Formal Methods for the Specification and Design of Real-Time Safety Critical Systems*

Jonathan S. Ostroff

April, 1992

Abstract

Safety critical computers increasingly affect nearly every aspect of our lives. Computers control the planes we fly on, monitor our health in hospitals and do our work in hazardous environments. Computers with software deficiencies that fail to meet stringent timing constraints have resulted in catastrophic failures. This paper surveys formal methods for specifying, designing and verifying real-time systems, so as to improve their safety and reliability.

*To appear in *Journal of Systems and Software*, Vol. 18, Number 1, pages 33–60, April 1992. Jonathan Ostroff is with the Department of Computer Science, York University 4700 Keele Street, North York, Ontario, Canada, M3J 1P3. This work is supported by the Natural Sciences and Engineering Research Council of Canada.

Contents

1 Introduction

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

Safety critical systems must satisfy real-time constraints if they are to effectively perform their intended function. The newsletter *Software Engineering Notes* regularly reports incidents involving malfunctioning of real-time or embedded computer systems. For example, the first flight of the space shuttle was delayed by a subtle timing error, which was traced to an improbable race condition in the flight control software [31]. In another incident, a software error caused a stationary robot to move suddenly with impressive speed, to the edge of its operational area. A nearby worker was crushed to death. The firing mechanism of an already deployed ballistic missile system was recently analyzed using methods discussed in this survey. It was discovered that a certain sequence of events, unknown to the design team, would lead to the inadvertent firing of a missile [40].

Real-time software must satisfy not only functional correctness requirements, but also timeliness requirements. For example, consider the following "hard" real-time constraints: if the temperature of a nuclear reactor core is too high an alarm must be generated within some deadline; spray painting a car on a moving conveyor must be initiated at some suitable time and terminated some time later; when an aircraft enters an air traffic control region, the flight controller must be informed in a timely fashion; once the approach of a train is detected, car and pedestrian traffic at the train intersection must be halted before the train reaches the intersection; if the computer controlling a robot does not command it to stop or turn in time, the robot might collide with another object on the factory floor.

There is general consensus in the software and control systems literature that real-time systems are difficult to model, specify and design [142,42,17, 121,74,137,73]. In addition, experience has shown that software components of systems are problematic perhaps even more so than mechanical or other hardware components.

Software is complex (consider the documentation needed for even simple modules), non-robust (small errors have major consequences) and software is notoriously difficult to test (the number of test cases that must be checked becomes unmanageably large even in small systems) [115].

This does not necessarily mean that the software controlling real-time systems is poorly designed. Many companies do as much as is commercially feasible with current design methods to make their products more reliable. However, as the burden of controlling complicated systems is shifted onto the computer, so does the complexity of the resulting software increase. Old-fashioned servo-control systems could be tested in isolation. The new more complex software controllers are more difficult to check exhaustively, no matter how intelligently designed the test suite is.

It has been conjectured that formal, mathematically precise methods should be used to design real-time safety-critical systems. Turning this conjecture into sound practice has proved to be extremely difficult — many practically-oriented software engineers will probably consider the conjecture to face insurmountable hurdles.

But what benefits do "theorists" hope to obtain by the use of a formal framework? A list of the benefits includes:

- In the process of formalizing informal requirements, ambiguities, omissions and contradictions will often be discovered.
- The formal model may lead to hierarchical semi-automated (or even automated) system development methods.
- The formal model can be verified for correctness by mathematical methods (rather than by intractable case by case testing).
- A formally verified subsystem can be incorporated into a larger system with greater confidence that it behaves as specified.
- Different designs can be evaluated and compared.

Some researchers think that a specific real-time verification methodology is not needed. Conventional wisdom dictates that programs should be designed to function correctly, independent of hardware speed. One extreme position [138] views introducing time with grave suspicion. Time may be an issue in implementation ("use a faster machine if you miss the deadline") but should never appear in a specification. Furthermore, there is a danger in overspecifying and making the verification task more complex than it ought to be. In contrast, de Roever [22] considers it essential that foundational research be undertaken into formal methods for real time systems if reliable and safe systems are to be constructed.

If the designer is dealing with fixed schedules on a single processor, then it may be possible to get away with using untimed standard verification methods. Certainly one should abstract out time and use standard techniques wherever it is possible to do so. However, the work discussed in this survey clearly shows that tampering with the speed of the computer will not solve the main problems facing the designers of real-time software.

Time is not just another programming variable. Time is continuous, monotonic and divergent whereas program variables generally do not have such properties. Since time ranges over an infinite domain, all the tools that have been developed for finite state verification cannot be naively applied to real-time systems. With some effort, finite state methods can be made to work for real-time systems, but to do this certainly requires the adoption of specific formalisms to deal with explicit timing constraints. Proof systems for dealing with untimed infinite state systems must be refined if they are to deal safely with timed systems.

If time dependencies are introduced into a design, then there should be good reasons to do so. Mok [92] cites the following cases in which timing constraints play an important role.

- The control surfaces of some modern aircraft must be adjusted at a high rate to prevent catastrophic destruction. This places an upper bound on the response time of the avionics software system. Lower bounds are needed in operating systems which require a potential intruder to wait for some minimum time before retyping a password that has been entered incorrectly. *In these cases, the "physics" of the application dictates the timing requirements.*

- In the Byzantine Generals Problem, the non-faulty processors (generals) must arrive at a consensus to perform some action in the presence of other processors that can exhibit faulty behaviour. There is no asynchronous solution to the Byzantine Generals Problem. However, a solution is possible if the generals adopt the synchronous protocol of voting in rounds. In each round of voting the generals must complete a set of communication actions within a real-time deadline. *Thus, time is an essential synchronization mechanism for solving certain task coordination problems.*

- In the NETBLT protocol proposed by a group at MIT, the receiver guarantees the sender that it will be able to process incoming packets at a certain rate, or alternatively, it will meet the deadline associated with each packet. Since the sender does not need to wait for an acknowledgment from the receiver, network throughput can be significantly improved, especially for networks where the round-trip transmission time is long compared with the width of a packet (e.g. in fiber optics communication systems). *Thus, time is a control mechanism which can be exploited to solve problems more efficiently.*

In the rest of this survey, we define more carefully what a real-time system is (Section 2). Three increasingly formal techniques are surveyed for dealing with real-time systems.

- Section 3 discusses real-time programming languages.
- Section 4 discusses structured analysis methods and graphical or visual modelling languages.
- Section 5 discusses logics and algebras.

Section 6 speculates on future trends.

2 Defining the terms

This section describes the important features of real-time systems. The following problems are also posed: the modelling problem, the verification problem, the design development problem and the controller synthesis problem. These are some of the main problems that theorists hope to tackle with formal methods.

An algorithm is usually represented as a *program*. Correctness of the algorithm means that the program *satisfies* some desired *specification* (or property). Therefore, in standard program verification, three concepts are needed. A programming language is needed for representing algorithms, a specification language for expressing properties, and a satisfaction relation (or proof system) for verifying the correctness of the algorithm. An important subjective issue revolves around the choice of *syntax* for the programming and specification languages — does the syntax simplify the expression of algorithms and specifications, or does it get in the way of the designer.

To properly develop the concepts mentioned above, some additional notions must be introduced. A formal *semantics* must be provided so that the behaviour of the program and the meaning of the specifications are clearly defined. The proof system must be shown to be *sound* with respect to the semantics, so that only those programs whose behaviours satisfy the specification can be proven correct. An unsound proof system is dangerous because it can be used to prove anything (papers on the assignment axiom for arrays were, for years, filled with errors). Furthermore, it is useful if the proof system is *complete* so that every correct program can be verified for correctness in the proof system. Other issues such as the expressiveness of the languages and the complexity of decision procedures must be explored.

An algorithm usually takes some data as input, performs a computation and outputs the result of the computation to the user. For real-time programs, the situation is more complex. The environment in which the program operates can no longer be ignored, because of the intensive nonterminating interaction of the program with its environment. Such systems are often called embedded systems, discrete event dynamic systems, reactive systems, or process control systems. We shall refer to all of these as "real-time systems".

Ordinary programming languages are not expressive enough to represent the complex features of real-time systems such as concurrency, nondeterminism, synchronization between processes and real-time constraints on the events of such systems. Programming languages must therefore be extended if they are to deal with real-time systems. A *model* of the intended real-time notions is therefore needed.

A *model* is a representation, often in mathematical terms, of the important features of the system that is being studied. A common modelling technique is to define the *state* of the system (a "snapshot" at an instant in time of all the variables defining the system). A state may persist for some period of time, after which there is some change to a new state (as real-time systems are dynamic). Such a state change is referred to as an *event* or a *transition*. The model may be used to *simulate* possible behaviours of the system which helps the designer understand the system better. A simulation of the system is a sequence of states and events capturing the behaviour of the system. A simulation may show the presence of bugs in the system, but never their absence [23]. *Analysis* of the system behaviour must be undertaken to show the correctness of the system. The system is correct provided that its behaviour satisfies the associated specification.

The terms "model" and "specification" are often used interchangeably. In this survey, a *model* is a *description* of the system, perhaps in great detail, or perhaps at a more abstract level. A *specification* is the list of *requirements* that the system must ideally satisfy. A model describes how the system actually behaves. A specification prescribes how we would like it to behave.

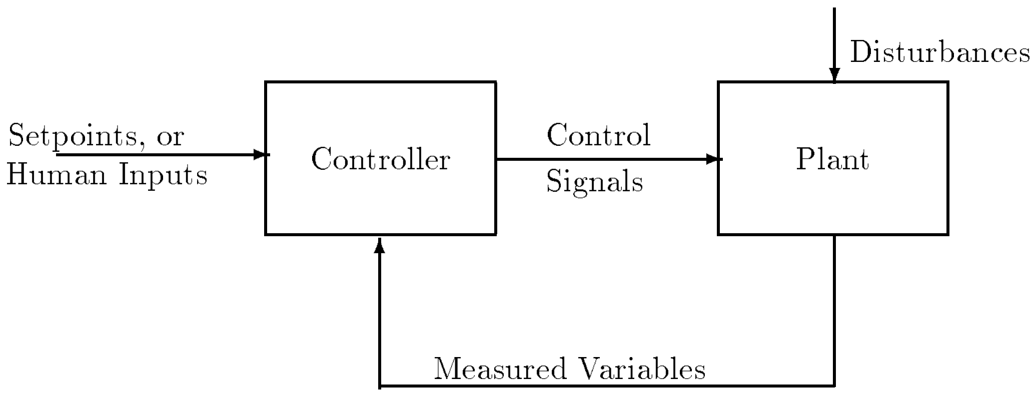
What are the most important features of a real-time system? The Oxford Dictionary of Computing defines a real-time system as follows:

> [A real-time system is] any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

The IEEE Standard Dictionary of Electrical and Electronic Terms (Wiley 1978) gives the following definition:

> *Real-Time:* (A) Pertaining to the actual time during which a physical process transpires. (B) Pertaining to the performance of a computation during the actual time that the related physical process transpires in order that results of the computation can be used in guiding the physical process.

In software engineering, the term "real-time system" usually refers to the software or programming code (called the *controller* in this survey). The lag time from input to output in the controller must be sufficiently small. However, there is implicitly always another object that is associated with the controller. That object is the physical world or environment in which the controller finds itself. Since the environment is always implicitly there, let us give it a name and call it the *plant*. Then we can refer to the plant and perhaps even reason about it. In fact, the lag time or response time of the controller is determined by the physical nature of the processes in the plant. The primary goal of the designer is to ensure the correct behaviour of the plant. This is achieved by designing a controller that will interact and control the plant. We may draw an automatic control diagram that is familiar to control engineers:



The complete system under development (SUD) is divided into two parts: the controller and the plant. The plant is that part of the system that is to be controlled. It is often a physical or technological process such as a chemical reaction, airplane or robot. The plant is usually a "given"; the designer is not free to change it, although there is usually some kind of "control technology" through which the plant can be controlled (e.g. control valves can be opened or certain plant events can be forced to occur through interlocks). The "open-loop" behaviour of the plant (without the controller) is usually unsatisfactory in some important respect.

It is the task of the controller to ensure that unsatisfactory behaviour in the plant is eliminated. The diagram above indicates the role that feedback plays in the controller. Feedback restores equilibrium after disruptions caused by disturbances to the plant. After measuring the current state of the plant the controller can take corrective action by issuing appropriate control commands.

There is usually much more freedom in the design of the controller than in that of the plant. In fact, the controller is often implemented by real-time software precisely so that the logic of the controller, or the computation that it performs, can easily be changed if necessary. As a result, the design of a real-time system differs from straight programming (e.g. coding an algorithm) in two important respects:

1. *The plant is part of the overall system.* A real-time system is one in which the *controller* software must synchronize with the *plant* processes whose progress it cannot directly control, so as to ensure that the plant behaves safely and reliably.

   The design formalism must be flexible enough to represent the plant as an integral part of the complete system. Unsatisfactory plant behaviour can then be examined, and the effect of different control policies on the plant can be evaluated. Potential failures of the plant must be represented in the plant model so that the controller can be checked to see that it functions correctly. A model of the plant will enable the designer to extract the lag and real-time response times that the controller must implement. Furthermore, a deterministic formalism developed for a real-time programming language (to guarantee predictable behaviour), may be unsuitable for representing nondeterministic and asynchronous event driven plant behavior.

