Languages and methods for specifying real-time systems

Jan Carlson
Department of Computer Engineering
Mälardalen University, Västerås, Sweden
jan.carlson@mdh.se

Abstract

The specification of complex systems, such as software, requires well-defined languages and methods. In the case of real-time systems, where temporal correctness must be achieved in addition to functional, descriptions of time dependant behaviour must be expressable within the specification language.

This report presents a selection of languages and methods that are used to specify real-time systems, ranging from fully formal temporal logics and process algebras, to the less formal but widely used UML.

Contents

1 Introduction

The specification of complex systems, such as software, requires well-defined languages and methods. This is especially true for safety critical systems, for which faulty behaviour may cause serious damage to people or property.

A language that is to be used for specification of real-time systems must of course meet all the requirements for specification languages in general, such as readability and non-ambiguity. Additionally, it should provide means to specify temporal constraints or time dependent behaviour, and allow properties of safety and liveness to be expressed.

This report presents a selection of languages and methods that have been suggested, and used, for specifying real-time systems. The next section contains a brief introduction to the real-time area, and Section 3 gives a general discussion on specification concepts. Sections 4 to 8 describe a selection of methods and languages, and Section 9 gives an overview of real-time support typically provided by programming languages.

2 Real-time systems

The significant characteristic of a *real-time system* is the existence of requirements concerning timeliness as well as functional behaviour. The correctness of the system depends not only on the results it produces, but also on the time at which the results are available. Real-time systems range over a wide spectrum of applications, from small embedded systems in control applications, to complex software systems for controlling railway switches and power plants.

Most real-time systems are also *reactive*, i.e. they react to events or changes in their environment. However, reactive is not the same as real-time. For example, consider a system with an emergency break. Since the break is supposed to stop the system when the button is pressed, the behaviour is reactive. If the requirements state that the system should stop within two seconds after the button is pressed, the emergency break would be classified as a real-time system.

2.1 Strictness and criticality

We distinguish between hard and soft real-time systems, as well as between critical and non-critical ones.

In a *hard real-time system*, timeliness requirements are strict, and a result delivered too late (or too early) is considered incorrect. The system must meet all temporal requirements in order to be correct. In *soft real-time systems*, the value of a result decreases if the timeliness restrictions are violated, but a late result might be better than no result at all. Typically, the system requirements define a minimal *quality of service*, i.e. that at most one deadline out of one hundred can be missed.

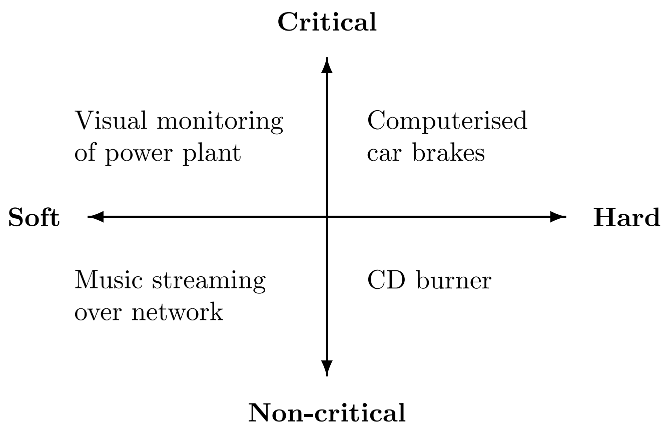

Critical
Visual monitoring of power plant
Computerised car brakes
Soft
Hard
Music streaming over network
CD burner
Non-critical

Figure 1: The orthogonality of criticality and strictness

The criticality of a system is based on the consequences of failures. In a *critical system*, the violation of a timeliness requirements could cause serious damage to people or equipment, or result in a significant loss of money. *Non-critical systems* might be useless if deadlines are missed, but it does not cause any significant harm.

The classic view has been to incorporate criticality in the concept of hard real-time systems, but it is often argued that the two concepts are in fact orthogonal. Figure 1 gives some examples of typical systems from the four categories.

2.2 Event triggered or time triggered systems

Real-time systems are often categorised as being either event- or time triggered. In a *time triggered system*, activities are performed at certain points in time, often in a periodic fashion. Such a system can not respond directly to external events when they occur, but it can still respond to events by scanning the environment at certain points in time. This style is naturally suited for implementing periodic activities such as control loops.

In an *event triggered system*, the execution is guided by the non-deterministic occurrences of asynchronous external and internal events. This provides a higher degree of flexibility, but also makes the system harder to analyse with respect to timeliness criteria. The event triggered approach suits the implementation of systems with reactive behaviour.

As an example of the differences, consider a system where it must be guaranteed that if a certain catastrophic event (such as the temperature reaching a certain level) occurs, some emergency action is carried out within one millisecond. To ensure this, a time triggered system would have to sample the temperature more than once every millisecond, which is quite inefficient if the event does not occur often.

An event triggered system, on the other hand, can perform other duties without wasting time checking the temperature, as long as the catastrophic event does not occur. This is usually more efficient than the time triggered method, but makes it more complicated to ensure that no matter what happens, the emergency action will always be carried out in time.

Providing a complete definition of the difference between the two types of systems is not trivial. A time triggered system can be viewed as an event triggered system with one single event (the clock tick). Similarly, event triggered systems are normally implemented in hardware that is inherently time driven. Rather, the time- and event triggered approaches should be viewed as two conceptual abstractions focusing on different aspects of real-time systems.

We can also think of them in terms of services provided by some low-level layer of the system (such as an operating system) to higher-level parts. A system that provides functionality to react asynchronously to external events, would be classified as event triggered, while one that only permits timed constructs like timeouts, etc. would be considered time triggered.

2.3 Offline and online scheduling

If a system is to meet certain timeliness constraints, and especially in order to prove that it always will, the activities of the system must be planned in a rational way. The activities are usually split into smaller units called tasks, and planning the order in which these tasks are to be executed is referred to as *scheduling*.

Scheduling can be performed during the design of the system, called *offline scheduling*. When the tasks are defined, a schedule is created that contains information about when the tasks are to be executed. Once a schedule is created, it is included in the system, and the runtime decision of which task to execute is simply a matter of looking into the schedule. The other approach, denoted *online scheduling*, is to include in the system an algorithm for making scheduling decisions at runtime.

Scheduling tasks in an optimal way with respect to timeliness requirements is in general an NP-hard problem. Offline scheduling has the advantage that a lot of effort can be put into creating a suitable schedule. Any scheduling algorithm can be used, and a wide variety of timeliness restrictions can be taking in consideration. Since online scheduling is done at runtime, considerably less effort can be allowed, and thus the types of timeliness constraints allowed in the system are often restricted.

Systems scheduled offline are easier to verify, since one need not consider the way the schedule was created, only the schedule itself. In online scheduled systems the scheduling algorithm must be proven never to violate the timeliness constraints of the tasks.

The main advantage of online scheduling is its flexibility. Such systems can make scheduling decisions based on information that is not known at design time, like the arrival times of external events. If an offline scheduled system is to respond to external events, the scheduler must reserve time for this in the schedule, without knowing when it will actually be needed.

There is also the possibility of combining the two approaches. An offline schedule is constructed for a subset of the system, usually the safety critical parts. The runtime mechanism executes tasks according to the schedule, but schedules additional tasks when the system is idle. More sophisticated algorithms allows the original offline schedule to be altered by the online scheduler, to enable more efficient scheduling, as long as the timeliness constraints are still met.

For an introduction to real-time scheduling, including descriptions of common scheduling algorithms, see [But97].

2.4 Timeliness constraints

In a real-time system, each task is characterized by a set of parameters. The parameters vary between different types of systems, for example time- and event triggered, but a few common ones are explained below.

Computation time The time needed to execute the whole task without interruption. Often, only a worst-case execution time (WCET) is given, rather than the exact computation time.

Deadline The time at which the execution of the task should be complete in order to be (fully) useful to system.

Period The time between two consecutive instances of a periodic task.

Priority Priorities are used to guide the allocation of resources (including processor time) among tasks. Typically, one wants to guarantee that a task is never delayed because of another task of lower priority.

