$$

and similarly for X2′, We apply the hide operation that allow us to replace all the hidden events by the appropriate τ. We then apply the closure operator that allow us to eliminate the $\mathcal{T}$ altogether and we obtain:

$$\begin{aligned}\text{Crossing}' &= \delta_\infty(\emptyset, \emptyset : \emptyset^{\text{k}} : \emptyset : \emptyset : \emptyset^{\text{m}} : \{\text{dn!}\} : \emptyset^{296-\text{k}-\text{m}} : \text{X1}') \\ &= \delta_\infty(\emptyset, \emptyset^{\text{a}} : \{\text{dn!}\} : \emptyset^{299-\text{a}} : \text{X1}') \text{with} 3 \leq \text{a} \leq 100 \\ \text{X1}' &= \delta_\infty(\emptyset, \{\text{ic!}\} : \delta_\infty(\emptyset, \{\text{pc!}\} : \emptyset^{\text{n}} : \emptyset : \{\text{mu!}\} \\ &\quad : (\emptyset^{\text{p}} : \emptyset : \emptyset^{96-\text{n}-\text{p}} : \text{Crossing}' + \emptyset^{96-\text{n}} : \text{X2}')) \\ &= \delta_\infty(\emptyset, \{\text{ic!}\} : \delta_\infty(\emptyset, \{\text{pc!}\} : \emptyset^{\text{b}} : \{\text{mu!}\} \\ &\quad : (\emptyset^{99-\text{b}} : \text{Crossing}' + \emptyset 98 - \text{b} : \text{X2}')) \text{ with } 2 \leq \text{b} \leq 51 \\ \text{X2}' &= \delta_\infty(\emptyset, \emptyset : \text{Crossing}' + \emptyset : (\emptyset^{\text{q}} : \emptyset : \emptyset^{\text{r}} : \emptyset : \emptyset : \emptyset^{\text{s}}\end{aligned}$$

$$\emptyset^{t} : \emptyset : \emptyset : \emptyset^{u} : \{dn!\} : \emptyset^{296-t-u} : X1) +$$
$$\emptyset : \emptyset^{v} : \emptyset : \emptyset : \emptyset^{w} : \{dn!\} : \emptyset^{296-v-w} : X1')$$

Summing up the exponents and using the idempotence of choice twice we obtain:

$$X2' = \delta_{\infty}(\emptyset, \emptyset : \text{Crossing}' + \emptyset : \emptyset^{c} : \{dn!\} : \emptyset^{299-c} : X1') \text{ with } 3 \leq c \leq 100$$

Substituting Crossing′ for its value and using the idempotence of choice gives:

$$X2' = \delta_{\infty}(\emptyset, \emptyset : \text{Crossing}')$$

and hence:

$$X1' = \delta_{\infty}\Big(\emptyset, \{ic!\} : \delta_{\infty}(\emptyset, \{pc!\} : \emptyset^{b} : \{mu!\} : (\emptyset^{99-b} : \text{Crossing}' + \emptyset^{98-b} : \text{Crossing}')\Big)$$

It follows immediately from the definition of δ that $\delta_{\infty}(A, A : P) = A : \delta_{\infty}(A, P)$ and that δ is idempotent on its second argument therefore, using also the idempotence of choice:

$$X1' = \delta_{\infty}(\emptyset, \{ic!\} : \delta_{\infty}(\emptyset, \{pc!\} : \emptyset^{b} : \{mu!\} : \emptyset^{99-b} : \text{Crossing}'))$$

Moving into the expression for Crossing′ we obtain:

$$\text{Crossing}' = \delta_{\infty}(\emptyset, \emptyset^{a} : \{dn!\} : \emptyset^{296-a} : \delta_{\infty}(\emptyset, \{ic!\} : \delta_{\infty}(\emptyset, \{pc!\} : \emptyset^{b} : \text{Crossing}')$$

Which is of the form:

$$\text{Crossing}' = \emptyset^{a} : \{dn!\} : \emptyset^{b} : \{ic!\} : \emptyset^{g} : \{pc!\} : \emptyset^{m} : \{mu!\} : \emptyset^{n} : \text{Crossing}'$$

Where $a, b, g, m, n \in N \cup \{\infty\}$ □

One remark about this proof is that it depends heavily upon the trick of using integer variables as exponents of the actions which is not part of the theory, the alternative, expanding all the cases corresponding to all the possible combinations values of the various exponents, would defy the purpose of getting to a simpler expression.

6 Comparing the three approaches

Let us now look at those three approaches, first in terms of the characteristics of the languages, then in terms of the readability and maintainability of specifications. We will also compare their notion of time, synchronization and communication, their way of dealing with properties, and their proof systems.