2. *The plant must be verified for correctness.* The essential concern of the designer is to ensure that the plant behaves in a safe, acceptable fashion. The correctness of the control software is only a means for ensuring correct plant behaviour. If the software of the controller fails (relative to its local specification) in a fashion that has no impact on the plant, then no harm has been done. By contrast, the controller can satisfy all kinds of requirement specifications, but if those requirements do not translate into proper behaviour of the plant, then the verification effort will have been in vain. An important corollary is that a specification of the system must primarily refer to the states, events and properties of the plant (not to the behaviour of the controller software).

Consider the following robot example[1]. The system under development

[1] The robot example was used in a discussion on real-time issues on Usenet in 1990 with

(SUD) is a robot arm together with position sensors, a force sensor, a camera, a tactile sensor, a gripper, a joystick and keyboard for input, and a computer to control the various parts. Formally, we may write

$$SUD = plant \| controller$$

where $\|$ indicates the parallel composition of plant and controller processes. The plant is further defined by

$$plant = arm \| sensors \| actuators$$

where

$$sensors = positionSensors \| forceSensor \| tactileSensor \| Camera$$

and

$$actuators = gripper \| joystick \| keyboard$$

The model of the plant will have to represent certain timing constraints. The camera works at 30 Hz, and the force sensor at 400Hz. The robot and position sensors work at 1000Hz, the joystick at 25Hz, and the tactile sensor at 120Hz.

The required specifications are: design a controller that will enable a user to manipulate the arm to perform various tasks at certain speeds, e.g. welding a part to an auto on the assembly line. The sensor measurements must be scanned at the correct rate. The actuators must be activated at the correct time. If an unfamiliar obstacle is encountered stop the arm movement within one second.

The specification S of required behaviour so far only refers to elements of the plant such as the arm, the sensors and actuators. The controller must still be designed, and so its constituent parts cannot yet be referred to in the top level specification.

Having specified as precisely as possible the plant, and the specification S that it must satisfy, the controller must now be designed. It was necessary to structure the description of the plant with the parallel composition operator. So too, it is often also necessary to structure the controller description.

---

contributors Duncan Thomson and David Stewart

For example, it is possible that only some of the sensors will be used at any one time. A logical way to separate the control functions is by having one task supervise the control signals to the robot and another task read the camera. Other distinct tasks can read the position and force sensors. If only the force sensor is needed for a particular task, and not the camera, then it is simply a matter of starting up the appropriate tasks without the need to change any code.

Without concurrency the software must be constructed as a single control loop called a cyclic executive. Such a large sequential program with multiple conditional flags such as "if (using camera) do action" is difficult to design robustly because of the different time frames and functions that must be accommodated. The structure of this loop cannot retain the logical distinction between controller modules. It is difficult to ensure that the executive synchronizes in a timely fashion with the plant processes, without the explicit notion of concurrent tasks in the controller software.

In a concurrent controller, each time frame is handled by a different task. The frequency of each task can be specified separately, and the burden of deciding what to run when is placed on the operating system scheduler. Even if the timing changes, there is no need to reprogram the controller. The real-time operating system is designed so as to be capable of adapting to the new requirements. If a large sequential program is used, the entire program would have to be restructured to satisfy the new timing requirements.

The use of concurrency is not without cost. There must be a run time support system to manage execution of controller tasks or processes. Such run time schedulers are often not considered in formal verification methods, but should be for a complete treatment of all system issues. Scheduling is treated as one discipline and verification as a different one. Methods that treat, in a unified framework, all aspects of verification and scheduling have not been sufficiently developed.

There is one part of the design process that is informal and intuitive. When the real world (e.g. of valves, pumps, vehicles, and robot arms) is translated into a formal mathematical model *plant* (e.g. of states, events and time bounds), there is no guarantee that the mathematical model properly represents the actual objects to be controlled. Similarly, there is no guarantee that the model *controller* is an accurate representation of all important facets of the actually implemented software (together with the hardware, CPU and run time system). This translation of real world objects into mathematical entities is by its very nature informal and intuitive.

As more experience is gained with a particular formalism, and actual designs based on the formal methods are experimentally checked in the field, so the formalism will gain more credibility. This is no different from the methods used in related disciplines. For example, a civil engineer will model the real world of bridges, beams and winds using the formal techniques of Newtonian physics. As more bridges are built and actually succeed in practice, so the Newtonian models gain credibility.

In certain cases, the pure Newtonian model must be adjusted with "safety factors" to account for the approximate nature of the models used. It is not clear what the software equivalent of safety factors is.

The fact that one part of the design procedure remains informal and experimental does not in any way detract from the need to use formal design procedures in the rest of the design. Just as the civil engineer uses formal Newtonian models for bridge building, and thereby increases confidence in the design, so too the designer of software uses formal methods to increase confidence in the correctness of the real-time system design. A proof of correctness is always relative to the formal models and specifications provided.

2.1 Major issues that formal theories must address

Formal methods for real-time systems must address the following problems:

Modelling: Select appropriate models and formal notations for adequately describing plants and controllers. These notations must deal with the dynamic and reactive nature of the plants, and allow for the proper expression of timing properties.

Verification: The verifier is presented with a formal mathematical model SUD where $SUD = plant \| controller$, and a specification S of how the plant should behave. The *verification problem* involves demonstrating that SUD satisfies the specification S.

Development: In controller *development* a specification S is given that the *plant* must satisfy (the *controller* is not given). A disciplined method is sought whereby designers can be helped to construct a *controller* so that SUD satisfies S. In development the controller should be built in a modularly structured compositional fashion ("controller architecture").

Synthesis: If controller development is fully automated, then it is called controller *synthesis*.

There are a few surveys in the literature of formal methods for real-time systems design [60,22,133]. This survey uses syntactic categories to classify the various formalisms. There are three main directions that must be distinguished, arranged by increasing formality and hence abstractness of approach. The first category to be dealt with is real-time languages. Then formalisms with visual specification languages are treated. Finally logics and algebras are discussed.

3 Real-Time Programming Languages

Modern real-time languages such as Ada [139], Chill [18], Occam [76] and Conic [65,79] typically have delay and timeout features for implementing timing constraints. In addition, these languages incorporate features such as task decomposition, abstraction (information hiding), communication and concurrency mechanisms that simplify the description of complex controller software.

For example, in Ada, a module can be decomposed into several tasks. A task can be further divided into two parts: the task specification and the body of the task. During initial program development, only the specification part of the tasks needs to be defined. The specification parts can be compiled to check for overall controller consistency. Later, the task bodies can be specified in more detail. The specification part is done by senior designers, because the specification represents the interface between software components. An error in such code may therefore have more serious ramifications than an error within a task body. A task can then be given to a junior programmer for coding.

Conic implements several useful reconfiguration primitives. The primitives allow for dynamic insertion or deletion of tasks or modules, while the rest of the system continues to run. Thus maintenance to controller software can be implemented without shutting down the plant. Occam has been used as a target language for process algebras (see Section 5.2).

The advantage of these languages is that at the end of the design process, the controller is available in directly executable code. However, most real-time programming languages lack an underlying abstract mathematical model. As a result the precise semantics is unspecified or even uncertain, and there is a proliferation of irrelevant detail not needed at the level of abstract specification. Usually the code must be converted into a formal notation (e.g. Petri Nets) before the code can be verified. If these languages alone are used, then the plant is not usually represented, although pseudo plant processes can be coded. Thus there is no method to formally verify whether the controller satisfies the requirements specification S.

4 Structured Methods and/or Graphical Languages

4.1 Structured Methods

Structured methods for real-time systems [44,141] originated in systems analysis methods used in industry starting a decade ago. These methods provide a structured set of system (read: controller) requirements. The system requirements specify what problem the controller must solve, and how the controller must be structured.

The system requirements includes various views and layers of the controller such as (a) data flow diagrams to decompose the controller, its functions and its flow of data, (b) control flow diagrams (enhanced state transition diagrams) to represent the system dynamics, (c) a requirements dictionary which is an alphabetical listing of all inputs, outputs, data and control flows, and (d) a table of response times in which the incoming and outgoing events (from and to the plant) are listed with their respective repetition rates or response times. The timing requirements are not particularly well integrated with the rest of the requirements.

These methods have been used with some success in actual industrial applications. However, these methods have no formal semantics. The resulting controller design cannot be executed for simulated behaviour, nor can they be compiled into code (e.g. into Ada or C). There is no support for formal verification. Nondeterministic plant behaviour cannot be suitably modelled.

4.2 Graphical Languages with a formally defined semantics

4.2.1 Statecharts and synchronous languages:

Statecharts [38] represents an improved version of the structured methods. A graphic tool called "Statemate" [41] exists to implement the formalism. Methods similar to that of Statecharts may be found in [29].

In Statecharts, the normal state transition diagram is enhanced with hierarchical and compositional features. For example, states can be clustered into super-states with the possibility of "zooming in" and "zooming out" of states. In AND decomposition, states are split into orthogonal (concurrent) subcomponents that communicate via broadcasting. OR decomposition decomposes a state into sub-states such that control resides in exactly one sub-state.

Statemate is formally based on a precisely defined (although rather complex) semantics. As an important consequence, there is an automated simulation tool which allows the user to execute the model. Exhaustive checking of all possible behaviours (for small systems) is supported. Models (e.g. of the controller) can be compiled into a target environment, although this tool is not yet fully developed. Formal verification is not yet supported, nor are time constraints treated in sufficient detail.

According to [86], the real-time aspects of Statecharts semantics is underdeveloped. A notation is needed for periodic timing functions, and for the specification of timing exceptions without the need to introduce additional states.

Recently, some progress has been made in providing formal analysis techniques for Statecharts. A fully abstract compositional semantics of Statecharts has been presented [53].

The simplest kind of semantics uses the operational notion of the set of all observable behaviours of a program or system. A behaviour may be defined as an infinite sequence of states that the system computes in one run. Such an operational semantics cannot usually be used to deliver a compositional (modular) semantics. In a compositional semantics, a compound system $M_1 \| M_2$ is described in terms of the semantics of the component systems M_1 and M_2. A fully abstract semantics defines the behaviour of the system in sufficient detail without violating the principle of compositionality, i.e. such a semantics does not distinguish between more systems than is necessary to provide a compositional semantics.

The importance of compositionality is that structured program verification can be undertaken. Components of the system can be checked independently, and then combined to provide a verifiably correct compound program.

In addition to the abstract semantics of Statecharts presented in [53], there is an alternative version of Statecraft semantics. In [119], a semantics is presented that uses micro and macro-steps. Observable macro-steps are decomposed into a number of micro-steps. If event A triggers B then these events occur in subsequent micro-steps within the same macro-step. Thus a chain of causality inside one macro-step is modelled by a sequence of micro-steps. This avoids some of the causal paradoxes that occur in synchronous languages such as Statecharts.

The "synchrony hypothesis" assumes that a program instantly reacts to external events. In practice this means that the reaction time of the controller is always shorter than the minimum delay separating two successive external events in the plant. In addition to Statecharts, the languages Esterel [11], Signal [9] and Lustre [16] have been designed with the synchrony hypothesis in mind. The synchrony hypothesis must be applied with care as it can lead to causal paradoxes such as events disabling their own cause. This hypothesis is not always realistic, as the plant usually involves spontaneous behaviour that may occur at any moment of time. Languages such as Esterel do not allow for nondeterminism which is also not realistic for representing plants.

4.2.2 Petri Nets

Petri Net theory [116,129] was one of the first formalisms to deal with concurrency, nondeterminism and causal connections between events. According to [91] it was the first unified theory, with levels of abstraction, in which to describe and analyze all aspects of the computer in the context of its environment (computer + program + aircraft). Such a method must perforce contain a theory of concurrency, because of the three ingredients, at most one — the program — is sequential. Previously the various components of the total system had to be described in diverse and unrelated ways. The computer hardware would be described by low level automata theories, the program by code in a sequential programming language, and the interaction of the program with its environment by narrative prose.

The classic Petri Net model is a 5-tuple (P, T, I, O, M). P is a finite set of places (often drawn as circles) representing conditions. T is a finite set of transitions (often drawn as bars) representing events. I and O are sets of input and output functions mapping transitions to bags of places (the incidence functions). M is the set of initial markings.

Places may contain zero or more tokens (often drawn as black circles). A marking (or state) of the Petri Nets is the distribution of tokens at a moment in time (i.e. $M\colon P \rightarrow N$ where N is the nonnegative integers). Tokens in Petri Nets model dynamic behaviour of systems. Markings change during execution of the Petri Nets as the tokens "travel" through the net (modelling flow of materials for example).

The execution of the Petri Nets is controlled by the number and distribution of the tokens (the "state"). A transition is enabled if each of its input places contains at least as many tokens as there exists arcs from that place to the transition. When a transition is enabled it may fire. When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places.

Given an initial state (distribution of tokens), the reachability set is the set of all states that result from executing the Petri Net. Properties such as boundedness, liveness, safety and freedom from deadlock can be checked by analyzing the reachability graph. The reachability graph is usually constructed using an interleaving operational semantics.

Boundedness means that the number of tokens which any place in the net can accumulate is bounded. Boundedness implies that the system is finite state. In a Petri Net, a transition is said to be *live* if it is potentially fireable in all reachable markings. Liveness in the program verification sense is a different concept meaning that the transition must eventually occur.

In Petri Nets causal dependencies and independencies in some set of events are explicitly represented. It is therefore easy to provide a nonrestrictive partial order semantics. Events which are independent of each other are not projected onto a linear timescale. Instead a non-interleaving partial order relation of concurrency is introduced. In an interleaved execution, one cannot differentiate whether two events occur one after the other because the first is a prerequisite of the second, or whether this order in time is solely by chance [129].