Value In some systems, tasks are given values to denote their importance to the system. The concept is similar to that of priority, but values are typically interpreted in a cumulative way, meaning that the system prefers a set of low-valued task over a single task of slightly higher value.

In many applications, tasks are not allowed to execute in arbitrary order, but required to meet certain *precedence restrictions*. These restrictions may originate from task communication, or directly from the environment of the system. The precedence constraints are often represented by a directed acyclic graph, defining both direct and indirect precedence relations.

The most common timeliness constraint is that of a task finishing its execution before its deadline, but many other types of constraints can be of use in certain applications. For example, it is sometimes necessary to restrict the deviation of the finishing time among the instances of an aperiodic task, called jitter.

Sometimes, it is also necessary to express timeliness assumptions on the environment. A particular scheduling policy might guarantee, for example, to avoid deadline violations provided that the time between two consecutive arrivals of an aperiodic task is never less than 10ms.

Correctness of reactive systems are often expressed in terms of *safety* and *liveness* properties. Safety properties state that the system never does something it should not do, such as delivering a faulty response. These properties are important, but not enough, since they are all trivially met by a system that simply does nothing. Liveness properties guarantee the progress of the system, for example that no event is left unattended.

3 Specification

In order to effectively reason about complex systems, many details must be omitted. In some cases because they have not been decided yet, or because they do not contribute to the particular aspect that one currently wants to focus upon. A specification captures essential properties of software behaviour, but depending on what purpose we have for specifying, these properties can vary.

From a software engineering point of view, a specification can be defined as *a very precise description of possible effects of a software component* [vV93]. This definition gives a lot of room for interpretation, but it provides some general intuition. It limits the aspects to be described to those regarding the *effects* of the software. Further, it allows incomplete specifications, not describing all possible effects.

When comparing specification methods, it should be kept in mind that different methods give the term different meanings. Definitions range from fully formal ones to informal descriptions of the intended usage.

3.1 Reasons for specifying

As mentioned above, specifications are used in many different ways. Naturally, the purpose of the specification has a big impact on the appropriateness of a specific method. Some methods are focused on non-formal concerns such as communicating the intuition behind design or requirement decisions. Other methods aim at fully formal descriptions of safety critical components.

Common reasons for specifying include the following:

- A specification can act as an outline of a system during the design phase. It serves as a common base for discussing requirement and design issues with customer, management and developers. Using natural language for this can prove too ambiguous, and providing a simple, well defined, notation decreases the risk of misunderstandings.

- During the design of a system, a specification can be used to clarify ideas and concepts. It can aid in finding ambiguities and incompleteness in the intended system at an early step.

- Specifications can act as contracts between the developer and a customer. Upon delivery, the system can be compared to the specification to validate that the customer gets what was paid for. The specification can be produced by the customer ("We want you to build a system that meets these requirements."), or by the developer ("If what we build corresponds to this, we have met your requirements.").
- For safety critical systems, one might be required to formally prove properties such as safety and liveness. One way of doing this is to prove that the specification has the desired properties, and that the system correctly implements the specification.
- Executable specifications can be used to gather non-formal information of a system, such as average load of a component or how the system behaves under temporary overload, before an implementation of the system exists.
- A specification can be viewed as a first step of an implementation. Transformational development, perhaps using a refinement calculus, can transform a specification through a series of steps into an implementation.
- As well as for development of new software, specifications provide a good starting point for software maintenance.

From a theoretical point of view, specifications are often thought of as a way to rule out the possibility of errors in a system. Thus, a specification method must be able to produce formal proofs of correctness.

From a practical point of view, one may consider the goal of specifications as increasing the quality of the produced systems. In this case, proofs are viewed as a cost-effective way of uncovering errors.

3.2 Desirable properties of specification methods

Depending on the intended use of the specifications, different requirements are implied on the specification method. A number of general requirements or desirable properties can be identified:

Simple It must be possible to create a specification with reasonable effort. The more details that can be omitted, the more efficient.

Readable The specification should be easy to understand given some training. It should be possible to grasp the major structure of the specified software without having to understand the details. Preferably, this is true not only for the specifiers, but for customers and non-technical members of a developer team as well.

Wide-range Most methods aim at providing good support for a certain type of software components. While making it easy to write specifications for such systems, this means that concepts outside the intended domain can be difficult to specify and result in complicated specifications. In general, a method should aim at as wide a range of applications as possible.

Continuous A small conceptual change to the problem domain should require only small changes in the specification.

Compositional Specifying large, complex systems requires that the system can be divided into parts that can be specified individually. Composing these specifications to obtain a specification of the complete system should be as easy as possible, and not require changing the partial specifications.

Unambiguous In order to ensure that the specifications are unambiguous, the method must contain a formal definition of the semantics of specifications. A danger of using informal or semi-formal methods is that they tend to hide ambiguities and misunderstandings behind a seemingly unambiguous notation.

Useful Specifications are of limited use on their own, and to benefit from the method there must exist simple methods to formally reason about the resulting specifications. Examples of reasoning include proving properties such as liveness and safety, and checking whether two specifications are consistent.

In addition to these, specifications are sometimes required to be *executable*. An executable specification technique has a constructive semantics, and thus provides a way to evaluate specifications. Such specifications can be treated as prototypes, used to get early feedback during the design phase. Early testing can be a cost-effective way of finding design flaws compared to constructing formal proofs [Kem85, DK94].

3.3 Classification of specification methods

Specification techniques can be classified in many different ways, and this section presents two possibilities. In practice, many methods and tools cover aspects from more than one category, either by combining different more specific techniques, or as a single all-embracing one.

Descriptive and operational methods

One method for classifying is based on the extent to which a technique is descriptive or operational [BCN95, EPO99].

Descriptive methods, sometimes referred to as algebraic or axiomatic, specify properties of the system by means of equations or first (or higher) order logical formulas.

For example, a descriptive specification of a stack of integers might include the following axioms. The first one states that the combined effect of a *push* and a *pop* operation leaves the stack unchanged. The second axiom states that *top* returns the last element pushed onto the stack.

$$pop(push(st, e)) = st$$
$$top(push(st, e)) = e$$

Descriptive methods provide a straightforward way to reason about the specifications, since ordinary theorem provers can be used to prove different properties. The main difficulties related to these methods are poor readability of large, complex specifications, and cumbersome treatment of errors.

Operational methods build a model of the system from a set of basic objects such as numbers, tuples and sequences, and operations on these such as addition and concatenation. The semantics of an operation is then specified by its effects on the model. Since these methods essentially define states and transitions, they naturally produce executable specifications.

An operational specification of the integer stack might for example model it by a sequence of integers. It would also contain definitions of the operations in terms of well-defined operations on sequences (e.g. *cons* and *first*), such as the following.

$$\begin{array}{ll} \mathit{pop}(st): & st' = \mathit{rest}(st) \\ \mathit{push}(st, e): & st' = \mathit{cons}(e, st) \;\wedge\; e' = e \\ \mathit{top}(st, e): & e' = \mathit{first}(st) \;\wedge\; st' = st \end{array}$$

The postfix accent ($'$) is used to refer to the variable value after the operation is carried out.

One difficulty associated with these methods is that the intended data type might have properties that can not be specified easily using the basic objects and operations. In these situations, the specification can become very complex.

A typical example of operational methods is Z, described in Section 5.2.

Data, structural, process, and temporal formalisms

Methods can also be classified according to what aspect of the software they focus on [EPO99].

Data type formalisms describe the functional view of the system by specifying the data types of the system, and how data is transformed by the individual software components.

Structural formalisms focus on system decomposition into sub-systems. The decomposition can be hierarchical, in which case sub-systems are allowed to be further decomposed, or flat, permitting only one level of decomposition.

Process view formalisms specify the dynamic behaviour of the system, usually as a collection of processes or activities that (possibly through cooperation) carry out the desired scenarios. Such specifications can for example be used to ensure that deadlock, livelock or starvation situations do not occur.

Different methods use different means to specify the processes of the system, including algebraic descriptions, automata, Petri nets, and graph transformation. Subsequent sections describe some of these in more detail.