6.1 Expressiveness, readability and maintainability

All three approaches did allow us to adequately specify the railroad crossing example without much difficulty. The resulting systems have a similar architecture, but they differ in the way they relate to their environment. The HMS defines a closed system without any formal way to specify interaction with an external environment, neither in a proactive nor a reactive sense. Modechart does define external events whereby the environment can influence the system; conversely actions can be used to specify the effect of the system on its environment. In CCSR, the effect of the environment can be specified via events, but there is no formal way for the system to impact its environment except by leaving un-hidden events for the environment to synchronize with.

The concept of system state seems well supported in HMS, but this is somewhat misleading because the history can have as much influence on the system behavior as the current state. In modechart, the notion is even more diffuse because the true system state includes not only the set of modes the system is in, but also the value of all the state variables and possibly the set of actions that are in progress. This notion is totally absent from CCSR. Interestingly enough, the decomposition of the railroad crossing system definition that we made to make the proof more readable resembles strangely a decomposition into states and transitions.

Modechart directly supports the notion of delay and deadline. These notions are also adequately supported in HMS via controls. In CCSR, however, Scope can be used to define deadlines, but it is very awkward to use, in the example, we defined the δ function which, strictly speaking, is an extension of the language, but simplified both the definition and the proof drastically. To define delays, we had to resort to the exponentiation of events, again an extension of the language.

HMS has a very sophisticated transition control mechanism that makes it possible to specify the behavior of the system based on a history of arbitrary length. This can be considered good or bad; it is good in the sense that it affords a lot of flexibility in the system definition, it is bad in that slight modifications of transition controls may have far reaching consequences in the future. Fortunately the graphical representation of the system should make such dependencies obvious.

Modechart is much more restrictive in that transition control is based only on the present state and on the amount time spent since last entering each one of the current modes. Controls based on more distant history must be implemented by setting state variables and additional modes; this can become fairly cumbersome.

Like modechart, CCSR does not offer language constructs to test past history. This can be accomplished by "firing off" separate processes to record events that will be used in the future. This mechanism is very powerful and allows the specification of behaviors based on unbounded history. For example, the expression fix(X.{a} : (X || $\delta_\infty(\emptyset$, b :

NIL))) implements an infinite counter that accepts {b}'s up to the number of {a}'s that it has seen.

While all three systems support the notion of non-determinism, none support any notion of probability which would be useful to study system efficiency and average behavior, for example.

6.2 Readability, abstraction and code reusability

The first thing that strikes one when looking at an HMS diagram is that it is very busy and confusing with lines of different thickness all over. The corresponding set of transitions is even more difficult to grasp. The modechart diagram, on the other hand is very clear and understandable. Of course, not everything is said in the diagram itself, variables and actions are defined separately, but this may be a good compromise after all. CCSR lacks a visual representation and the algebraic expression is not very intuitive. To be fair to CCSR, it must be mentioned that it is not meant to be a user's language. Rather one would define a system in a language called CSR with a more traditional high level, statement oriented syntax. CSR programs are then compiled into CCSR to allow the algebraic manipulations of the proof system.

The restricted version of HMS presented here does not support hierarchies and there is no base to support abstractions. A system specification is totally flat and transitions can address directly any individual states without any kind of structure. Modechart does support some level of abstraction; for example, one can refine a mode by structuring it with sub-modes either to disable outgoing transitions until a given action is complete, or to implement multiple concurrent actions. Such abstractions, however, are limited to a single input and single output transition. CCSR offers the best support for abstraction of the three languages. The definition of CCSR variables, combined with the hiding, or rather leaving un-hidden, events offers a way to independently specify sub-systems with a restricted set of externally accessible events. Unfortunately there is no distinction between input events, where the environment influences the behavior of the abstraction, and output events where the abstraction reacts upon the environment.

Code re-usability is often based on the abstraction mechanism. None of the systems reviewed offer good support for code re-usability. Both modechart and CCSR lack a renaming mechanism that would allow one to use multiple copies of the same abstraction in the specification of a higher level system.

6.3 Time, Synchronization and Communication

All three systems support a discrete notion of time, but while such a restriction is fundamental to CCSR (an event duration is a quantum of time), it is less fundamental to

HMS, where continuous time could be implemented at the expense of some additional complexity in the proof system. In modechart, the restriction is purely incidental continuous time could be easily implemented. Of all three systems, only HMS allows some level of time expressions. None support time variables.

Of all three systems, only CCSR support the notion of time-sharing or interleaving via the use of resources. The other two systems assume total parallelism.

All three systems offer mechanisms for synchronization. In HMS, synchronization between multiple transition is achieved by having mutual references on transition controls — e.g. transition γ_1 has a control $(\gamma_2, 0)$ and transition γ_2 has a control $(\gamma_1, 0)$. A very similar technique of mutual referencing can be used in modechart. CCSR, of course, directly supports synchronization via the concept of connection sets.

While full HMS offers communication via token passing, the restricted version for which a proof system has been developed does not; as most other finite state machines, it relies on the multiplication of states to store information. Modechart offers some limited amount of information transfer through the use of the boolean state variables. CCSR does not have the concept of information storing or passing; on the other hand, it is the only one of the three systems that is not limited by a fixed number of states at execution time. The infinite counter that we have seen earlier is a good example of that.