There are also some structural analysis techniques that use linear algebra to check for invariants. Every marking can be represented as a vector, and the Net can therefore be modelled as a set of linear algebraic equations. An example of an invariant is a place-invariant, in which every place in the Net is assigned a weight, so that the weighted token count remains constant during the execution of the Net. The dual of the place-invariant is the transition-invariant. Since invariants are the characteristic solutions of the net equations, it is possible to compute them by well known techniques in linear algebra. These invariants are useful in the analysis of net liveness properties or of facts (propositional formulas that are true in all cases).

For large nets it is hard to compute the invariants. Usually, there are infinitely many invariants (a linear combination of invariants is also an invariant). Therefore it is often difficult to obtain the interesting ones. If the user supplies the invariant, it is somewhat easier to check automatically that the invariant holds true. If it does not hold, it is relatively easy to see where the net or the invariant must be modified.

Generic "health" checks such as the absence of deadlock can be performed. Such checks are not always useful as some kinds of deadlock may be allowed (e.g. program termination).

Controllers are specified by augmenting the plant Net with additional places and transitions to model the required controlled behaviour. For example, let the plant be a Petri Net *elevator*, and suppose it is required that the elevator be allowed to be disabled for possible maintenance work. A new place must be added to *elevator*, representing the emergency stop request. A new transition, representing the stop event, is also inserted into the net. The stop event is connected to the place in *elevator* that controls the elevator movement. Short of introducing a specification logic, there is no way (other than by modifying the details of the Net) to specify the abstract requirement that "if the stop event is requested then the elevator must eventually come to a halt and be disabled".

Ordinary Petri Nets have been criticized for not being able to deal with fairness and data structures (e.g. the data in a measure header) [62,94], although the number of tokens at a particular place in the net can simulate a local program variable. Structuring mechanisms such as composition operators are not inherently part of the theory, and there is no calculus to transform a net into a real-time programming language. Unlike state machines, a "place" in a Petri Net cannot easily be identified with a place in the corresponding program code. A further problem is that the reachability graph suffers from state explosion as Petri Nets become larger, thus impacting on the ability to scale up analysis to larger systems.

Ordinary Petri Nets are still an object of intense research aimed at putting Petri Net theory on firm mathematical grounds. However, practically speaking such standard nets are not up to the task of modelling complex systems. For this reason, higher level nets (coloured nets) and stochastic nets have been introduced to extend the modelling power of Petri Nets [13].

In coloured nets, the main idea is that the tokens themselves may have values (or colours). Coloured Nets allow for a more concise manageable representation of systems. They can be used to model data and resources that reside in the system. As long as the number of colours is finite, a Coloured Net is equivalent to a (much larger) ordinary Petri Net. An infinite number of colours allow for the expressive power of Turing machines so that most general questions become undecidable.

Petri Net theory was one of the first concurrent formalisms to deal with real-time. Two basic timed versions of Petri Nets have been introduced: time Petri Nets [88] and time*d* Petri Nets [124]. Both have been used in recent work [145,125,87,72,12,27]. There are two questions that arise when time is introduced to net theory: (a) the location of the time delays (at places or at transitions), and (b) the type of the delay (fixed delays, intervals or stochastic delays) [140].

Timed Petri Nets are derived from classical Petri Nets by associating a firing finite duration (a delay) with each transition of the net. The transition is disabled from occurring for the delay period, but is fired immediately after becoming enabled. These nets are used mainly in performance evaluation.

Time Petri Nets (TPNs) are more general than timed Petri Nets. A timed Petri net can be simulated by a TPN, but not vice versa. Both a lower and an upper bound are associated with each transition in a TPN. A state in the reachability graph is a tuple consisting of a marking, and a vector of possible firing intervals of enabled transitions in that marking. Since transitions may fire at any time in the allowed interval, states in the reachability graphs may have an unbounded number of successors (if continuous time is used). This adds complexity to the next-state function. Various techniques such as state classes and enumerative analysis techniques [12] have been introduced to overcome this problem. There are no necessary and sufficient conditions for boundedness (finiteness of the reachability graph); however, some sufficient conditions have been provided.

Perhaps the most general Petri Nets available for real-time problems are the ITCPN (interval timed colour Petri Net) models of [140]. These nets are higher level nets, and an interval is used to specify the timing characteristics, by attaching a timestamp to every token. The resulting semantics is quite elegant because both the timing and the colour are attributes of tokens. A software tool called *ExSpect* is available for performing analyses on ITCPNs.

In [72], safety properties of TPNs are analyzed without the need to necessarily generate the entire reachability graph. The idea is to work backwards from high-risk states to determine if these hazardous states are reachable. This backward method uses the inverse Petri net (reversed input and output functions), and is practical only when a small number of unsafe states is considered. The idea is to work backwards from unsafe states to all critical states (i.e. states having at least two successors). When a critical state is reached, interlocks can be used to force the system to take those paths that do not lead to unsafe states.

A similar backward method is employed in the real-time temporal logic approach to controller design in [103,107] (see Section 5.1.1). Instead of dealing with unsafe states one at a time, a predicate (characterizing a possibly infinite set of unsafe states) is used. Weakest preconditions are used to work backwards to critical predicates. The advantage of using predicates (rather than individual states) is that much larger systems can be treated. The method has been semi-automated. The backward method is an example in which the same basic idea is applicable in two very different computational models.

5 Logics and Algebras

Logics and algebras provide the most abstract approach to the analysis of real-time systems. These approaches typically consist of several elements.

One element is a high level, formal specification language in which the requirements that the system must satisfy can be specified. A second element is a proof system (or finite state decision procedures) in which the correctness of the system relative to the specification can be verified. Another important element (that is not always provided) is a set of heuristics and general guidelines for using the approach on systems larger than the standard "textbook" examples.

Some researchers have investigated the general properties that any specification formalism must satisfy [55]. This is in contrast to the development of specific frameworks for doing verification. In [114], the relations that must hold between plant (environmental) variables and controller (software) variables are specified. These relations are used to express under what conditions a design is feasible, and what formal functional or relational properties a software requirements document must satisfy.

The current rigorous approaches to real-time systems are:

- Real-Time Temporal Logics,
- Algebraic Approaches (process algebras),
- RTL (Real-Time Logic) and Event/Action Models, and
- Assertional calculi

In this survey, a syntactic classification has been used based on the style of specification. The same underlying semantic model may often be applied to different syntactic specification styles. For example, both process algebras and temporal logics may be endowed with a discrete time semantics. Alternatively, in each case, a dense time semantics may be chosen. The classification of formal approaches could also have been performed on the basis of semantic models rather than syntactic specification styles.

There are a variety of timed transition systems (e.g. the timed automata discussed in [77]) that are being promoted as general models of real-time systems. These timed transition systems are not discussed in this survey as an independent group, but are treated individually where they relate to a particular specification language.

Some of the specification approaches are complementary. For example, temporal logic is good at describing properties that pertain to the complete system such as safety, liveness, fairness, and real-time response. However, temporal logic specifications are relatively unstructured, and could benefit from the more structured notions of processes algebras [22]. Process algebras are not good at specifying inherently global properties such as fairness which involve the complete computation. There is currently no theory that combines the best features of each formalism.

The following areas of research are listed below for the sake of completeness. They do not at present deal with real-time issues. However, they may be extended in the future to deal with real-time constraints.

- VDM [59] and Z [136] are specification languages based on set theory and predicate logic. These methods have been found useful for specifying large commercial systems [36], but are weak in their ability to deal with concurrency and real-time. They use concepts from classical programming logics such as Hoare's triples [49], and Dijkstra's weakest precondition calculus [23,35]. In [113], the first extension of Hoare triples to concurrent systems was provided.

- Control theoretic approaches based on algebraic methods and formal language theory have been developed for complex discrete event dynamic systems [123,122,54,15,75]. These methods are good at modular synthesis of controllers. They have on the whole not yet been extended to deal with real-time constraints, nor do they deal with data structures. Also noteworthy are the synthesis methods proposed in [84,24], and the automata based methods of [80,1].

- UNITY [19] is a specification and verification framework that uses extended state-transition systems to model plants and controllers (abstract programs). A logic, which is similar to temporal logic (but simpler), is used for specification and verification. A UNITY program describes *what* should be done. A *mapping* to a particular architecture is concerned with implementation details such as the type of machine that should be used. For example, the same abstract program can be mapped to a set of concurrent processors, to a single CPU using multiprogramming, or to a systolic array. This leads to a separation of concerns. Correctness is concerned with verifying the abstract program. Complexity is computed from the mapping to an implementation.

5.1 Real-Time Temporal Logic

The following discussion is taken from [104]. Temporal logic has its origins in philosophy, where it was used to analyze the structure or topology of time [130]. In recent years, it has found application in computer science, especially in the areas of software verification and knowledge-based systems [30,117,70,112,69,67,28,66].

In physics and mathematics, time has traditionally been represented as just another variable. First order predicate calculus is used to reason about expressions containing the time variable, and there is thus apparently no need for a special *temporal* logic.

Philosophers found it useful to introduce special temporal operators, such as $\Box$ (henceforth) and $\Diamond$ (eventually), for the analysis of temporal connectives in language. The new formalism was soon seen as a potentially valuable tool for analyzing the topology of time. For example, various types of semantics can be given to the temporal operators depending on whether time is linear, parallel or branching. Another question that may be asked is whether time is discrete or continuous.

The temporal operators have been found useful for specifying program behaviour. A *structure* of states (e.g. a sequence or tree of states) is the key concept that makes temporal logic suitable for program specification. A formula, containing temporal logic operators, is interpreted over a structure of states.

In programming languages, the structures represent the computations executed by a program. Such a computation may be used to interpret a temporal formula. In this way, a programming language is said to be endowed with a temporal semantics.

Some of the different types of temporal semantics include:

- *Interval semantics* [134,95,96]. The semantics is based on intervals of time, thought of as representing finite chunks of system behaviour. An interval may be divided into two contiguous subintervals, thus leading to the *chop* operator.

- *Point semantics*, in which temporal formulas are interpreted as requiring some system behaviour with respect to a certain reference point in time. *Past* operators refer to the time prior to the reference point. *Future* operators refer to the time after the reference point.

Obviously, a point cannot be divided, and there is thus no simple definition of a *chop* operator.

Point semantics may be further divided[2] into three classes.

– *Linear semantics* [83,120,81]. In linear semantics, each moment has only one possible future corresponding to the history of the development of the system.

– *Branching semantics* [25,20]. In branching time semantics, time has a tree-like nature in which, at each instant, time may split into alternative courses representing different choices made by a system.

– *Partial order semantics*. Partial order semantics has been explored only recently. The reader is referred to [81] and other articles in the same volume as [81] for further information.

Once the type of *structure* to be used for interpreting temporal formulas is selected, there is still a further decision to be made. How are the structures to represent program executions or computations? There are at least two possibilities.

- *Maximal parallelism* [52,62]. The number of instructions in concurrent processes that can be executed simultaneously is maximized. Thus, two processes are never both waiting to achieve a shared communication.

- *Interleaved executions*. Concurrent activity of two parallel processes is represented by interleaving their atomic actions [82]. Fairness and time bound constraints are then used to exclude inappropriate interleavings (e.g. a sequence in which an enabled action is never executed).

The various temporal logics can be used to reason about *qualitative* temporal properties. Safety properties that can be specified include mutual exclusion and absence of deadlock. Liveness properties include termination and responsiveness. Fairness properties include scheduling a given

[2]Ron Koymans has pointed out [61] that interval semantics may also be coupled with linear/branching/partial orders — in practice interval semantics have tended to use linear time orders. However, in theory these are orthogonal issues.

process infinitely often, or requiring that a continuously enabled transition ultimately fire.

Various proof systems and decision procedures for finite state systems can be used to check the correctness of a program or system.

In real-time temporal logics, *quantitative* properties can also be expressed such as periodicity, real-time response (deadlines), and delays. Early approaches to real-time temporal logics were reported in [10,63,111]. Since then, real-time temporal logics have been explored in great detail.

5.1.1 The TTM/RTTL framework — explicit clock linear logics

The TTM/RTTL framework was first presented in [111], and in detail in [102]. It includes the following elements: (a) a semantic model of time, (b) a generic computational model (Timed Transition Models[3]) for modelling plants and controllers, (c) an abstract specification language (Real-Time Temporal Logic), (d) verification methodologies including model-checking for finite state systems, and a deductive proof system for infinite state systems, and (e) heuristics for constructing proofs and controller synthesis methods [104,105,110,107,109,108]. The first four elements represent an extension of untimed Manna-Pnueli temporal logic [82] to timed systems. Each of these elements is discussed below.

Semantic model of time: The notion of a possible behaviour or *trajectory* σ of a system is given by an infinite sequence of states q_i and events τ_i defined as

$$\sigma \stackrel{\text{def}}{=} q_0 \stackrel{\tau_0}{\rightarrow} q_1 \stackrel{\tau_1}{\rightarrow} q_2 \cdots \stackrel{\tau_{i-1}}{\rightarrow} q_i \stackrel{\tau_i}{\rightarrow}$$

representing a possible run or computation of the system.

A *discrete* notion of time is employed using an external (conceptual) *explicit clock*. There is a distinguished *tick* transition, corresponding to the movement of the clock whose current time is represented by the variable t, with $type(t)$ (the set of all values that t can have) the nonnegative integers. The clock event *tick* occurs infinitely often ("time must make progress") in the trajectory σ. When the clock tick occurs it increments t by one ("time never decreases"). No event other than the clock tick can change the time. The clock ticks are *interleaved* with other system transitions.

[3]Originally called Extended State Machines.