One important aspect of the process view is the way in which processes may communicate and synchronize. This can be done in essentially two ways: either by shared data or by message passing [BST89]. Shared data means that processes are allowed to share some variables, through which they can communicate. On an implementation level, this is fairly easy to accomplish, and thus this method is commonly used in real-time systems. Message passing allows processes to send and receive messages, possibly containing data. On a specification level, this has a number of advantages over shared data. It provides for less complex specifications, since issues like mutual exclusion can be omitted. Also, it is more suited for specifying loosely coupled systems.

Temporal view formalisms add to the process view some concept of time. Process formalisms usually introduce a simple concept of causality when defining for example the order of events in the system, but this is not enough to express elaborate temporal properties such as response time and deadlines.

The temporal information can either be included within the process view, or given as additional restrictions on the system. Also, the choice of temporal domain strongly influences both the expressiveness and the usefulness of the technique, as discussed in the next section.

3.4 Real-time aspects of specification

The significant aspect of specifying real-time systems compared to general systems is the need for explicitly stated temporal properties. Depending on the nature of the constructs used to specify data and behaviour, the temporal information will be incorporated in different ways, but more important when comparing different methods is the characteristics of their temporal domains.

Temporal domain structure

A *discrete domain* models time as a monotonically increasing sequence of integers. This structure is well suited for formal treatment, since the operations needed to manipulate specifications typically can be performed efficiently. Continuous or asynchronous behaviour can only be handled as approximations, by introducing a fixed granularity that limits the accuracy of observations.

Dense temporal domains use monotonically increasing real numbers rather than integers to model time, resulting in a higher accuracy. At the same time, the complexity of analysis operations like verification and identification increases.

Both of these structure types can be either *unbounded* or *bounded* in the future and in the past. A structure is bounded in the past if there exists a time instant such that no earlier instant exists, and bounded future is defined similarly.

When considering real-time systems, a very important distinction is whether the domain structure contains a *metric for time* [BMN00]. Without this, only temporal-order relationships between events can be described. The result is a purely causal temporal domain where the only temporal information available is precedence relationship between events in the system or its environment. Using this structure, it is not possible to refer to points in time when no events occur, or to give exact temporal measures for events and between events, such as duration, separation and timeout.

Semantic models

There are two important semantic models used for timed specifications, based on *linear time* and *branching time*, respectively. Linear time semantics interpret a specification as a set of linear structures of states, where every sequence represents a possible execution sequence of the system. Using this model, a system is completely determined by the set of observable partial runs.

With a branching time semantic model, a specification is interpreted as a tree structure of states. Each path in such a tree represents a possible execution sequence, similar to the sequences of the linear time model. The difference is that a system is determined not only by the possible executions, but also by the structure of the tree, i.e. the choices it has made during its execution.

Section 6 describes the difference between the two models for process algebras. Linear time semantics is represented by CSP traces, and branching time by state transition graphs in CCS.

Introducing time in the language

Temporal information can be included in the specification language in an *implicit* or *explicit* manner. Implicit time allows specifications to contain properties that vary over time, but properties may not refer to time explicitly. An example of such a property is "the door has been locked for two minutes". The truth of this property depends on the time it is evaluated, without explicitly referring to it.

A method with explicit time represents the current time by a specific variable. This allows specification to refer to absolute points in time, such as "the door is unlocked between time instants 5 and 9". Using explicit time, any useful real-time property can be expressed [BMN00].

In a compositional specification language, where specifications can be constructed by combining smaller specifications, the composition of temporal information must defined. In particular, it must be decided if the combined parts refer to a single *global clock*, or if each of them keep a *local clock*. If local clocks are used, the events of the whole system can only be partially ordered.

While the global clock approach provides a higher accuracy in terms of a total temporal ordering, it is sometimes considered an oversimplification to assume that the entire system can communicate synchronously. A possible trade-off is *self-timed* systems, where a subsystems with local time communicate asynchronously through a well-defined protocol that ensures a predictable event ordering in the system as a whole [SZ96].

4 Logics

Using logics to describe the behavior of systems allows for unambigous specifications that can be analysed using standard techniques such as model-checking and automatic theorem provers. If timeliness properties are to be expressed in addition to functional ones, the logic (propositional, first-order or higher-order) must be extended with some concept of time.

A first step towards such an extension is *modal logic*, where a logic formula has a set of interpretations (called worlds) rather than a single one. In each world, the formula is either true or false. A modal logic system is defined by the triple $\langle W, V, R\rangle$, where W is a set of worlds, V an evaluation function that assigns a truth value to every formula in every world, and R a relation defining the possible transitions between worlds. Thus $V(f, w)$ denotes the truth value (*true* or *false*) of the formula f in the world w, and $w_1 R\, w_2$ denotes that w_2 is directly reachable from w_1.

Two new operations, $\mathbf{L}$ and $\mathbf{M}$, are added to the symbols and operators of classical logic. The formula $\mathbf{L}f$ is true in w iff f is true in every world directly reachable from w. For $\mathbf{M}f$, it is enough if f is true in at least one of the worlds directly reachable from w.

Modal, and temporal, logics are often used in combination with modelling techniques such as timed automata or a process algebra, to express properties of models. In such combinations, the worlds, evaluation function, and transition relation are defined by the given model, and concepts like states, variables and signals can be used when creating the logic formulas. It is possible to ask, for example, if a given automata satisfies the property that the value of x is less than 2 in all states directly reachable from the initial state ($\mathit{initial} \rightarrow \mathbf{L}(x < 2)$).

A major weakness of logics as a specification technique is that they suffer from bad readability when the complexity of the expressed properties grows. Thus, they are best suited for expressing and verifying relatively simple properties of complex systems.

4.1 Temporal logics

If we want to use modal logic to reason about time, we let the worlds correspond to all possible instants of a given temporal domain, and use the *precedence* relation, denoted $<$, as the reachability relationship. The precedence relation is usually transitive and irreflexive, thus being a strict partial ordering on the temporal domain. A modal logic of this type is called a *temporal logic.*

A good introduction to the subject, and a survey of existing temporal logics can be found in, for example, [BMN00] and [AH92].

Temporal operators

Like modal logic, temporal logics add new operators to those of classical logics in order to quantify over the temporal domain. Different systems use different sets of operations, but the following operations, or slight modifications thereof, usually included.

To reason about the future, the operators *always* and *eventually* are introduced, denoted $\Box$ and $\Diamond$ respectively. There are also corresponding operations for reasoning about the past, denoted $\boxminus$ and $\diamondminus$.

$$\begin{array}{lll}
V(\Box f, t) & \text{iff} & \forall\, t'(t < t' \Rightarrow V(f, t')) \\
V(\boxminus f, t) & \text{iff} & \forall\, t'(t' < t \Rightarrow V(f, t')) \\
V(\Diamond f, t) & \text{iff} & \neg\, V(\Box(\neg f), t) \\
V(\diamondminus f, t) & \text{iff} & \neg\, V(\boxminus(\neg f), t)
\end{array}$$

Another common operator is *until.* It is denoted $\mathcal{U}$, and is defined as follows.

$$V(f_1 \mathcal{U} f_2, t) \quad \text{iff} \quad \exists\, t'(t < t' \wedge V(f_2, t') \wedge \forall\, u(t < u < t' \Rightarrow V(f_1, u)))$$

Informally, $f_1 \mathcal{U} f_2$ is true if f_2 is eventually true, and f_1 is true until then.

These, and similar, operators can be used to express statements about systems that change over time, such as precedence relationships and reactive behavior. Still, they are not expressive enough to specify real-time timeliness constraints such as deadlines and explicit duration of events. To make this possible, the temporal domain must have a metric for time (see Section 3.4), often introduced in the temporal logic by bounded operators.

The bounded versions of $\Box$ and $\Diamond$ are denoted by a subscript interval, for example $\Box_{[1,2)}$. Following standard mathematical notations, square and round brackets denote the inclusion or exclusion of that end point, respectivly. The formula $\Box_i f$ means that f is true in all time instants in the interval i. Similarly, $\Diamond_i f$ means that f is true in at least one time instant in i. For both constructs, intervals are interpreted relatively to the time of evaluation. For example, $\Diamond_{[1,4]}(\Box_{[0,1]} f)$ is true at time 2 if f is true for a whole interval of length 1, starting at some point between 3 and 6.