6.4 Specification of properties

Beyond the safety properties that we have been using so far, there are many other classes of properties that one might interested in when studying a real-time system. For example liveness, "something good will eventually happen", as opposed to safety which is "something bad will never happen", separation: maximum and minimum elapsed time between successive events, just to name just a few.

The richest system in the specification of properties is by far modechart. RTL allows the specification of all sorts of properties including liveness and separation.

In HMS, safety and maximum separation properties are expressed using state expressions and are implemented as un-reachability of unsafe states. This method can be extended to minimum separation fairly easily, but it not clear how one would go about specifying liveness properties.

Unfortunately, CCSR does not include any formalism to specify properties. As we have seen, one has to define a system which exhibits the required property and prove that the two systems are equivalent. This is not necessarily a simple matter and the risk of defining a reference system that does not have the required property is great.

6.5 The proof systems

The first remark, which is common to all three formalisms is that proving a property is even harder than specifying a system, and that the likelihood of developing an incorrect proof is at least as great as the likelihood of a bug in the specification. This implies that, without a good support tool, the usefulness of these methods is greatly reduced. In that respect we attach a great deal of importance to the implementability of the proof system. From that point of view, all three methodology are of exponential complexity, which is not surprising since they all solve the reachability problem.

As mentioned in [FG89], while the proof system of HMS is complete, i.e. all the true facts are provable, and while the proof of completeness is constructive, the number of steps it requires makes it impractical for automation. One must rely on intuition and a good choice of the rules to apply (out of a fairly large number of rules) to keep the proof tree manageable. This leans towards the development of interactive tools where the user drives the proof while the system insures the correctness, and of heuristics to help decide which rule should be applied.

The proof system of modechart has been implement by Douglas Stuart [Stu90], which clearly proves that it is implementable. The current version, although somewhat restrictive does allow to prove such properties as safety and maximum or minimum separation. It is interesting to note that the definition of distance equivalence used in the implementation is a greatly simplified version of the one described in [JS88]; this new condition is even less tight than the previous one, which could lead to bigger graphs, but it probably covers most of the practical cases.

The proof system for CCSR is proven to be complete only for finite behaviors, the completeness for infinite behaviors is still an open question. The implementation of a tool for proving properties of CCSR expressions is a subject of current research. It is interesting to note that CCSR roughly corresponds to a subset of CCS called "small CCS" in [Bol87], and it is likely that similarly, if recursion through parallelism (i.e. expressions similar to our infinite counter above) is avoided, a CCSR expression can be transformed into a computation graph similar to the CCSR one and hence similar proof algorithm can be utilized. In addition, using hiding and closure before implementing the graph would lead to a much simpler graph.

7 Conclusion

Although widely different in their approach, the three methods that we have studied did allow us to specify the common example without much difficulty, in fact, the underlying structure of the three solutions was fairly similar. The proof mechanisms for all three systems, on the other hand, was widely different except for the extreme difficulty of

developing a proof manually. This highlights the need for tools to automate the process.

We also compared the three approached and saw that they have widely different capabilities and potentials for the implementation of large systems, but a common weakness of all three approaches is the lack of support for abstraction and code re-usability; without which those systems will remain toys for laboratories.

Much research still needs to be done before systems, methods and tools to specify and prove properties of real-time distributed system are widely in use in the industry, but the progress so far are encouraging.

References

[AS85] Bowen Alpern and Fred Schneider. Defining Liveness. *Information Processing Letter*, 21:181–185, 1985.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[Bol87] Tommasso Bolognesi. Fundamental Results for the Verification of Observational Equivalence. In *Protocol Specification, Testing and Verification*, North-Holland, 1987.

[FG89] M.K. Franklin and A. Gabrielian. A Transformational Method for Verifying Safety Properties in Real-Time Systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 112–123, December 1989.

[GF88] A. Gabrielian and M.K. Franklin. State-Based Specification of Complex Real-Time Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1988.

[GL90a] R. Gerber and I. Lee. A Proof System for Communicating Shared Resource. In *Proc. 11th IEEE Real-Time Systems Symposium*, 1990.

[GL90b] R. Gerber and I. Lee. *CCSR: A Calculus for Communicating Shared Resources.* Technical Report MS-CIS-90-16, University of Pennsylvania, Department of Computer and Information Science, March 1990.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[JLM88] F. Jahanian, R.S. Lee, and A. Mok. Semantics of Modechart in Real Time Logic. In *Proc. 21st Hawaii Int. Conf. on System Sciences*, Jan. 88.

[JS88] F. Jahanian and D.A. Stuart. A Method for Verifying Properties of Modechart Specifications. In *Proc. IEEE Real-Time Systems Symposium*, pages 12–21, December 1988.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer, Berlin, 1985.

[Stu90] Douglas Stuart. Implementing a Verifyer for Real-Time Systems. In *Proc. 11th IEEE Real-Time Systems Symposium*, 1990.