Time bounds on events determine when they may occur relative to the clock ticks. A lower time bound l on the event τ means that it may not occur prior to l ticks of the clock. An upper time bound u means that τ must occur no later than u ticks of the clock (unless τ is preempted by some other event). Thus an event that is continuously enabled over an interval of time does not actually occur for l ticks of the clock, but must occur by u ticks of the clock or become disabled.

Many events may occur between two ticks of the clock, in which case the events can only by distinguished by temporal ordering, not by time. A discrete time domain necessarily sacrifices information about precise times.

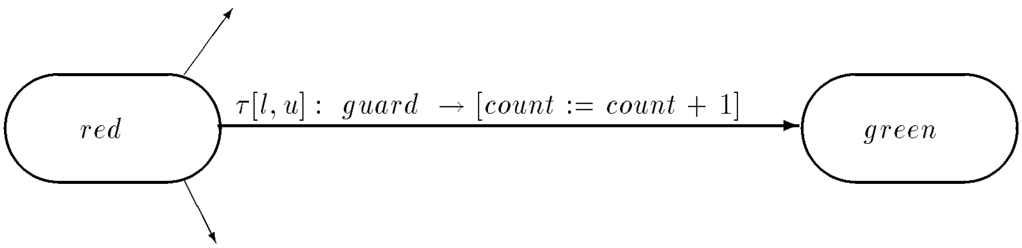
A computational model: Timed Transition Models (TTMs) provide an effective representation of realizable systems. Concrete real-time programming languages, Petri Nets and Statecharts can be mapped into TTMs. Most real-time features such as delays, timeouts and various scheduling constraints can be represented. True parallel processing, multiprogramming, communication through shared variables as well as message passing over channels can also be modelled.

A TTM M is defined as a 3-tuple $M = (\mathcal{V}, \Theta, \mathcal{T})$, where $\mathcal{V}$ is the set of *activity*[4] and *data* variables (including the clock variable t). Θ is a predicate asserting an initial condition on the variables. $\mathcal{T}$ is the set of all transitions (representing events). Each transition has an enabling condition, transformation function, and lower and upper time bounds.

TTMs may be composed in parallel with each other to obtain more complex TTMs. A *graphical language* such as Statecharts is used to visually represent TTMs to the designer. The visual notation of Statecharts is extended by annotating each transition with its enabling condition, transformation function, and lower and upper time bounds.

For the transition

[4] An *activity* variable is also sometimes called a *control* or *location* variable. An *activity* is any particular value that the activity variable may assume. The word "activity" is used rather than "state" as the word "state" is reserved for the global vector of all values of activity and data variables. An activity has persistence as opposed to a transition which occurs instantaneously.



the variable *count* is an example of a data variable that counts the number of times the traffic light has changed from the activity *red* to *green*. The intended operational meaning is as follows. The transition is said to be *enabled* if (a) the guard is true, and (b) the TTM currently resides in activity *red*. The lower time bound l guarantees that once τ becomes enabled, then it may not occur for at least l ticks of the clock. Meanwhile, other enabled transitions may occur and disable τ. After τ has been continuously enabled for more than l ticks of the clock, then the transition τ *may* be taken, instantaneously moving the TTM from red to the activity *green*, and simultaneously incrementing the counter by one. After having been continuously enabled for u ticks of the clock, either τ *must* be taken before the next clock tick or be preempted by the occurrence of some other transition that disables τ.

A *spontaneous* transition $\tau[0, \infty]$ represents an event that may occur at any moment. However, it may also delay occurring forever. Spontaneous transitions are useful for modelling nondeterministic unpredictable behaviour in the plant. Examples include the failure of various devices (e.g. a pump or a valve). Spontaneous transitions can also representing situations where the designer initially has no knowledge of the time bounds.

More than one transition may be enabled and eligible (by virtue of its bounds) to occur at any point in time. In such a case, the order of transitions (in a behaviour) are chosen nondeterministically. This is typical of how the interleaving approach represents a process that must occur along several edges at the same time.

In [71], a set of transformations for taking a given TTM into a new TTM that is bisimulation equivalent is proposed. These transformations can be used to show that an implementation TTM is correct with respect to a specification TTM (in the style of process algebras discussed in the next section).

Concise specification language: RTTL (real-time temporal logic) is used to specify the properties to be verified. Given the timed transition model $SUD \stackrel{\text{def}}{=} plant \| controller$, the objective is to check that SUD satisfies a specification S (which is a formula of RTTL).

An example of an RTTL formula is the *bounded response* time given by

$$\forall T[(red \wedge t = T) \rightarrow \Diamond(green \wedge T + 3 \leq t \leq T + 5)]$$

In the above formula, the clock variable t is a flexible variable. The quantified variable T is a rigid variable. The clock variable is called flexible because it changes from state to state in the trajectory. By contrast, the rigid variable T retains the same value over all states in the trajectory, and is used to record the time when red becomes true. Accordingly, the above formula asserts that if the traffic light is red at time T, then eventually within 3 to 5 ticks from T the light must turn $green$. The bounded response property may be *abbreviated* by the formula

$$red \rightarrow \Diamond_{[3,5]}\, green$$

RTTL is an *explicit clock* logic because an RTTL formula may explicitly use the clock time variable t with any of the arithmetic operators (e.g. $+, -, =, \leq, >$), using arbitrary first order quantification over rigid time variables. Hence RTTL is very expressive but undecidable (see Section 5.1.2). RTTL may be used to: (a) refer to absolute times (e.g. "do some action on January 22nd 1998"), or (b) relate adjacent temporal contexts (e.g. "every stimulus A is followed by a response B, and then by another response C that is within 5 ticks of the original stimulus A").

Verification via model-checking and proof systems: The *model-checking*[5] problem is as follows: check whether all legal trajectories of a *finite state* timed transition model SUD satisfy a specification S. The challenge is to construct a finite reachability graph, even though the time domain is unbounded.

[5]The notion of model-checking was first explored by Clarke and others [20] in the context of (untimed) branching time temporal logics.

In [105,109], algorithms (and an implemented verifier) are provided for checking a subset of RTTL specifications. The verifier represents processes in a fashion close to the mathematical definition of TTMs. All kinds of data variables are treated including booleans, enumerated or numerical types, lists, sets and sequences. If an RTTL property fails to hold, then the failing trajectories are provided, making it possible to debug the system.

Model-checking suffers from exponential explosion in the size of the state space, when dealing with the parallel composition of processes. Model-checking is suitable for checking small core parts of real-time systems. Reduced models, such as the synchronization skeleton of a mutual exclusion protocol, can also be treated. Large systems will often require different techniques.

For large systems (or infinite data domains) a deductive proof system must be used. A sound first order proof system for RTTL is presented in [104]. The proof system reduces to standard Manna-Pnueli temporal logic if all the upper and lower bounds are zero and infinity respectively.

An advantage of RTTL is that no new temporal operators are introduced. As a result, all the proof rules of Manna-Pnueli temporal logic can be used. In addition, certain rules are added for the real-time part of the reasoning [104,108]. Systems in which there are mixed fairness and time constraints can also be handled.

Pragmatics — Semi-automated proof and synthesis methods: A proof system, with perhaps some small examples to illustrate the method of proof, is not on its own sufficient to make the proof system practically useful for infinite state systems. Additional guidelines and heuristics must provide insight to the design and verification procedures. Automated support tools (beyond model checking for finite state systems) are needed. Theorem provers must be provided with adequate tacticals (built in heuristics) to support verification.

RTTL has heuristics for doing proofs using proof diagrams and weakest-preconditions [104,108]. A proof diagram is an abstract view of a state reachability graph. It is not confined to finite state systems because a node in the proof diagram is a predicate that can characterize a possibly infinite set of states. The proof diagram contains the intuition of system executions without the distracting proliferation of states. Most of the reasoning takes place in the ordinary predicate calculus, with temporal or real-time reasoning introduced only where absolutely needed.

Constraint logic programming has been used to semi-automate the heuristics. The language CLP($\Re$) was used initially [106], but more recently the constraint logic language PrologIII [107] has been investigated for providing automated support. Methods of synthesizing controllers (for some infinite state systems) to satisfy given plant specifications are provided in [107] (see also the last paragraph of Section 4.2.2).

In [64], it is claimed that explicit clock logics such as RTTL do not "hide" time in accordance with the original philosophy of temporal logic (which was to abstract from time as much as possible). Specifications are not as succinct as hidden clock logics. However, most of the reasoning about time in RTTL involves the use of abbreviated formulas (similar to the bounded operators of hidden clock logic). Explicit time is resorted to only in those instances where needed (e.g. to refer to adjacent time contexts or other properties that hidden clock logics cannot specify).

It has also been claimed [56,62] that while the interleaving model of computation may be adequate for qualitative analysis of systems, a more realistic model such as maximal parallelism is needed for real-time systems. This claim is refuted by the interleaving model in the TTM/RTTL framework, in which system actions are interleaved with clock ticks. A careful incorporation of time allows for an adequate representation of most real-time phenomena while preserving the simplicity associated with interleaving models, namely, at any one point in time only one transition can occur and has to be analyzed.

The main disadvantage of the framework is its lack of compositional proof methods, although some of the synthesis methods allow for a modular style of controller development. The question of modular specification and verification methods is an active area of research in the field. Manna-Pnueli temporal logic has recently been provided with a compositional proof system [82]. Since RTTL is based on the untimed Manna-Pnueli system, it appears that their compositional methods should also carry over to RTTL.

The TTM/RTTL framework is a state-based, linear discrete time, interleaved, asynchronous, explicit clock logic formalism. It is state-based because states rather than actions are the primitive components of behaviours. State-based approaches may be more general than action based methods (e.g. process algebras) because state based methods can easily encode actions as well as histories of actions It is difficult to extract the state from action based formalisms.

TTM/RTTL uses a discrete time domain rather than a dense (e.g. the rationals) time domain. If simplicity of use can be preserved, the more general modelling powers of a dense time domain would be preferable. Concurrency is modelled by interleaving rather than a partial order or maximal parallelism. It is asynchronous because a finite number of events can occur between clock ticks. In a synchronous model, all concurrent activity happens in lock-step with the tick of the clock.

Many of the choices made in TTM/RTTL computational model are independent of each other. For example, RTTL can be defined over dense as well as continuous time domains. It can be given a branching time semantics or be based on time intervals. The choice of semantics is, in principle, independent of the syntax of the specification language [61,46]. The reader is referred to [118,46,47] for other explicit clock logics. In particular, [47] compares explicit clock logics with hidden clock logics.

5.1.2 MTL — hidden clock linear logics and other RTTL fragments

Metric Temporal Logic (MTL) [64] is a fragment of RTTL in which references to time are restricted to bounds on the temporal operators. For example, the formula $A \to \Diamond_{\leq 5} B$ means that if A occurs then eventually within 5 time units B must occur. No references to an explicit clock are allowed and hence MTL is called a *hidden clock* or *bounded temporal operator* logic.

Interpretations for MTL are metric point structures based on a linearly ordered time domain. A distance function provides a metric for time. Various time constraints can be imposed on the distance function, depending on the notion of time that is used (e.g. transitivity, irreflexibility, and the existence of absolute differences).

In the hidden clock approach, $\Diamond_{\leq 5}$ is a new temporal operator that restricts or bounds the scope of the qualitative operator $\Diamond$. The bounded formula $\Diamond_{\leq 5} B$ predicts the occurrence of B within 5 time units from now. The qualitative formula $\Diamond B$ asserts that B will eventually happen, but puts no bound on when.

In [64], a real-valued time domain is used. This allows MTL to express certain properties of continuous time variables (e.g temperature and pressure) more succinctly than discrete time logics. MTL does not allow references to an absolute point in time, nor does it allow the specifier to relate adjacent temporal contexts. A sound proof system for MTL is provided.

An important extension to the literature on MTL is a compositional proof system for Occam style programs [52,50]. The proof system uses the maximal parallelism model of program execution . Because the proof system is compositional, the properties of a compound system $P_1 \| P_2$ (the parallel composition of two simpler processes $P1$ and P_2) can be deduced from specifications of its constituent parts (P_1 and P_2), without any further information about the internal structure of these parts. Compositionality is important for scaling up the application of the proof system to deal with large systems in a structured fashion.

It is not clear whether the proof system can be extended to reason about complex plant descriptions, which are not always representable in Occam. The extra *chop* operator, needed for compositional proofs, makes the reasoning relatively complex. Nevertheless, once a module's specification is fixed, any implementation of the module with the same specification can be used, without having to redo the proof.

Alur and Henzinger [6] have compared various ways in which to restrict RTTL to obtain decidable fragments of the logic. These fragments are restricted to finite state, propositional temporal properties. There are at least four interesting syntactic fragments of RTTL:

- MTL [7,64] is a discrete time propositional version of the bounded operator logic discussed above.

- XCTL [43] is a discrete time propositional explicit clock logic. The atomic timing constraints allows the primitives of comparison and addition. XCTL restricts the quantification level. It allows only one outermost level of quantification over rigid time variables. The quantification is therefore never explicitly displayed. On the other hand, it allows general arithmetic timing expressions, including addition and subtraction of variables and constants.

- TPTL [7] is a discrete time propositional logic whose timing constraints allow comparison and addition (but only of integer constants, i.e. no variables). TPTL uses auxiliary static timing variables to record the value of the clock at different states, but replaces the explicit references to the clock itself by a special type of freezing quantification.

- MITL [5] employs a dense time domain. It has a bounded-operator syntax, but cannot express punctuality properties. A *punctuality* property states that the event B follows A in *exactly* 3 seconds.