Specifications of real-time systems

The basic temporal operations can be used to describe properties related to reactivity and precedence, and the bounded operators allow many important timeliness constraints to be expressed. A few examples are given below.

- The actuators are only activated (represented by the predicate symbol a) after the sensors have been read (s), and never in emergency mode (e): $a \Rightarrow (\diamondminus s \wedge \neg e)$.
- Whenever the button is pressed (b), the door must be unlocked (d) within t time units: $b \Rightarrow (\Diamond_{[0,t]} d)$.
- The time between two consecutive message arrivals (m) is between t and t' time units: $m \Rightarrow (\Box_{(0,t)} \neg m) \wedge (\Diamond_{[t,t']} m)$.

When comparing different temporal logics, the most significant distinctions are the order of the classical logic it is built upon and the choice of temporal domain. A system based on predicate logic and a discrete domain is less expressive than a higher order logic with dense temporal domain, but is simpler to deal with when proving properties or checking satisfiability.

Temporal logics are not typically used as a complete specification method, but as a part of more elaborate methods. For example, the logic used for invariants in Z (described in Section 5.2) can be extended with temporal operations to facilitate reasoning about system liveness [DS89].

4.2 Duration calculus

The duration calculus [Cha99] is an extended temporal logic over intervals, developed to suit the modelling of real-time systems. It is based on finite continuous intervals and functions from time instants to the boolean values 0 and 1. The ordinary temporal logic is extended with operators to access subintervals.

Basic concepts

In duration calculus, a *system state* represents a logical property of the system, modelled by a boolean function over time. Simple states can be combined by boolean operators to achive more complex states.

The key concept, the *duration* of a state S in an interval, is denoted $\int S$ and defined as the integral of S over the interval. Thus, the meaning of $\int S$ is for how long the system state S is true in an interval. A couple of abbreviations are used to denote common durations, such as $\ell = \int 1$ representing interval length.

Formulas formed from durations, constants, global variables and functions are used to express properties of the system. Formulas can also be combined to create new formulas, using propositional connectives or the *chop* operator, denoted $\frown$. If F_1 and F_2 are formulas, $F_1 \frown F_2$ is a formula that is true for an interval i if the interval can be split into two sub intervals i_1 and i_2 such that F_1 is true for i_1 and F_2 is true for i_2. Operators corresponding to $\Box_i$ and $\Diamond_i$ of interval temporal logics, denoted $\Box$ and $\Diamond$ since intervals are implicit, can be defined in terms of chop.

$$\begin{aligned} \Diamond F &\overset{def}{=} \mathit{true} \frown F \frown \mathit{true} \\ \Box F &\overset{def}{=} \neg \Diamond \neg F \end{aligned}$$

Specifications of real-time systems

As an example of how duration calculus can be used to specify real-time systems, consider a part of the specification of a gas burner [OS95]. Let *Gas* and *Flame* represent the state of the gas valve and the flame, respectively. Combining these states, the undesired state of leakage can be formulated.

$$\mathit{Leak} = \mathit{Gas} \wedge \neg \mathit{Flame}$$

The formula *Lowleak* expresses the safety property that leakage is limited to at most 4 time units if the interval is shorter than 30 time units. Finally, *Safe* states that the safety property must be true at any time (i.e. for any interval).

$$\begin{array}{lll} \mathit{Lowleak} & = & \ell < 30 \Rightarrow \int \mathit{Leak} \leq 4 \\ \mathit{Safe} & = & \Box \mathit{Lowleak} \end{array}$$

4.3 Real time logic

A different approach to dealing with time is taken in Real time logic (RTL). Instead of considering time dependent formulas that are evaluated w.r.t. a point in time, RTL is extends an ordinary predicate logic with a special predicate which relates the events of a system with the times they occur. Compared to propositional linear temporal logic, RTL is strictly more expressive [JMS88].

RTL uses a discrete model of time, and formulas are built from mathematical relations ($<$, $\leq$, etc.), restricted algebraic expressions, existential and universal quantifiers, and the first-order connectives. In addition, the *occurrence relation* R is used to capture the notion of time, with $R(e, i, t)$ denoting that the ith occurrence of an event e occurs at time t.

Two classes of events are of particular interest. The start and end of a non-instantaneous action A is denoted $\uparrow A$ and $\downarrow A$, respectively. To keep track of system state, *transition events* are used. Let S be a predicate that asserts some property of the state, such as a variable having a certain value. Then (S:=*true*) and (S:=*false*) denote the events that occur when the predicate turns from false to true, and from true to false.

As an example of RTL, consider the following property, stating that the engine never runs for more than 20 time units:

$$\forall i \ \forall x, y \ (\ R(\uparrow \text{Engine}, i, x) \wedge R(\downarrow \text{Engine}, i, y) \ ) \rightarrow x + 20 \leq y$$

5 Automata- and model based languages

An automata oriented language focuses on the system state and how it changes over time. This can be captured in a straight-forward way by an automaton that explicitly defines all system states and the possible transitions between them. To make the system respond to its environment, transitions can be labeled whith triggering events and responses.

In a complex system, the number of states can be very large or infinite. A way to decrease the size of the automaton is to introduce state variables, allowing similar system states to be represented by a single state in the automaton. The exact state of the whole system is defined by the values of all state variables and the state of the automaton. To handle the variables, transitions or states can be labeled with constraints on variables, or with statements that update them.

As an example of the difference, consider the two equivalent automata in Figure 2, modelling the behaviour of a vending machine. The machine change states when a coin is inserted (represented by the event *coin*) or when the button is pressed (*button*). In the left automaton, an infinite number of states is needed to represent the number of coins currently in the machine, while the right automaton uses a state variable c for this purpose.

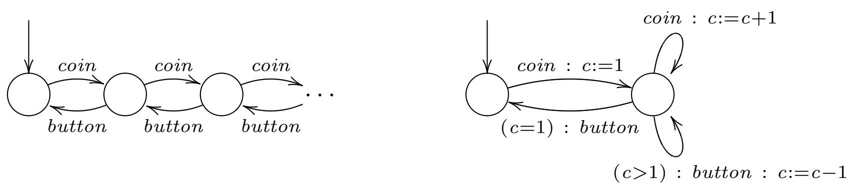


Figure 2: Using state variables to reduce automata size

These methods provide simple and mathematically clear ways of specifying behaviour of systems or components. When specifying large systems, however, they suffer from low simplicity and readability, at least in their basic forms.

Automata can also be used to represent the sematics of specification produced by some other method. The method provides a set of high level concepts and constructs for combining these into intuitive and readable specifications. Since the meaning of a specification can be represented by a (probably very complex) automata, it can be formally analysed or verified. Examples of techniques defined in terms of automata include process algebras and Petri nets.

Another solution to the complexity problem is used by model oriented methods like Z, presented later in this section, and VDM. The system state is split into smaller parts built from well-known mathematical entities such as sets and functions, and the specification defines operations on individual parts, or a small number of parts. Knowing how the full state of the system is divided into parts, and the effect of each operation on the corresponding parts, the behaviour of the whole system is defined.

5.1 Timed automata

The basic automata model is capable of specifying reactive behaviour, but in order to use it for real-time systems, some concepts of time must be introduced. A couple of different extensions have been suggested, using different notions of time (discrete, dense, etc.) and different means of specifying timeliness constrains.

A simple example is the *timed transition system* presented in [HMP91]. It uses a discrete time model with a single, external clock initially set to 0. The transitions are labeled with a minimal and a maximal delay, and the automata can be analysed w.r.t. bounds on the time between stimulus and response, formulated in temporal logic.

This method can be modified by making the clock variable explicit, allowing the constraints to refer directly to the time at each state. As a result, liveness properties can be reformulated into ordinary safety properties, and thus verified using standard timeless rules.

The theory of *timed automata* [AD94] is another, more elaborate, method. It uses dense time (i.e. time is modelled by real numbers) and multiple, explicit clocks that can be reset and read by the transitions. The example in Figure 3 shows a timed automaton with two clocks (x and y) that continuously accepts the event sequence a-b-c-d. The clock x is reset at every occurence of a, and the constraint $x < 1$ ensures that c occurs within 1 time unit of the preceding occurence of a. A similar restriction, using the independent clock y, is put on the occurences of b and d. Note that the usage of multiple clocks allows this type of interleaved restrictions without explicitly bounding the time between a and b.