All of the above fragments can be interpreted over discrete, dense and continuous time domains. The fragments can then be compared for expressiveness, and complexity of satisfiability and model-checking.

Satisfiability is important in the homogeneous verification case. Let the implementation I and the specification S (that the implementation must satisfy) can both be given as temporal logic formulas F_I and F_S respectively, in the appropriate fragment of RTTL. Then the implementation I meets the specification S iff the implication $F_I \rightarrow F_S$ is valid (or, equivalently, if the conjunction $F_I \wedge \neg F_S$ is unsatisfiable). Only propositional versions of temporal logics are decidable for satisfiability. Decidability depends on the nature of the time domain (discrete, dense etc.) and the operations on the time domain that are permitted.

Model checking is important in the heterogenous verification case. The implementation I is given by a timed automaton or timed transition system [4,5] A_I. The specification S is given by a temporal logic formula F_S. To do the check, a timed automaton $A_{\neg F_S}$ is constructed from the negation of the specification $\neg F_S$. The implementation I satisfies the specification S precisely when the product automaton $A_I \times A_{\neg F_S}$ has no run (timed observation sequence).

Let RTTL(<,s) denote the restriction of RTTL to propositions and timing constraints containing only ordering (<), successor(s), and congruence over time (e.g. all times with an even time difference from the initial state). Thus, RTTL(<,s) is interpreted over a discrete time domain, and can be used to specify constant lower and upper time bounds on the time distance between events. The various logics are compared in Table 1 for satisfiability, model checking and expressiveness over discrete and dense time domains relative to RTTL.

For any real-time logic that is closed under boolean operations, and that can express punctuality, the satisfiability problem is undecidable for

Discrete Time	Satisfiability	Model Checking	Expressiveness
RTTL	undecidable	undecidable	
RTTL(<,s)	nonelementary	nonelementary	< RTTL
XCTL	PSPACE-complete	PSPACE-complete	< RTTL
TPTL	EXPSPACE-complete	EXPSPACE-complete	=RTTL(<,s)
MTL	EXPSPACE-complete	EXPSPACE-complete	=RTTL(<,s)
MITL	EXPSPACE-complete	EXPSPACE-complete	< RTTL(<,s)
Dense Time	**Satisfiability**	**Model Checking**	
RTTL	undecidable	undecidable	
RTTL(<,s)	undecidable	undecidable	
XCTL	?	?	
TPTL	undecidable	undecidable	
MTL	undecidable	undecidable	
MITL	EXSPACE-complete	EXPSPACE-complete	

Table 1: Comparison of linear propositional real-time logics

a dense time domain. MITL cannot express punctuality properties, and is thus less expressive than the other logics, but remains decidable even if the time domain is dense.

In [43] it is shown that XCTL and TPTL/MTL are incomparable. For each of these logics, there is a property expressible in one which is not expressible in the other. Each of these properties is a reasonable real-time requirement. In the discrete time case, TPTL and MTL are equally expressive (it is conjectured that this equality does not extend to dense domains [46]).

There are doubly-exponential-time decision procedures for both TPTL and MTL. The verification algorithm for MTL depends, exponentially, on the value of the largest time constant involved. It is a little less expensive than the algorithm for TPTL, which depends exponentially on the value of the product of all time constants.

XCTL is not closed under negation, and hence cannot be used to solve the homogeneous verification problem. However, a special model-checking algorithm for XCTL has been given that is doubly exponential in the size of the specification formula and singly exponential in the size of the model.

5.1.3 Branching time temporal logics

Linear time and branching time logics are incomparable. For example, branching time logics cannot express general fairness constraints in the syntax of the language (although certain fairness conditions can be imposed by external constraints on the reachability graphs when doing model checking for example). Linear time logics can directly specify fairness. On the other hand, branching time logics can express certain existential path conditions (e.g. it is always possible for the system to eventually do some action). Such path constraints cannot generally be expressed in linear time logics.

Although the validity problem for (untimed) branching logics is EXTIME-complete, model checking is in PTIME. Model checking is performed with a special purpose algorithm (relabelling of the program reachability graph) and does not use the satisfiability procedure. The algorithm for model-checking branching time logics is linear in the size of the reachability graph, and linear in the size of the formula to be checked.

Linear time logics cannot use this special purpose algorithm, and hence untimed model checking in the linear case is exponential in the size of the formula. Model checking for branching time logics has therefore been applied more successfully than that of linear time logics, although additional constructs must be employed to deal with fairness [20].

There is no consensus among researchers as to whether branching time or linear time logics are more suited to verification. In practice, branching time logics are usually used to verify finite state systems by model checking because of the efficiency of the model checking algorithms. Linear time logics usually come with a deductive proof system for dealing with the infinite state systems.

In [3,2], a branching real-time time logic called TCTL is proposed. It is based on hidden clock bounded operators. For TCTL, the validity problem for dense time domains is undecidable, yet model-checking is decidable. The complexity of model checking is exponential in the number of clocks (each new process or hardware device needs its own clock), and doubly exponential in the product of the timing constants that appear in the formula. It is linear in the product of program and formula size.

In [26], a bounded operator branching time logic called RTCTL is introduced. The satisfiability problem in this logic is doubly-exponential-time-complete. Model checking has a polynomial time algorithm.

A branching time logic called TPCTL is introduced in [37]. The logic deals with real-time constraints and reliability. For example, the following property can be specified: "after a request for service there is at least a 98% probability that the service will be carried out in 2 seconds".

Formulas of TPCTL are interpreted over a discrete time extension of Milner's Calculus of Communicating Systems (see process algebras later) called TPCCS. Probabilities are introduced by allowing two types of transitions, one labelled with actions and the other labelled with probabilities. A probabilistic strong bisimulation for equivalence of processes is defined, which has a sound and complete axiomatization. The model checking algorithm is exponential in the size of the TPCCS process, and polynomial in the size of the formula and number of arithmetic expressions.

Because of the action based nature of TPCTL, it is difficult to specify state-based properties such as: "henceforth, if the train is at the crossing then the gate must be down". Propositions such as "the gate is down" must be encoded indirectly through actions that change the state of the model, in which case the specification becomes unnecessarily complicated.

TPCTL is one of the few logics that can express both hard and soft real-time deadlines, a feature useful in the verification of communication protocols in noisy media. Strong assumptions on the behaviour of the medium (e.g. the medium never loses more than three consecutive messages) can be replaced with weaker assumptions (e.g. successful transmission with some probability).

5.1.4 Interval and other temporal logics

Interval based real-time temporal logic specifications have been developed in [96,36]. In [36], Moszkowski's ITL [95] is embedded in the theorem prover HOL (higher order logic). Why not use HOL directly? There are two reasons. First, ITL avoids the proliferation of time variables in specifications, as do all temporal logics. Second, ITL is sufficiently general to express any discrete computation, yet specific enough to have a natural operational interpretation, which HOL does not. Since ITL has an executable subset (called Tempura), programs and their specifications can be expressed using the same notation. Very little is known about the expressiveness and decidability of these logics in comparison to the linear and branching time logics. Interval logics have been used for specification and simulation, but not much work has been done on deductive calculi or model checking for verification.

TRIO [34] is a first order logic augmented with temporal operators (similar in style to linear temporal logics). The choice of first order logic was motivated by reasons of naturalness, simplicity and compactness of specification. The two temporal operators $Futr(A, t)$ and $Past(A, t)$ mean that A holds at a time t in the future (resp. in the past) with respect to the current time. From these two basic operators many other derived temporal properties can be defined such as "always in the future" and "sometime in the past".

TRIO does not have a deductive calculus. Instead a major goal of TRIO is executability of specifications, although this is only done on finite time domains, leaving it up to the user to extrapolate to infinite domains. This method runs the risk of making the user think a formula is true (by a check on a finite time domain) when in actuality the property is false on say a dense time domain.

5.2 Process Algebras

5.2.1 UPA — Untimed Process Algebras

Algebraic approaches such as CSP [48], CCS [90] and CIRCAL [89] have been important in analyzing concurrent processing. These process algebras provide structured methods for the analysis of discrete event systems. A few constructs lead to a language capable of expressing the full complexity of parallel or distributed computing. The constructs include sequential and parallel composition, nondeterministic choice, concealment and recursion. Several computational models have been developed giving these algebras a precise denotational semantics. The computational models lead to methods for doing compositional verification.

Algebraic laws relating the algebraic constructs allow for the transformation of one system into another. In CSP, if P is a process and a an event, then $a \rightarrow P$ denotes a process that first engages in the event a, and then behaves exactly as described by P. Shared events require simultaneous participation of both the processes involved. A typical example of a shared event is the communication over a channel in which a message is sent by one process and received by some other.

If a pair of processes initially engage simultaneously in some shared event c, then the relevant algebraic law is

$$(c \rightarrow P) \| (c \rightarrow Q) = (c \rightarrow (P \| Q))$$

STOP is a process that never engages in any events, but also never terminates. If c and d are both shared events (in the alphabet of both processes) then the law

$$(c \rightarrow P \| d \rightarrow Q) = STOP \qquad \text{if } c \neq d$$

shows how a pair of processes P and Q, running in parallel, deadlock if they disagree on what the first action should be. Non-shared events are events in the alphabet of P but not in that of Q, or vice versa. Non-shared events occur independently of Q whenever P engages in them (and vice versa). Thus the parallel composition operator $\|$ is defined in such a way that events in the alphabet of both operands require simultaneous participation of them both, whereas the remaining actions of the system occur in an arbitrary interleaving.

The *synchronous* nature of interactions simplifies the analysis of concurrent systems. It is then claimed that the asynchronous nature of lower level handshaking via semaphores, monitors and condition queues can be abstracted away. However, broadcasts to the world (as in Ethernet protocols) are not easily modelled by CSP processes.

A common computational model of a process P is a 3-tuple (E, F, D), where E is the alphabet or set of process events. F is the set of *failures* and D the set of *divergent traces*.

A trace is a behaviour of a process. It is a finite sequence of events of the process, recording the actions that the process has engaged in up to some point in time. Divergent traces ("infinite loops") occur when a process engages in an infinite unbroken sequence of internal events invisible to the environment. As a result the environment is left waiting eternally for a response.

A failure is a tuple (tr, X) where tr is a trace of P and X is a set of events offered by the environment of the process. If it is possible for P to deadlock when placed in this environment, then X is said to be a refusal of P. Thus a failure (tr, X) means that P can engage in the sequence of events recorded by tr, and then refuse to do anything more, in spite of the fact that its environment is prepared to engage in any of the events of X. The set of all traces (behaviours) of P can easily be obtained from the failures of P; the domain of the failures relation F is the set of traces of P.

Several constraints are imposed on (E, F, D) to fully capture the behaviour of nondeterministic concurrent processes. The computational model provided a mathematically precise definition of the operators (e.g. sequential or parallel composition). The model is also used to prove the correctness of the algebraic laws. An operator (such as parallel composition) is shown to be well-defined by assuming that the operands satisfy the constraints, and then demonstrating that the composition of the operands also satisfies the constraints.

A complete description of a family of increasingly sophisticated models for providing CSP specification semantics is provided in [100]. These computational models include the counter model, the trace model, the divergences model, the readiness model, and the failures model (described above).

Most existing proof methods in CSP focus on bottom-up methods, in which each component is proved and compositionally used to develop more complex systems. Recent work has been performed on top down development (refinement) from abstract high level system requirements to implemented executable component parts (e.g. as an Occam task). The algebraic rules preserve the meaning and equivalence of the abstract process and its implementation. So far, the refinement methods treat only safety properties of trace based nondivergent cyclic networks.

A specification S is allowed to be any predicate on behaviours (traces). For example, consider the specification that the process $SUD = controller \| plant$ must never deadlock. Let SUD/tr be the process described by SUD after engaging in the trace tr. Then the specification S is given by:

$$(SUD/tr) \neq STOP \qquad \text{for all traces } tr \text{ of } SUD$$

A semantic proof that SUD **satisfies** S proceeds by taking an arbitrary trace tr, and showing that in all cases there is at least one event by which tr can be extended. The use of **satisfies** is compositional, i.e. a proof of a property of a compound process can be constructed from a proof of correctness of its parts.

Another way of doing proofs is to provide a proof system. For example, an axiom for the process $STOP$ is given by $STOP$ **satisfies** $tr = <>$, where $<>$ is the empty trace. An axiom or rule is provided for each operator. A proof that P **satisfies** S is reduced to a number of smaller derivations on the syntactically simpler subcomponents of P. Ultimately, the verification task is reduced to tautology checking of statements written in the specification language. Allowing specifications to be any predicate over traces has the advantage of expressivity, but its generality makes fully formal verification difficult. For this reason, a restricted specification language such as temporal logic is often used [128].

In other frameworks such as CCS, the emphasis is on defining a series of equivalences (bisimulations), each equivalence defining a different model of concurrency. Thus certain processes that might be considered identical in CSP, would be different in CCS. CCS has a form of modal logic to specify the observable behaviour of processes. CSP has a richer set of laws than CCS allowing for optimizing designs and implementations. CCS concentrates on a minimal set of operators needed for the full expression of nondeterministic concurrency and its resulting equivalences.

In CCS, a system is verified by using the notion of a bisimulation. For example, consider a protocol verification problem where one has a specification S and an implementation I. The main idea is that both are formulated in the same language, namely, as processes in a process algebra. The specification is abstract and high level. The implementation contains many details, data structures and microcode. One can abstract from the implementation details by renaming certain internal actions to be a silent action. Then, one can apply the axioms of the algebra for proving the equality between the specification and the implementation. Bisimulation on finite state automata can be decided in $O(m \log n)$ time (using the Paige/Tarjan algorithm).

Let $\|$ stand for the parallel composition operator with some hiding of internal actions. Let $\sim$ stand for the bisimulation equivalence between processes. The specification S of required system behaviour is specified as a (simple) process. The object is to find the unknown process *controller* that is the solution to the equation

$$plant \| controller \sim S$$

Thus the notion of correctness is captured by bisimulation. Using equational laws such as $P \| Q \sim Q \| P$, we can try to solve the above equation for *controller*.

5.2.2 TPA — Timed Process Algebras

Untimed Process Algebras have been extended with timing constructs in several ways including (in alphabetical order):

- ACP_ρ (Real-Time ACP) [8]. A dense time domain is used.
- ATP (Algebra of Timed Processes) [97].
- CCSR (Calculus of Communication Shared Resources based on CCS) [32,33]. This algebra also provides a proof system for dealing with priority based access to scarce resources.
- TCSP (Timed CSP) [126,127,132,128]. A dense time domain is used.
- TCCS (Temporal CCS) [93]. A complete set of axioms is presented for discrete time domains, but the semantics can be given as either discrete or dense.

- TCCS (Timed CCS) [144]. Discrete or dense time domains can be used.
- TPCCS (Timed Probabilistic CCS) [37]. TPCCS was already discussed in the context of branching time temporal logic in the previous section. A discrete time domain is used.
- TPL (Temporal Process Language) [45]. A discrete time domain is used.
- U-LOTOS (Urgent LOTOS) [14]. Discrete or dense time domains can be used.

We mention a few examples illustrating syntactic extensions for timed behaviour and then describe the general principles involved.

In TCSP, the CSP syntax is enriched with the additional process `WAIT d`, where `d` is a non-negative unit of time. The wait process terminates successfully after `d` units of time. All events recorded by processes relate to a conceptual global clock, and a process can only engage in a finite number of events in a bounded period of time.

There is a constant delay δ associated with each action. Consider a user interface of a vending machine VM. The insertion of a coin is modelled by the action *coin* and a time t_{drop} is the time allowed for the coin to drop before the event *button* is made available. The user then presses the button and the machine then offers a drink *coke* after a short delay t_{coke}. If the operator ";" stands for sequential composition of processes (i.e. $P;Q$ means do P then do Q) then the timed behaviour of the vending machine can be specified as [21]

$$\begin{aligned} VM \;=\; & coin \;\rightarrow\; WAIT(t_{drop} - \delta); \\ & button \;\rightarrow\; WAIT(t_{coke} - \delta); \\ & coke \;\rightarrow\; WAIT(t_{reset} - \delta); VM \end{aligned}$$

The vending machine presents the user with no choice of product, so the button is an unnecessary feature of the interface. The hiding operator $VM \backslash button$ may be used to conceal the *button* event from the user. Hidden events occur as soon as they become available so we obtain using the algebraic rules

$$\begin{aligned} VM\backslash button &= coin \rightarrow WAIT(t_{drop} + t_{coke} - \delta); \\ &\quad\ coke \rightarrow WAIT(t_{reset} - \delta); VMS \end{aligned}$$

The delays before and after the *button* event are unaffected by the hiding operator.

In TCCS (temporal CCS), the construct $(t).P$ denotes a process that will evolve into process P after exactly t units of time. $\delta.P$ denotes the process that is willing to wait any amount of time before behaving like P.

As an example, consider the alternating bit protocol ABP. The process ABP is the parallel composition of the sender, the receiver and the medium. Internal events (such as the acknowledgment signal between the medium and the sender) are hidden or projected out.

In the internal behaviour of the sender, r is the time that the sender will wait after sending a message before assuming the message has been lost and retransmitting.

The specification S that ABP must satisfy is

$$S = \delta.send.(d).receive.(d).S$$

i.e. S is a process that after some time sends a message, followed by a transmission delay of d clock units. The transmission delay is the time that it takes to transmit the message and receive an acknowledgment. Once the message is received and acknowledged (this also takes d clock units), the same behaviour is then repeated. The protocol is verified to be correct by showing that if $r > 2d$ then

$$ABP = S$$

The equality is interpreted in a labelled transition graph semantics. It means that the graph of ABP is the same as that of S, modulo the occurrence of any number of internally hidden transitions.

In [32,33], an extended version of CCS called CCSR is presented. CCSR deals with timing properties as well as resource sharing based on a priority semantics. Most real-time formalisms capture delays due to synchronization. Resource specific details are abstracted out by assuming idealistic operating environments. The problem with this abstraction is that true parallelism may take place only at the system level where a group of shared resources is executed simultaneously. Each resource, however, is inherently sequential in nature. A resource can only execute a single action at one point in time. This constraint leads naturally to an interleaving notion of concurrency. On the other hand, scheduling algorithms ignore the effect of process synchronization except for simple precedence relations between processes.

The computation model of CCSR treats synchronization and scheduling. It is resource based in that multiple resources execute synchronously, while processes assigned to the same resource are interleaved according to their priorities. CCSR possesses a prioritized equivalence for terms, based on strong bisimilarity. An equational proof system is provided for syntactic manipulation of terms based on resource configuration and priority ordering.

In [98], an overview of the above timed process algebras (TPAs) is presented as well as a unifying framework for treating them. The authors of [98] point out that most TPAs implicitly adopt the following view concerning their operation:

- A timed system is the composition of cooperating sequential components or processes. Each component has (semantically) a time variable as part of its state defined on an appropriate domain D with binary operation $+$ (addition on nonnegative numbers). A component modifies its state either by executing some atomic action (atomic actions take no time) or by increasing its time variable (letting time progress).

- The time variable increases synchronously in all processes only if all components accept to do so. (In some TPAs this results in the counterintuitive notion that time stops).

- An execution sequence takes place in two phases. In phase 1, components may execute a finite though arbitrarily long sequence of actions either independently or in cooperation with each other. In phase 2, components coordinate to let time progress by some finite (or infinite) amount.

An untimed process algebra is a quadruple $UPA = (O, A, R, \sim)$ where O is a set of operators defining the syntax of the language. L is a set of transition labels (actions). R is a set of rules defining the operational semantics or models, associating with each term of the language a transition system. The relation $\sim$ is a behavioural equivalence (e.g. bisimulation) defined over the models.

A labelled transition system (whose states are process expressions) is the main computational model considered for the sake of comparing the various algebras. $P \xrightarrow{a} Q$ means the process P may perform the atomic and timeless action a and then it behaves like Q. The transition $P \xrightarrow{d} Q$ means that the process P may idle for d time units after which it behaves like Q. A time domain is a commutative monoid $(D, +, 0)$. As D is usually infinite, models of processes are generally infinitely branching transition systems.

A timed process algebra is obtained from the untimed algebra $(O, A, R, \sim)$ by adding a set O' of time constraining operations. The timed process algebra is then defined as $TPA = (O \bigcup O', A, R', \sim')$. It must then be decided how to obtain the operational semantic rules R' and the strong equivalence $\sim'$ with respect to the labelled transition system.

The two main constraints that must be satisfied are:

Semantics Conservation. The untimed process and its timed equivalent should have the same behaviour as long as we observe execution actions only. This means that the untimed rules R remain valid in the corresponding TPA, as far as they are applied on terms of the UPA.

Isomorphism. For any terms P, Q of UPA, the equivalence $P \sim Q$ holds iff $P \sim' Q$. This requirement guarantees that any theoretical development in UPA remains valid in TPA and conversely.

See [98] for a further discussion of how and to what extent these constraints are satisfied in the various timed process algebras.

An important issue that arises is the *finite variability* property (or non-Zeno behaviour) of the algebras. TCSP is the only algebra for which all processes satisfy the finite variability property, namely, that a process can only perform a finite number of actions in a finite time interval. A process that has Zeno behaviours may be unrealizable. In TCSP, non-Zeno behaviour is achieved by enforcing a system delay between two actions of a sequential process. Thus the isomorphism requirement is not satisfied, and hence not all laws of CSP are valid in TCSP. The assumption of a system delay seems to be the only solution that ensures finite variability, but yields instead a complicated theory which also destroys the abstractness of time [98].

In algebras such as TCCS, U-LOTOS and ACP_ρ there exist processes whose models can block the progress of time. For timed systems it is natural to demand that a terminated process does not block time; but then a distinction must be made between termination and deadlock. The designers of algebras that block time admit that such time-locks are counterintuitive. However, they claim that time-locks can detect certain types of timing inconsistencies and unrealizable specifications.

In summary, verifying that a timed process P satisfies a specification S depends on whether a dual language framework is used or a single language framework.

- In a dual language framework the specification S is a formula in a high level logical language (e.g. S is a predicate over behaviours or a temporal logic formula). The process P is an expression of an algebraic process construction language. A proof system is provided in which there is a rule relating each operator of the algebra to a predicate representing its satisfying behaviours. A formal derivation that P **satisfies** S can then be constructed. This is the main method used for verification in TCSP. The TCSP proof system has been used to prove the correctness of control software for aircraft engines, of realistic telephone switching networks and of a local area network protocol [128]. In TCSP there is also a method of verification called *timewise refinement*. There is a hierarchy of theories from the most abstract untimed theories to the most detailed timed theories. Some specifications may be decomposed into a functional part and a timed part. The functional part of the specification is checked more simply in the untimed model, by using correctness-preserving migration between models in the hierarchy. Only the timing part of the specification need be checked in the more complex timed model.

- In a dual language system it is possible to consider automated model checking of finite state systems. Such model checking has been investigated for the algebra TPCCS [37]. For ATP [99] processes are translated into timed transition systems [3] which has already available model checking algorithms (see section on temporal logics).

- In a single language framework both the detailed implementation P of the system and its abstract specification S are terms of the process algebra. A bisimulation is then used to show that the implementation is equivalent to its abstract specification, i.e. $P \sim S$.

Often, the requirement that two specifications be equivalent is too strong. For example, it may not be necessary to prove two processes equally fast, as perhaps "faster" would be sufficient. On the other hand equivalences are interesting tools for abstractions as a larger concrete implementation can be replaced by its smaller high level specification, and vice versa [37]. The approach of using a bisimulation has been successful in the untimed case. However, some researchers claim that it will not be useful in the timed case owing to the complexity introduced by the timed constructs [21].

The dual language approach of using a specification language such as temporal logic, has the advantage of separation of concerns. Each relevant requirement can be expressed as a separate formula, and each formula can be verified separately [37].

5.3 RTL — Real Time Logic and Event Action Models

Real-Time Logic (RTL) is a formal language for reasoning about events and their times of occurrence [56,57,92]. An external event, such as an operator pushing a button, is denoted $\Omega BUTTON$. The start and stop phase of a compound sampling event is denoted $\uparrow SAMPLE$ (denoting the beginning the action), and $\downarrow SAMPLE$ (its completion). The time domain is the set of nonnegative integers. Time is captured by the occurrence function @ which assigns time values to event occurrences. This function is defined as

$$@(EVENT, i) = \text{time of the } i\text{-th occurrence of } EVENT$$

The specification which asserts that upon pressing the button the action SAMPLE is executed within 30 seconds, is written as follows:

$$\forall x[@(\Omega BUTTON, x) \leq @(\uparrow SAMPLE, x)$$

$$\wedge @(\downarrow SAMPLE, x) \leq @(\Omega BUTTON, x) + 30]$$

Let the system under development SUD be represented by the conjunction of a set of RTL formulas. Let the specification of legal behaviour be another RTL formula S (e.g. specifying a safety property that one plant event must occur 20 seconds after some other event). Then the objective is to prove that $SUD \Rightarrow S$.

Once the syntax of the logic is fixed as above, an underlying computational semantic model is needed. This is provided by the event-action specifications, which is a textual language for specifying event ordering that is easier to use (and more readily understandable than RTL). The event action model can be used to generate sequences of event sets indexed by times of occurrences, thus providing an operational semantics for interpreting RTL formulas. $SUD \Rightarrow S$ is valid precisely when the formula F, given by $SUD \wedge \neg S$, is unsatisfiable (has no satisfying sequence of event sets).

Various techniques are provided for showing F to be unsatisfiable. One possibility is to translate F into a corresponding formula F' in quantifier free Pressburger arithmetic extended with uninterpreted integer functions. Pressburger arithmetic is doubly exponential in complexity, and undecidable with even a single uninterpreted function. Therefore, only a semi-decision procedure can be given. The verification is also impractical for large systems. A resolution based inequality prover (the Bledsoe-Hines algorithm) has been used with slightly better results than the Pressburger procedures.

To improve the decision procedures, two approaches have been used. First, the RTL formulas are limited to safety assertions. Second, the RTL formulas can be better structured, using domain knowledge, into a computation graph. With this improved structuring (and hence better choice of clauses in the resolution theorem prover), an exponential time decision procedure (in the worst case) is obtained. The decision procedure involves checking for positive cycles in the computation graph, and checking for unsatisfiability based on the positive cycles detected.

In [58], a visual formalism called Modecharts is introduced. Modecharts have some similarities to Statecharts. Modecharts specify a decidable fragment of RTL, in a "natural", state-based, visual fashion preferred by design engineers. A method is provided for translating Modecharts into computational graphs, from which the verification can be performed. For modular extensions to RTL and the use of the HOL theorem prover see [78].

RTL's event occurrence function allows for a rich expression of periodic and non-periodic real-time properties. However, unrestricted RTL is undecidable. It does not treat data structures or infinite state systems. RTL formulas impose a partial order on computational actions which is useful for representing high level timing requirements.