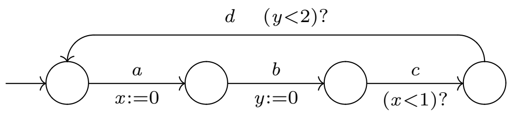


Figure 3: Example of a timed automaton

5.2 Z

The formal specification notation Z was developed by the Programming Research Group (PRG) at the Oxford University Computing Laboratory (OUCL). It is built around the concept of *schemas* that each define a small piece of the system. A schema can capture static aspects, such as possible states and invariants, as well as dynamic aspects like the effects of operations, relationships between input and output, and when state transitions are allowed to occur. The description in this section is mainly based on [Spi92], [WD96] and [Som96].

The VDM language is similar to Z in many aspects. For a tutorial of the differences between Z and VDM, see [HJN93].

Basic concepts

A schema contains the following information:

- The name of the schema.
- A *signature* part that defines the name and type of new state variables, and lists the included sub-schemas.
- A predicate that define relationships between entities in the signature and in the included sub-schemas.

The notation for defining a schema *Name*, with signature *Sig* and predicate *Pred*, is:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Name}} \\ \hline
\textit{Sig} \\ \hline
\textit{Pred} \\ \hline
\end{array}$$

There is also an equivalent horizontal notation, used mostly for simple schemas:

$$\textit{Name} \mathrel{\widehat{=}} [\, \textit{Sig} \mid \textit{Pred} \,]$$

The signature of a schema defines a number of typed state variables. The possible types range from simple types such as integers and user defined enumerable types, to compound types like sequences, tuples, sets and functions. Types can also be defined by more complex expressions containing lambda-expressions, function application, if-statements, etc. In addition to state variables, the signature also show included sub-schemas, as described later.

By convention, variables names ending with a question mark are used to indicate input to the system. Similarly, an exclamation mark is used for output variables.

The predicate of a schema describes an invariant, i.e. a statement that is true throughout the system lifetime. It is given in ordinary first order logic and may include equality and membership tests on expressions.

When defining an operation, the predicate can refer to two different versions of an entity (e.g. a state variable) N. The expression N refers to the value of the variable before the operation is carried out, and N' refers to the value after. Thus, the predicate can define pre- and postconditions as well as invariants. For example, the predicate in the following schema states that provided that a is a multiple of two (precondition), the result of the operation is that a is halved (postcondition).

$$\textit{Half} \mathrel{\widehat{=}} [\, a : \mathbb{N} \mid a \bmod 2 = 0 \land a' = a/2 \,]$$

Combining schemas

Schemas can be combined in a number of ways to create complex specifications. As mentioned above, one way of combining schemas is by *inclusion*, where a reference to a schema S is included in the signature of another schema T. This corresponds to the merging of the definition of S into that of T. As a result, the predicate of T may refer to entities in S, or of S itself.

In addition to the simple version, there are two kinds of restricted inclusion. The expression ΔS includes the entities and the predicate of S in two versions, the ordinary one and one where every entity in N is changed into N'. Two corresponding versions of the predicate is also included, thus preserving any invariant. This simplifies the definition of operations on complex schemas. For example, the schemas in Figure 4 define a limited counter data type and an operation on it. Note that there is no need to add the precondition $a < 9$ to the *Inc*-operation, since it is implicitly stated by $a' \leq 9$, one of the predicate versions included by $\Delta Counter$.

$$\begin{array}{|l} \hline \textit{Counter} \\ a : \mathbb{Z} \\ \hline a \leq 9 \\ \hline \end{array} \qquad \begin{array}{|l} \hline \textit{Inc} \\ \Delta \textit{Counter} \\ \hline a' = a + 1 \\ \hline \end{array}$$

Figure 4: Schema inclusion using Δ.

The second restricted inclusion, denoted ΞS, is similar to ΔS but with $N' = N$ added to the predicate for each entity N in S. As a result, the included schema becomes read-only.

Apart from inclusion, schemas can be combined by *conjunction* or *disjunction*. In both cases, the signatures are merged to create the new signature, and the predicates are combined by conjunction or disjunction, respectively. To permit this type of combination the signatures must be compatible, meaning that each variable name common to the to schemas must have the same type in both. Other logic connectives, such as implication and equivalence can also be used to combine compatible schemas, and negation creates a schema with the same signature but negated predicate.

A common use of schema disjunction is to combine a schema defining the normal behaviour of an operation with one that defines error handling. As an example, the schemas in Figure 5 define a more robust version of the increase operator, *RobustInc*, using *Counter* and *Inc* defined above. The boolean output variable *res*! is used to signal to the user if the operation was successful or not.

$$\begin{array}{|l} \hline \textit{Ok} \\ \textit{res}! : \mathbb{B} \\ \hline \textit{res}! = \text{true} \\ \hline \end{array} \qquad \begin{array}{|l} \hline \textit{IncErr} \\ \Xi \textit{Counter} \\ \textit{res}! : \mathbb{B} \\ \hline a = 9 \ \wedge \ \textit{res}! = \textit{false} \\ \hline \end{array}$$

$$\textit{RobustInc} \mathrel{\widehat=} (\textit{Inc} \wedge \textit{Ok}) \vee \textit{IncErr}$$

Figure 5: Combining schemas using disjunction and conjunction.

Z and real-time

If Z is to be used to specify real-time systems, a number of issues must be considered. Some notion of time must be introduced, eighter by allowing duration calculus or temporal logic [DS89] in the schema predicates, or by an explicit Z specification of time [BBR95]. Further, a computational model must be decided on that can handle nondeterminism and concurrency. Given these extensions, Z specifications can be analyzed with respect to safety and liveness.

6 Process algebras

The key concepts in a process algebra are *processes* (called *agents* in π-calculus), which represent the behaviour pattern of some object, and atomic *events* or *actions* initiated by the system or by its environment. Processes are defined by algebraic equations, and evolve into new processes by performing some action. Simple processes can be combined in a number of ways (sequentially, concurrently, using selection, etc.) to describe complex systems.

A process algebra usually consists of a set of operations, syntactic rules for how to combine these, a semantic mapping that assigns meaning to processes, a notion of equivalence and a set of algebraic laws.

Three examples of process algebras will be described briefly: CSP, CCS and π-calculus.

6.1 Communicating sequential processes (CSP)

CSP was developed by Hoare, and this section briefly presents the most important concepts [Hoa85].

Basic constructs

The simplest process is $STOP$, which simply terminates without engaging in any event. The expression $a \rightarrow P$ denotes a process that evolves into the process P after engaging in the event a. The deterministic selection construct $(a \rightarrow P \mid b \rightarrow Q)$ defines a process that either engages in the event a and then behaves like P, or engages in the event b and then behaves Q. The events a and b in a deterministic selection construct must be distinct.

Formally, each process is associated with an *alphabet* that defines the set of events it may engage in.

Processes are defined by equations, and infinite behaviours are accomplished by recursive, or mutually recursive, definitions. For example, consider the following processes:

$$\begin{aligned} C1 &= tick \rightarrow C1 \\ C2 &= (tick \rightarrow C2) \mid (break \rightarrow STOP) \end{aligned}$$

The process $C1$ engages in the event *tick*, and then behaves like $C1$. The result is a process engaged in an infinite number of *tick* events. The process $C2$ also engages in *tick* events (possibly an infinite number), but may at any time terminate after a *break* event.

Traces

A *trace* is a finite sequence of events engaged in by a process up to a certain time. It is in general impossible to know in advance the events that will occur when a process runs, since the behaviour of a process can depend on events initiated by the environment. However, it is possible to define *traces*(P), the possibly infinite set of all possible traces of the process P.