5.4 Assertional and other formal methods

5.4.1 Real-Time Hoare Logic

In [51,50], an assertional style of reasoning about real-time systems is introduced, based on classical Hoare triples $\{q\}P\{r\}$. P is a program, and q and r are first order predicates [49]. Hoare triples can only express partial correctness (properties that hold *if* the program terminates). This however is not suitable for real-time programs which must deal with non-terminating processes and intensive interaction with the environment.

Therefore Hoare triples are extended with a third assertion C called a *commitment*. This leads to formulas of the form C: $\{q\}P\{r\}$. The commitment expresses the communication interface of the program P (part of the controller) with the environment (the plant or other parts of the controller). The commitment must be satisfied by terminating and non-terminating computations. Communication between processes is by message passing only (there are no shared variables), so C contains no program variables. The special time variable *time* is allowed in the commitment.

The specification asserting that P does not perform any communication on channel c is written

$$(\forall t : \neg comm \text{ via } c \text{ at } t) : \quad \{time = 0\}P\{true\}$$

The specification asserting that P terminates after 12 time units, incrementing x by 5, is written

$$time < 12 : \quad \{x = v \wedge time = 0\}P\{x = v + 5\}$$

A sound and relatively complete compositional proof system is provided for Occam style programs. The first extension of Hoare style reasoning from sequential to qualitative (non real-time) concurrent reasoning is provided in [113], but the proof system required the presence of all the code and hence was not compositional. To aid in the development of the proof system, a denotational semantics for Occam is developed. The execution model is based on maximal parallelism and synchronous message passing.

[50] compares MTL (metric temporal logic) and the assertional Hoare style proof systems. In general MTL allows for more concise real-time specifications than the more verbose assertional style. The assertional style of reasoning is more suitable for detailed reasoning about sequential program fragments. A combination of both styles of reasoning is recommended. Use concise MTL formulas for top level specifications and the initial design outline. Use the assertional style to perform detailed verification of sequential components. In both cases, compositional proofs of correctness are complex even for simple examples.

5.4.2 Putting Time into Proof Outlines

In [131] a proof outline logic is provided for concurrent programming with additional rules to perform real-time verification. The logic is defined using the maximal parallelism computational model.

Safety properties assert that something bad does not occur during a run of the system. For example, assume that we wish to prove that $\neg Q$ never occurs during any run of the system. To prove this, the designer typically searches for an invariant I, that characterizes current and possibly past program states, and is not invalidated by system actions. If the invariant I holds in all initial states of the system, and $I \Rightarrow Q$ is valid, then $\neg Q$ never becomes true.

Timing properties can often be formulated as safety properties. For example, in a process control system the elapsed time between stimulus and response must be bounded by some time d. The "bad thing" (reaching time d before seeing the response) can be specified in terms of times at which various control points in the real-time program are reached.

A logic L to verify untimed safety properties can form the basis of a logic L' to verify timing properties. To do this, constructs must be added so that in L' the designer can specify in the predicates I and Q information about the times at which events of interest occur. It must also be possible to establish that the actions of the system do not invalidate the invariant I. This means that the rules of L have to be refined with information about execution times. In [131] such a proof outline logic (POL) is developed.

Knowledge of the execution times of individual actions is needed to reason about timing properties.

POL is similar to the Hoare-style logic in [135], which augments each action with an assignment to a clock variable to keep track of elapsed time. In contrast, POL augments the assertion language with additional terms. This results in a more expressive language.

The assertional logic of Hooman in [51] (discussed above) deals with synchronous message passing and has no shared variables (which makes compositional reasoning easier). Only certain times can be recorded, e.g. times at which externally visible events (such as communication) can occur, and the times at which program execution starts and terminates. Information about control points cannot be specified because internal activities betray the internal structure of a component, which would destroy compositionality. Certain liveness proofs can also be made. In contrast, POL deals with shared variables, but then relaxes the compositionality requirement. Liveness proofs cannot be given. POL can also be used to represent synchronous communication. The two logics are therefore incomparable although both are assertional in style.

Another assertional logic which is being extended to deal with real-time is the temporal logic of actions discussed in [68]. A report on this work was not available at the time this survey article was written.

5.5 Hybrid Models

Many of the computational models discussed previously assume that the plant or environment can be modelled as a *discrete event dynamic system*. This assumption is useful as it allows a symmetrical treatment of controllers (software implemented on a digital computer) and plants. It also encourages structured modular analysis. Many simplifying assumptions can be made that make the analysis much easier. The discrete description of digital logic circuits via boolean functions is an example of where a discrete model simplifies analysis. If the digital circuit was viewed as a continuously changing numerical vector that is a solution to a set of differential equations, analyzing its behaviour would be much more difficult compared to treating it as a boolean system.

While the discrete event approach is justified in many situations, there are other contexts in which the assumption of discreteness may lead to unreliable conclusions. A control program driving a robot through a maze is an example where it may be necessary to introduce a continuous variables analysis. Continuous variable dynamic systems have been extensively studied by control theorists [143].

Recently researchers have turned their attention to *hybrid systems theory*, i.e. systems consisting of a non-trivial mixture of discrete and continuous components. For such systems it is hoped to obtain just the right kind of computational model in which each kind of analysis (discrete and continuous) can be carried out in its relevant domain but in an integrated fashion.

In [101], a hybrid formalism based on Statecharts and real-time temporal logic is proposed. The underlying semantic model is that of hybrid traces which are mappings from continuous time to system states. A phase transition system is used as the computational model to represent the behaviour of the system. A behaviour is a sequence of phases alternating between continuous and discrete changes. A continuous phase takes some amount of time and allows changes over this time that are governed by differential equations. A continuous phase is followed by a discrete phase. The discrete phase consists of a finite number of discrete transitions, each of which causes a possibly discontinuous change in the value of the variables.

Two specification formalisms are allowed: Statecharts and temporal logic. Statecharts are used to describe the detailed behaviour of the system. An unstructured state (of the Statechart) may be labelled with the appropriate differential equations that describe the system behaviour while in that state. When a discrete event occurs (e.g. the pilot requests a new mode of flight to a higher altitude) then there is a discontinuous change in the variables (e.g. in the setpoints), and an instantaneous change to a new state in which some other continuous behaviour is observed.

A modest extension to real-time temporal logic is made enabling it to refer to continuous change and time. A proof system for safety properties is provided, leaving a more thorough investigation of the subject to subsequent research.

In [85], weakest precondition predicate transformers are used to derive sequential process control programs. Only one extension to Dijkstra's calculus had to be made, involving the use of function-valued auxiliary variables, for reasoning about physical processes during program transitions. A proof system for hybrid systems can thus be obtained with only some modest additions to Dijkstra's calculus. The proof system is limited to sequential real-time programs.

In [99], a generalization to Action Timed Graphs is proposed which allows for the modelling of hybrid systems. A set of timers, and a set of state variables is added to the model. The state variables to do not change linearly with respect to time, but according to a continuous evolution function (e.g. differential equations). No proof system is provided.

6 Future trends

This survey has shown that formal methods for real-time systems are being energetically explored. The proliferation of formalisms indicates that the field is still in its "adolescence". It would therefore be hazardous to speculate on new developments and trends and their range of applicability. Nevertheless, we summarize below some of the main ideas mentioned in this survey, and also mention further areas of future development as expressed in [40,22]:

- Real-time software is part of a greater system. Computational models must be flexible enough to represent both controllers (software) as well as the plant (the environment). Software harms no one. It is the plant (that the software must control) that does the damage. The total system, consisting of controllers and plant, must be modelled and validated.

- Two possible strategies for improving reliability of complex real-time systems have been developed. Industrial institutions have promoted structured, visual design methods and the use of real-time programming languages. In academic institutions, formal analysis and verification methods have been provided. Both strategies need to be further explored and integrated with each other.

- Structured development tools need to be improved to allow for powerful execution and compilation facilities (e.g. into Ada or Occam). Such tools are indispensable in the design of large scale systems. Improved visual methods for structuring systems should be explored. This will make the tools accessible to design engineers. Verification methodologies should be incorporated into these tools. Research into fast algorithms for automatic verification of finite state systems is already showing some promising results. The problems of state explosion must be dealt with by symbolic techniques and the analysis of reduced order systems.

- Different formal methods have evolved independently of each other. The best features of each method should be synthesized and incorporated into a more powerful and hierarchical unified theory. The expressiveness of specification methods should be improved without resulting in intractability. It is important to combine existing methods, broaden their scope and investigate their limits of applicability. New formalisms such as hybrid systems theory should be explored. This, and more, will be required to deal with realistic large scale systems.

- Scheduling of processes and the verification of timing properties must be integrated into a comprehensive theory. To do this, it may be necessary to develop new methods for efficient and concise expression of timing constraints. This will result in complex runtime environments that must be dealt with in the new formalisms.

- Totally automated synthesis of controllers that will satisfy arbitrary properties is probably not obtainable. That does not prevent theorists from exploring semi-automated methods and heuristics for disciplined development of controllers from their specifications. In addition, partial verification methods can be explored either by applying the methods to a small core part of the system (the most critical one), or by applying the method to an abstraction (simplified representation) of the system. In either case, partial verification methods must be accompanied with a clear explanation of the validity of these partial results to total behaviour, and how these methods can be safely applied to real world systems.

- Collaboration between industry and researchers on real problems is needed to ensure the right balance of practice and theory. Practical methods often lack a theoretical basis, thereby limiting their analysis capabilities. Theoretical work can be stimulated by the need to deal with practical applications.

There is a need to start using real systems as examples. These examples should be documented and published by industries interested in looking for solutions. Current models and procedures work on tiny examples in research papers. But these methods often carefully leave out anything the model cannot handle, which usually includes most things needed in a real specification. There is nevertheless cause for cautious optimism. Recent collaboration has shown some signs of closing the gap between theory and practice [39,22].

Acknowledgements: I gratefully acknowledge help received from Nancy Leveson, Willem-Paul de Roever, Al Mok, Dave Parnas, Murray Wonham, Jozef Hooman, Ron Koymans, Ben Moszkowski, Mike Gordon, Farnam Jahanian, Insup Lee, Rajeev Alur and Roger Hale. Of course, any errors or misinterpretation of concepts is my sole responsibility.

References

[1] B. Alpern and F.B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Language Systems*, 11(1), 1989.

[2] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, Dept. of Computer Science, Stanford, CA 94305 USA, 1991.

[3] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking for real-time systems. In *Proceedings 5th Conference on Logic in Computer Science*. IEEE, 1990.

[4] R. Alur and D.L. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *ICALP 90: Automata, Languages and Programming*, LNCS 443, pages 322–335. Springer-Verlag, 1990.

[5] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, 1991.

[6] R. Alur and T.A. Henzinger. Logics and models of real-time: A survey. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[7] Rajeev Alur and Thomas Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings of the 5th Anual IEEE Symposium on Logic in Computer Science*, pages 390–401, June 1990.

[8] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. Technical Report CS-R9053, Center for Mathematics and Computer Science, Amsterdam, 1990.

[9] A. Benveniste and P. LeGuernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Trans. on Automatic Control*, 35(5):535–546, May 1990.

[10] A. Bernstein and P.K. Harter. Proving real-time properties of programs with temporal logic. In *Proceedings of ACM SIGOPS 8th annual ACM Symposium on Operating Systems Principles*, pages 1–11, December 1981.

[11] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. Technical report, Ecole Nationale Superieure des Mines de Paris, 1988.

[12] B. Berthomieu and Michael Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3):259—273, March 1991.

[13] J. Billington, G.R. Wheeler, and M.C. Wilbur-Ham. PROTEAN: a high-level Petri net tool for the specification and verification of communication protocols. *IEEE Transactions on Software Engineering*, 14(3):301–316, March 1988.

[14] T. Bolognesi and F. Lucidi. LOTOS- like process algebra with urgent or timed interactions. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[15] K.P. Brand and J. Kopainsky. Principles and engineering of process control with Petri nets. *IEEE Transactions on Automatic Control*, 33(2):138–149, February 1988.

[16] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE : a declarative language for programming synchronous systems. In *Proc. 14th ACM Symposium on Programming Languages*, Jan. 1987.

[17] J.F. Cassidy, T.Z. Chu, M. Kutcher, S.B. Gershwin, and Y. Ho. Research needs in manufacturing systems. *IEEE Control Systems Magazine*, 5(3):11–13, August 1985.

[18] CCIT. CCIT high level language CHILL recommendation z.200, CCIT, Geneva, 1980.

[19] K.M. Chandy and J. Misra. *Parallel program design.* Addison-Wesley, 1988.

[20] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[21] J. Davis. *Specification and Proof in Real-Time Systems.* PhD thesis, Oxford University Computing Laboratory, 1991.

[22] W.-P. de Roever. Foundations of computer science: Leaving the ivory tower. In *EATCS Bulletin.* EATCS, June 1991.

[23] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[24] E.A. Emerson and E.C. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[25] E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'not never' revisited: on branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, January 1986.

[26] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinisvan. Quantitative temporal reasoning. In E.M. Clarke, A. Pnueli, and J. Sifakis, editors, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*. Springer-Verlag, Lecture Notes in Computer Science, 1989.

[27] F.S. Etessami and G.S. Hura. Rule based design methodology for solving control problems. *IEEE Transactions on Software Engineering*, 17(3):274–282, March 1991.

[28] N. Francez. *Fairness*. Springer-Verlag, 1986.

[29] A. Gabrielian and M.K. Franklin. State-based specification of complex real-time systems. In *Proceedings of the 9th Real-Time Systems Symposium*, pages 2–11, December 1988.

[30] A. Galton, editor. *Temporal logics and their applications*. Academic Press, 1987.

[31] J.R. Garman. The bug heard round the world. *ACM SIGSOFT Software Engineering Notes*, 6(5), 1981.

[32] R. Gerber and I. Lee. Ccsr: A calculus for communicating shared resources. In *CONCUR'90*, LNCS 458, pages 263–277. Springer-Verlag, August 1990.

[33] R. Gerber and I. Lee. A proof system for communicating shared resources. In *Proceedings of the Real-Time Systems Symposium*, 1990.

[34] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.

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

[36] R. W. S. Hale. Using temporal logic for prototyping: The design of a lift controller. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, LNCS 398. Springer-Verlag, 1989.

[37] Hans A. Hansson. *Time and Probability in Formal Design and Distributed Systems*. PhD thesis, Uppsala University, Dept. of Computer Science, S-751 20 Uppsala, Sweden, 1991.

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

[39] D. Harel. Biting the silver bullet: Towards a brighter future for systems development. Technical Report CS90-08, Weizmann Institute, 1990.

[40] D. Harel. Biting the silver bullet: Towards a brighter future for system development. *Computer*, 25(1):8–20, January 1992.

[41] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and M. Trachtenbrot. Statemate: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16:403–414, 1990.

[42] D. Harel and A. Pnueli. On the development of reactive systems. In K.R Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI*, pages 477–498. Springer-Verlag, 1985.

[43] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 402–413, June 1990.

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

[45] M. Hennessy and T. Regan. A process algebra for timed systems. Technical Report 5/91, Dept. of Computer Science, University of Sussex, UK, 1991.

[46] T. A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Dept. of Computer Science, Stanford Univ., 1991.

[47] T.A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodolgies for real-time systems. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 353–366, January 1991.

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

[49] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), October 1969.

[50] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* PhD thesis, Eindhoven University of Technology, Dep. of Maths and Comp. Sc., Eindhoven, The Netherlands, 1991.

[51] J. Hooman and W.-P. de Roever. Design and verification in real-time distributed computing: an introduction to compositional methods. In *Proceedings of of the 9th International Symposium on Protocol Specification, Testing and Verification.* North-Holland, 1989.

[52] J. Hooman and J. Widom. A temporal logic based compositional proof system for real-time message passing. In *Proceedings of PARLE89 vol. II*, LNCS 366. Springer-Verlag, 1989.

[53] C. Huizing. *Semantics of Reactive Systems: Comparison and Full Abstraction.* PhD thesis, Technische Universiteit Eindhoven, March 1991.

[54] K. Inan and P.P Varaiya. Finitely recursive process models for discrete event systems. *IEEE Transactions on Automatic Control*, 33(7):626–639, July 1988.

[55] M.S. Jaffe, N.G. Leveson, M.P.E. Heimdahl, and B.E. Melhart. Software requirements analysis for real-time process control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, 241 1991.

[56] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[57] F. Jahanian and A.K. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C36(8), 1987.

[58] F. Jahanian and D. Stuart. A method for verifying properties of modechart specifications. In *Proceedings 9th Real-time Systems Symposium*, pages 12–21. IEEE Computer Society, December 1988.

[59] C.B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice-Hall, 1986.

[60] M. Joseph and A. Goswami. Formal description of real-time systems: a review. Technical Report RR129, Dep. of Computer Science, University of Warwick, U.K., August 1988.

[61] R. Koymans. (real) time: A philosphical perspective. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[62] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. In *Proceedings of Logics of Programs (Brooklyn)*, LNCS 193, pages 167–190. Springer-Verlag, 1985.

[63] R. Koymans, J. Vytopil, and W.P. de Roever. Real-time programming and asynchronous message passing. In *Proc. 2nd Annual Symposium on Principles of Distributed Computing*, pages 187–197, Montreal, August 1983. (An extended version appeared in Information and Computation, Volume 79, Number3, December 1988).

[64] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, November 1990.

[65] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[66] F. Kroger. *Temporal Logics of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.

[67] L. Lamport. What good is temporal logic? In R.E. Mason, editor, *Information Processing 83*, pages 657–668. Elsevier Science Publishers, North Holland, 1983.

[68] L. Lamport. The temporal logic of actions. Technical report, DEC Systems Research Center, Palo Alto, CA, 1991.

[69] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 83.

[70] L. Lamport. 'Sometime' is sometimes 'not never'. *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 174–185, Jan 1980.

[71] M.S. Lawford. Transformational equivalence of timed transition models. Master's thesis, Dept. of Electl. Engrg., Univ. of Toronto, 1992. (available as Systems Control Group Report No. 9202, January 1992).

[72] N.G. Leveson and J.L Stolzy. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, SE-13(3):386–397, March 1987.

[73] S.-T Levi and A.K. Agrawala. *Real Time System Design*. McGraw-Hill Publishing Company, 1990.

[74] A.H. Levis. Challenges to control: a collective view. *IEEE Transactions on Automatic Control*, AC-32(4), April 1987.

[75] Y. Li. *Control of Vector Discrete-Event Systems*. PhD thesis, Dept. of Electl. Engrg., Univ. of Toronto, 1991. (available as Systems Control Group Report No 9106, July 1991).

[76] INMOS Limited. *Occam programming manual*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[77] N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992. (In Press).

[78] G. H. MacEwen and D. B. Skillicorn. Using higher-order logic for modular specification of real-time distributed systems. In M. Joseph, editor, *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 331, pages 36–66. Springer-Verlag, 1988.

[79] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, 15(6):663–675, June 1989.

[80] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In *Proceedings of the 14th ACM Symposium of Principles of Programming Languages*, pages 1–12, 1987.

[81] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J.W. de Bakker, W.P. de Roever, and G. Rozenburg, editors, *Models of concurrency: linear, branching and partial orders*, LNCS. Springer-Verlag, 1989.

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

[83] Z. Manna and A. Pnueli. Verification of concurrent programs: a temporal proof system. Technical report, Dept. of Computer Science, Stanford University, June 1983. See also Foundations of Computer Science IV, Amsterdam, Mathematical Center Tracts, pages 163-225, 1983.

[84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[85] K. Marzullo, F.B. Schneider, and N. Budhiraja. Derivation of sequential, real-time , process-control programs. Technical Report 91-1217, Dept. of Computer Science, Cornell University, Ithaca, New York 14853, 1991.

[86] B.E. Melhart, N.G. Leveson, and M.S. Jaffe. Analysis capabilities for requirements specified in statecharts. Technical report, Dep. of Information and Computer Science, University of California, Irvine, September 1988.

[87] M. Menasche. PAREDE: An automated tool for the analysis of time(d) Petri nets. In *International workshop on timed Petri nets*, pages 162–169. IEEE Computer Society, June 1985.

[88] P.M. Merlin and A. Segall. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, pages 1036–1043, September 1976.

[89] G.J. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, April 1985.

[90] R. Milner. *A Calculus of Communicating Systems.* LNCS 92. Springer-Verlag, 1980.

[91] R. Milner. Some directions in concurrency theory (panel statement). In *Proceedings of the International Conference on Fifth Generation Computer Systems.* ICOT, 1988.

[92] A.K. Mok. Towards mechanization of real-time system design. In *Foundations of Real-Time Computing: Formal Specifications and Methods.* Kluwer Press, 1991.

[93] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *CONCUR 90*, LNCS 458, pages 401–415. Springer-Verlag, 1990.

[94] E.T. Morgan and R.R. Razouk. Interactive state-space analysis of concurrent systems. *IEEE Transactions on Software Engineering*, SE-13(10):1080–1091, October 1987.

[95] B. Moszkowski. A temporal logic for multilevel reasoning about hardware. *Computer*, 18(2):10–19, February 1985.

[96] K.T. Narayana and A.A. Aaby. Specification of real-time systems in real-time temporal interval logic. In *Proceedings Real-time Systems Symposium*, pages 86–95. IEEE Computer Society, December 1988.

[97] X. Nicollin, J. L. Richier, J. Sifakis, and J. Voiron. ATP: an algebra for timed processes. In *Proceedings IFIP Working Group Conference on Programming Concepts and Methods*, pages 402–429, 1990.

[98] X. Nicollin and J. Sifakis. An overview and synthesis of timed process algebras. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[99] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid semantics. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[100] E.R. Olderog and C.A.R Hoare. Specification oriented semantics. *ACTA Informatica*, 23:9–66, 1986.

[101] O.Maler, Z. Manna, and A. Pnuelli. From timed to hybrid systems. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[102] J.S. Ostroff. Real-time computer control of discrete event systems modelled by extended state machines: a temporal logic approach. Technical Report 8618, Systems Control Group, Dept. of Electrical Engineering, University of Toronto, September 1986. revised January 1987.

[103] J.S. Ostroff. Synthesis of controllers for real-time discrete event systems. In *Proceedings of the 28th IEEE Conference on Decision and Control*, December 1989.

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

[105] J.S. Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, April 1990.

[106] J.S. Ostroff. Constraint logic programming for reasoning about discrete event processes. *The Journal of Logic Programming*, 1991. (In Press).

[107] J.S. Ostroff. Systematic development of real-time discrete event systems. In *Proceedings of the ECC91 European Control Conference*, pages 522–533, Paris, France, July 1991. Hermes Press.

[108] J.S. Ostroff. Verification of safety critical systems using TTM/RTTL. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992. (In Press).

[109] J.S. Ostroff. A verifier for real-time properties. *Real-Time Journal*, 4:5–35, 1992. (In press).

[110] J.S. Ostroff and W.M. Wonham. A framework for real-time discrete event control. *IEEE Transactions on Automatic Control*, April 1990.

[111] J.S. Ostroff and W.M. Wonham. A temporal logic approach to real time control. In *Proceedings of the 24th IEEE Conference on Decision and Control*, pages 656–657, Florida, December 1985.

[112] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, Jul 1982.

[113] S.S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5), May 1976.

[114] D.L. Parnas and J. Madey. Functional documentation for computer systems engineering. Technical Report TR 90-287, TRIO, Queen's University, Kingston, Ont., Canada K7L3N6, 1990.

[115] D.L. Parnas, A. J. van Schouwen, and S.P. Kwan. Evaluation standards for safety-critical software. Technical Report TR 88-220, Department of Computer Science, Queen's University, May 1988.

[116] J.L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

[117] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Annual Symposium on the Foundations of Computer Science*, pages 46–57, Providence, R.I., November 1977.

[118] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In *Formal Tehniques in Real-Time and Fault Tolerant Systems*, LNCS 331. Springer-Verlag, 1988.

[119] A. Pnueli and M. Shalev. What is in a step? In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, LNCS 298, pages 244–264. Springer-Verlag, 1991.

[120] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In J. de Bakker, W.P de Roever, and G. Rozenburg, editors, *Current trends in concurrency*, LNCS 244. Springer-Verlag, 1986.

[121] W.J. Quirk. *Verification and Validation of Real-Time Software.* Springer-Verlag, Berlin, 1985.

[122] P.J. Ramadge and W.M. Wonham. Modular feedback logic for discrete event systems. *SIAM Journal of Control and Optimization*, 25(5):1202–1218, September 1987.

[123] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, January 1987.

[124] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report MAC TR 120, MIT, February 1974.

[125] R.R. Razouk and C.V. Phelps. Performance analysis of timed Petri nets. In *Proceedings of 4th International Workshop on Protocol Verification and Testing*, June 1984.

[126] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *Proceedings ICALP 86*. Springer-Verlag, LNCS 226, 1986.

[127] G.M Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, June 1988.

[128] G.M. Reed, A.W. Roscoe, et al. Timed CSP: Theory and practice. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[129] W. Reisig. *Petri nets: an introduction.* Springer-Verlag, Berlin, 1985.

[130] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, Library of Exact Philosophy, 1971.

[131] F.B. Schneider, B. Bloom, and K. Marzello. Putting time into proof outlines. In *REX Workshop — Real-Time: Theory in Practice*, LNCS. Springer-Verlag, 1992.

[132] S. Schneider. *Correctness and Communication in Real-Time Systems.* PhD thesis, Oxford University, 1990.

[133] D. J. Scholefield. The formal development of real-time systems. Technical report, Dept. of Computer Science, University of York, England, 1990.

[134] R.L. Schwartz and P.M. Melliar-Smith. From state machines to temporal logic: Specification methods for protocol standards. *IEEE Transactions on Communications*, Com-30(12), Dec 1982.

[135] A. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, SE-15(7):875–899, July 1989.

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

[137] J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next generation systems. *Computer*, 21(10):10–19, October 1988.

[138] W.M. Turski. Time considered irrelevant for real-time systems. *BIT*, 28:473–486, 1988.

[139] USDOD. *Reference Manual for the Ada Programming Language.* Spinger-Verlag, New-York, 1983.

[140] W.M.P van der Aalst. *Timed Coloured Petri Nets and their Application to Logistics.* PhD thesis, Eindhoven University of Technology, 1992. (to appear).

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

[142] N. Wirth. Towards a discipline of real-time programming. *Communications of the ACM*, 20(8), August 1977.

[143] W. M. Wonham. *Linear Multivariable Control: A Geometric Approach.* Springer-Verlag, 3rd edition, 1985.

[144] Wang Yi. CCS + time = an interleaving model for real time systems. In *Proceedings of ICALP'91*, 1991. (Madrid, Spain).

[145] W.M. Zubrek. Timed Petri nets and preliminary performance evaluation. In *Proceedings 7th Annual Symposium on Computer Architecture*, La Baule, France, 1980.