Reusing the examples above, we have:

$$\begin{array}{lcl} \mathit{traces}(C1) & = & \{\langle\rangle,\ \langle \mathit{tick}\rangle,\ \langle \mathit{tick},\mathit{tick}\rangle,\ \langle \mathit{tick},\mathit{tick},\mathit{tick}\rangle,\ \ldots\} \\ \mathit{traces}(C2) & = & \{\langle\rangle,\ \langle \mathit{tick}\rangle,\ \langle \mathit{break}\rangle,\ \langle \mathit{tick},\mathit{tick}\rangle,\ \langle \mathit{tick},\mathit{break}\rangle, \\ & & \ \langle \mathit{tick},\mathit{tick},\mathit{tick}\rangle,\ \langle \mathit{tick},\mathit{tick},\mathit{break}\rangle,\ \ldots\} \end{array}$$

Additional constructs

Processes can be combined concurrently by the expression $P \| Q$, that defines a process that engages in the same events as P and Q, and in the same order. The relative order of events from P and Q is arbitrary, resulting in an interleaved notion of time.

Concurrent processes communicate through common events. Events that are common to the alphabets of P and Q have to occur simultaneously in P and Q in the expression $P \| Q$. From the surrounding environment, the simultaneous events are visible as a single event. In the following example, $C3$ describes two clocks emitting events at a possibly different pace, but eventually breaking at the same time.

$$\begin{array}{lcl} C3 & = & C4 \,\|\, C5 \\ C4 & = & (\mathit{tick} \rightarrow C4) \mid (\mathit{break} \rightarrow \mathit{STOP}) \\ C5 & = & (\mathit{tock} \rightarrow C5) \mid (\mathit{break} \rightarrow \mathit{STOP}) \end{array}$$

$$\langle \mathit{tick},\mathit{tick},\mathit{tick},\mathit{tock},\mathit{break}\rangle \in \mathit{traces}(C3)$$
$$\langle \mathit{tick},\mathit{break},\mathit{tock}\rangle \notin \mathit{traces}(C3)$$

The same restriction makes $C6$, defined below, unable to engage in any event at all (we have $\mathit{traces}(C6) = \{\langle\rangle\}$).

$$C6 \quad = \quad (a \rightarrow b \rightarrow \mathit{STOP}) \,\|\, (b \rightarrow a \rightarrow \mathit{STOP})$$

The concealment operation P/E denotes a process that behaves just like P, but where all events in E are concealed from outside viewers.

When defining complex systems of concurrent communicating processes, it is sometimes useful to define communication over a set of channels. In CSP, the expression $c.v$ denotes the event of transmitting the value v over a channel c. This does not add to the expressiveness of the language, since $c.v$ is still treated in the same way as an ordinary event by the operations on processes. The notation simply provides the developer with a useful concept.

The deterministic selection operation requires that the two processes start with different events. There is also two non-deterministic selection operations where the choice is made arbitrarily. The process $P \sqcap Q$ behaves like P or Q arbitrarily. Also, the environment can not influence (or have any knowledge of) the choice. The process $P [\!] Q$ is similar in the respect that it behaves like P or Q, but the choice is not made fully arbitrarily. Instead, the choice is deterministic as long as it can be determined directly from the first event.

$$\begin{array}{lcll} (a \to P)[\!](b \to Q) & \overset{def}{=} & (a \to P) \mid (b \to Q) & \quad \text{when } (a \neq b) \\ (a \to P)[\!](a \to Q) & \overset{def}{=} & a \to (P \sqcap Q) & \end{array}$$

Failure sets

When non-deterministic operations are included in the language, processes are no longer described properly by traces only. We define $\mathit{failures}(P)$ as the set of all pairs (s, X) of a trace s and a set of events X such that P might produce s and then refuse to engage in any of the events in X. In CSP, a process is considered to be fully represented by its failure set [vG97].

$$C7 = a \to b \to STOP$$

$$\mathit{failures}(C7) = \{(\langle\rangle, \{b\}),\ (\langle a \rangle, \{a\}),\ (\langle a, b \rangle, \{a, b\})\}$$

6.2 Calculus of communicating systems (CCS)

CCS, developed by Robin Milner [Mil85], is similar to CSP in many aspects. Although the syntax is different, the basic constructs are roughly equivalent. The main differences concern communication, hidden events and the semantic model.

Constructs

As already mentioned, the basic constructs of CCS are similar to those of CSP and the syntax corresponds as described in Figure 6. A minor difference is that the ordinary selection operation in CCS allows the two combined processes to have the same first event.

In addition to the ordinary events, CCS allows the use of a hidden event τ which is invisible to the environment. This event can be used to introduce non-determinism corresponding to the $\sqcap$ operator of CSP. A process defined in CSP as $P \sqcap Q$ behaves in the same way as the CCS process $(\tau P) + (\tau Q)$.

CCS	*CSP*
NIL	$STOP$
$a.P$	$a \rightarrow P$
$(a.P) + (b.Q)$	$(a \rightarrow P) \mid (b \rightarrow Q)$

Figure 6: The syntax of CCS and CSP compared.

Concurrent composition of processes is achieved by the $\mid$ operator. The meaning is broadly similar to that of the CSP operator $\|$, but with the following important distinctions [vG97]:

- Rather than communicating by a single common events, processes communicate by one process engaging in an event a and the other in the complementary event $\overline{a}$.
- In CSP the communication between two processes engaging in the event a is visible as the single event a from the outside. A CCS communication involving a and $\overline{a}$ corresponds to a single τ event, and is thus not visible from outside.
- In CSP the common events force communication between concurrent processes, but in CCS communication is optional. For example, the process $(a.NIL) \mid (\overline{a}.NIL)$ behaves like the process $(a.\overline{a}.NIL)+(\overline{a}.a.NIL)+(\tau.NIL)$. The events a and $\overline{a}$ can occur independently (in any order) or as a synchronization invisible outside the process.

State transition graphs

A CCS process can be represented by a directed graph where the nodes are labeled by process expressions and the edges are labeled with actions [Gup94]. From a node P there is an edge to Q labeled e if P can evolve into Q by engaging in the event e.

For example, the process $(a.NIL) \mid (\overline{a}.NIL)$ from the example above can be represented by the graph in Figure 7.

Traces, failure sets and transition-graphs provide different notions of identification of processes [vG97]. The weakest identification is that of trace semantics, stating that two processes P and Q are identical if $traces(P) = traces(Q)$. The failure semantics of CSP result in the somewhat stronger identification criteria $failures(P) = failures(Q)$.

The difference between these two concepts is shown by an example.

$$\begin{tikzcd}
 & \boxed{(a.\mathit{NIL}) \mid (\overline{a}.\mathit{NIL})} \arrow[dl, "a"'] \arrow[d, "\overline{a}"] \arrow[dd, bend left=60, "\tau"] \\
\boxed{\mathit{NIL} \mid (\overline{a}.\mathit{NIL})} \arrow[dr, "\overline{a}"'] & \boxed{(a.\mathit{NIL}) \mid \mathit{NIL}} \arrow[d, "a"] \\
 & \boxed{\mathit{NIL} \mid \mathit{NIL}}
\end{tikzcd}$$

Figure 7: State transition graph of $(a.\mathit{NIL}) \mid (\overline{a}.\mathit{NIL})$.

$$\begin{aligned}
P &= (a.b.\mathit{NIL}) + (a.a.\mathit{NIL}) \\
Q &= a.((b.\mathit{NIL}) + (a.\mathit{NIL}))
\end{aligned}$$

$$\begin{aligned}
\mathit{traces}(P) = \ \mathit{traces}(Q) &= \{\langle\rangle,\ \langle a\rangle,\ \langle a, a\rangle,\ \langle a, b\rangle\} \\
\mathit{failures}(P) &= \{(\langle\rangle, \{b\}),\ (\langle a\rangle, \{a\}),\ (\langle a\rangle, \{b\}),\ (\langle a, a\rangle, \{a, b\}), \\
&\quad (\langle a, b\rangle, \{a, b\})\} \\
\mathit{failures}(Q) &= \{(\langle\rangle, \{b\}),\ (\langle a\rangle, \{\}),\ (\langle a, b\rangle, \{a, b\})\}
\end{aligned}$$

Using trace semantics P and Q are considered identical, but they are different w.r.t. failure semantics.

An even stronger identification, called *strong congruence* is achieved by the CCS semantics of state transition graphs. Two processes are strongly congruent if their state transition graphs are identical, which is defined by the existence of a certain binary relation, a *bisimulation*, between them.

To compare strong congruence with failure semantics identification, consider the following CCS processes:

$$\begin{aligned}
P &= (a.a.b.\mathit{NIL}) + (a.a.c.\mathit{NIL}) \\
Q &= a.((a.b.\mathit{NIL}) + (a.c.\mathit{NIL}))
\end{aligned}$$

The visible behaviour of these processes, represented by traces and failure sets, are the same. Still, they are represented by different state transition graphs, as shown in Figure 8.

$$\begin{aligned}
\mathit{traces}(P) \ = \mathit{traces}(Q) \ &= \{\langle\rangle,\ \langle a\rangle,\ \langle a, a\rangle,\ \langle a, a, b\rangle,\ \langle a, a, c\rangle\} \\
\mathit{failures}(P) = \mathit{failures}(Q) &= \{(\langle\rangle, \{b, c\}),\ (\langle a\rangle, \{b, c\}),\ (\langle a, a\rangle, \{a, c\}), \\
&\quad (\langle a, a\rangle, \{a, b\}),\ (\langle a, a, b\rangle, \{a, b, c\}), \\
&\quad (\langle a, a, c\rangle, \{a, b, c\})\}
\end{aligned}$$

The reason why these processes are not strongly congruent is that the choice is made at different points in time. The difference between failure semantics

$$\begin{tikzcd}
& \boxed{(a.a.b.\mathit{NIL})+(a.a.c.\mathit{NIL})} \arrow[dl, "a"'] \arrow[dr, "a"] & \\
\boxed{a.b.\mathit{NIL}} \arrow[d, "a"] & & \boxed{a.c.\mathit{NIL}} \arrow[d, "a"] \\
\boxed{b.\mathit{NIL}} \arrow[d, "b"] & & \boxed{c.\mathit{NIL}} \arrow[d, "c"] \\
\boxed{\mathit{NIL}} & & \boxed{\mathit{NIL}}
\end{tikzcd}
\qquad
\begin{tikzcd}
& \boxed{a.((a.b.\mathit{NIL})+(a.c.\mathit{NIL}))} \arrow[d, "a"] & \\
& \boxed{(a.b.\mathit{NIL})+(a.c.\mathit{NIL})} \arrow[dl, "a"'] \arrow[dr, "a"] & \\
\boxed{b.\mathit{NIL}} \arrow[d, "b"] & & \boxed{c.\mathit{NIL}} \arrow[d, "c"] \\
\boxed{\mathit{NIL}} & & \boxed{\mathit{NIL}}
\end{tikzcd}$$

Figure 8: Processes with different state transition graphs, but identical visible behaviour.

and strong congruence is essentially similar to the difference between linear and branching time, discussed in Section 3.4.

6.3 π-Calculus

The π-calculus focuses on describing processes whose interconnections change during their lifetime [Par01, MPW92]. Like CCS, the semantics is based on state transition graphs.

Constructs

Most constructions (NIL, $P{+}Q$, $P \mid Q$ and $\tau.P$) are similar to those of CCS, but the communication operations are more elaborate. The expression $\overline{a}b.P$ denotes a process that sends b along the channel a and thereafter behaves like P. The complementary expression $a(x).Q$ denotes a process that receives a value over the channel a and then behaves like Q, where x acts as a placeholder for the received value. The placeholder can be thought of as a variable that will get its value from the input along a.

The restriction operation $(\nu x)P$ denotes a process that behaves like P, but where x is local to P and thus cannot be used (directly) for communication between P and its environment.

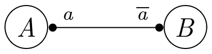
In addition to these, the calculus defines operations to compare values, corresponding to a simple if-statement in an ordinary programming language.

Communication

The most prominent feature of π-calculus is the fact that no distinction is made between names of values and names of channels. Hence, channels can also be sent between processes. As an example, consider the following definitions:

$$A = a(x).A' \qquad B = \overline{a}1.B' \qquad S = A \mid B$$

The process A expects to receive some value over the channel a. The value is placed in x and might guide the future behaviour (if it occurs in A'). In the system S, the value 1 is sent over a by B. This system can be described graphically as follows:



Now, we change the definition of B such that the channel a is sent to another process C (over a channel b). After receiving a, C can use it to send a value to A.

$$\begin{array}{lll} A = a(x).A' & B = \overline{b}a.B' & S = A \mid B \mid C \\ C = b(y).C' & C' = \overline{y}2.C'' & \end{array}$$

The corresponding graphical description of the system, and what it evolves into after one τ event (the result of a communication is the hidden event, just as in CCS), is shown in Figure 9. After this step, C' can send the value 2 to A over the channel a.

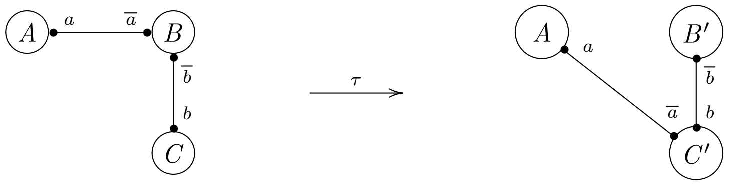


Figure 9: Sending a channel between processes.

Migrating local scopes

The graphical system descriptions in the previous section contain more information than the processes they are supposed to describe. There is nothing in the definition of the processes that specifies that C does not have access to the channel a also before it is sent by B. For example, the definition $C = \overline{a}2.C''$ would work just as fine.

By using the restriction operator, a name is made local to a process. If the restriction is put on the name of a channel, that channel becomes local and can not be used by processes outside the scope. In the case of the example, defining $S = (\nu a)(A \mid B) \mid C$ ensures that C does not refer to a at all.

Restriction ensures that names are not explicitly refered to outside the scope. Still, processes within the scope may send restricted names to processes outside, which then in turn can use them by refering to the corresponding placeholder. This is exactly what process C does in the previous section, by receiving a into the placeholder y, and then sending a value over y. Figure 10 shows how this is represented in state transition semantics.

$$\begin{tikzcd}
\boxed{(\nu a)(\,(a(x).A') \mid (\overline{b}a.B')\,) \;\mid\; (b(y).\overline{y}2.C'')} \arrow[d, "\tau"] \\
\boxed{(\nu a)(\;(a(x).A') \mid B' \mid (\overline{a}2.C'')\;)}
\end{tikzcd}$$

Figure 10: Migration of local scope.

After the transition, the scope of a includes the process $\overline{a}2.C''$ as well, and the placeholder y has been changed into a. For a formal treatment of the rules concerning migrating scopes, see for example [MPW92].

6.4 Process algebras and real-time specifications

Process algebras, together with an appropriate modal logic, provide a fairly easy way to formally model and reason about reactive, communicating systems. The basic ones, like CSP and CCS, have been extended in a number of ways to deal with related topics like scheduling analysis [Cle93] and shared resources.

Most process algebra use a simple causal notion of time where events occur in a ordered fashion, but with no explicit reference to the time of an event. While limiting the complexity of the method, this also reduces its expressiveness. As discussed in Section 3.4, an explicit notion of time is required to achieve a satisfactory specification of real-time behaviour.

TCCS is a process algebras with explicit time, and thus suitable for real-time systems [MT90]. It is based on CCS, but with two additional constructs. The term $(t).P$ denotes a process that behaves as P after exactly t units of time, and $\delta.P$ behaves as P but is willing to wait any amount of time before actually proceeding.

These constructs can be used to define useful real-time concepts such as the timeout construct $\delta.P + (t).Q$. This process can communicate with the environment by events in P, but if no event occurs within t time units, the timeout alternative Q is allowed.

In addition to time, process algebras can be extended with probability by introducing a new selection operator that assigns probabilities to the possible choices [Han94]. This allows for the verification of performance properties such as average response time, throughput, failure frequency, etc.

Process algebras have been especially successful for modeling and verifying communication protocols. For an introduction to the methodology, see [Par88].

7 Petri nets

Petri net methods are graphical notations with a solid mathematical foundation. They can be used in a number of ways to represent different aspects of computer systems, especially those concerning concurrency, distribution and non-determinism. The concept was invented by C. A. Petri in 1962, and has been proposed for a wide range of applications, including performance evaluation, communication protocols, and multiprocessor design.

Specification methods based on Petri nets are typically classified as operational methods, focusing on the process view. They provide a very flexible and expressive notation, while based on simple formal concepts that facilitate analysis. The most significant problem of Petri nets is related to complexity. The lack of a simple algebra of operations, that specifies how to combine and compose nets, leads to lack of modularity and thus makes large specifications difficult to understand [Gup94].

7.1 Basic Petri nets

Even in the basic form, Petri nets are very powerful and capable of describing a large number of different aspects of a system. The following description is based mainly on [Mur89].

Basic concepts

A Petri net is based on a weighted directed graph, with two types of nodes called *places* and *transitions*. The edges of the graph are restricted to be either from a place to a transition, or vice versa. Graphically, places are denoted by circles and transitions by boxes. A *marking* assigns a non-negative integer to each place of the graph, referred to as *tokens* and denoted by black dots. A Petri net is defined as a tuple of a graph and an *initial marking*.

To model dynamic behaviour, the marking of a net evolves in discrete steps, following very simple rules:

1. A transition is *enabled* when every place with an edge leading to the transition is marked with a number of tokens no less than the weight of that edge.

2. In each step, exactly one enabled transition is allowed to fire.

3. When a transition fires, each place with an edge leading to the transition looses a number of tokens equal to the weight of that edge. Also, each place with an edge leading from the transition gains a number of tokens equal to the weight of that edge.

Figure 11 exemplifies the rules.

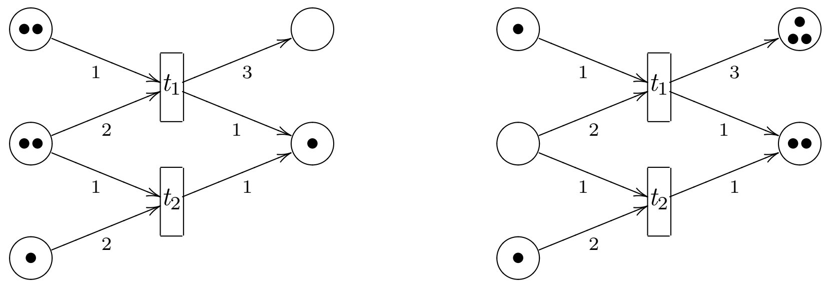


Figure 11: In the first net, t_1 is enabled and may fire, but t_2 is not. The second net shows the resulting marking if t_1 fires.

Interpretations and modelling possibilities

As mentioned above, Petri nets can be used in different ways to model different system aspects. For example, any finite state machine can be described as a Petri net. In this case, every transition have exactly one incoming and one outgoing edge, both with weight 1. The initial marking, and in fact all subsequent markings as well, contains exactly one token. The places represent states and the single token represents the current state.

The net in figure 12 models a finite version of the vending machine automata from Section 5. The three places correspond to the states where the machine has received 0, 1, or 2 coins, respectively. Transitions c_1 and c_2 represent the insertion of a coin, while b_1 and b_2 represent the button being pressed. To simplify the notation, unlabeled edges are assumed to have weight 1.

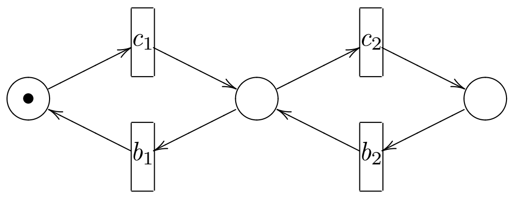


Figure 12: Representing automata with Petri nets.

State machines can model non-deterministic choice, represented in Petri nets by places with more than one outgoing edge. In order to model synchronization between parallel activities, however, another interpretation is needed. All edges still have weight 1, but transitions are allowed to have several in- and outgoing edges. In this setting, transitions represent activities or events, places represent the pre- and postconditions of activities, and a token denotes a satisfied condition.

As an example, consider a system with three activities, where two of them (a and b) can be carried out in parallel. However, none of them may start again before both are finished, which is ensured by the third activity (c). This system can be modeled by the net in Figure 13, assuming an interleaved concurrency.

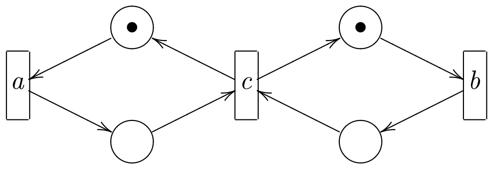


Figure 13: Modeling concurrency with Petri nets.

Allowing edges with weights greater than 1 permits representations of more elaborate synchronization behaviour. The net in Figure 14 describes a system of k processes that can read and write to a shared memory. During writing, no other process is allowed access to the memory, but any number of processes may read simultaneously. Tokens in p_w, p_r and p_p represent the number of processes currently writing, reading, and processing data, respectively. Tokens in p_m represent free access rights to the memory. In order to write to memory, a process must be able to claim all k access rights, but reading only requires one.

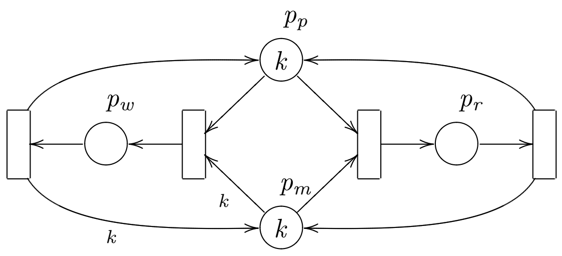


Figure 14: Modeling synchronization with Petri nets. Markings are denoted by numbers instead of dots.

Properties

The solid mathematical foundation of Petri nets makes it possible to apply a number of formal analysis methods to verify properties of a model. General properties such as reachability, liveness and safety have different meanings for different interpretations, but are defined independently of the interpretation. In addition, a particular interpretation can have its own specific analysis methods, useful only to models of that kind. Properties are classified as *structural* if they depend only on the structure of the graph, and *behavioural* if they depend on the initial marking as well.

For a given net, a marking M' is *reachable* from M if there exists a firing sequence that transforms M into M'. The set of markings reachable from the initial state of a net N is denoted $R(N)$.

A net N is *k-bounded* if no place is marked with more than k tokens in any marking in $R(N)$. As a special case, a net is said to be *safe* if it is 1-bounded. A net is *structurally bounded* if it is bounded for any initial marking. When Petri nets are used to model resources like processors or memory, boundedness analysis can guarantee that no behavioural pattern of may cause overflow.

A Petri net N is said to be *live* if for every marking in $R(N)$ it is possible to eventually fire any transition. The corresponding structural property guarantees the existence of a live initial marking. Liveness analysis is for example used to ensure the absence of deadlocks.

7.2 Modified Petri nets

Many extensions and modifications of the basic Petri net method has been proposed, often targeted at a specific usage such as real-time systems or performance evaluation.

High-level nets

The notion of high-level nets covers a number of modifications, including predicate/transition nets and coloured nets, that allows richer concepts of tokens and weighs than the basic method. As an example, consider the predicate/transition net in Figure 15.

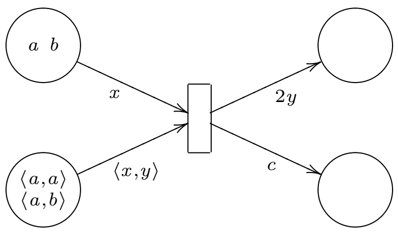


Figure 15: Example of a predicate/transition net.

The method allows atomic tokens of different types (a, b, …) as well as more complex structures like tuples. The edges are labeled with expressions, possibly containing variables, determining the number and types of tokens that can participate in a firing of the transition. A transition is enabled by a substitution s if the expression of every incoming edge, instantiated by s, is satisfied by the corresponding place. If a transition fires by the substitution s, the expressions of incoming and outgoing edges are instantiated by s to decide the amount and type of tokens to remove and add from each place.

For example, the transition in the net in Figure 15 is enabled by the substitutions $\{x/a,\ y/a\}$ and $\{x/a,\ y/b\}$. The resulting nets are shown in Figure 16.

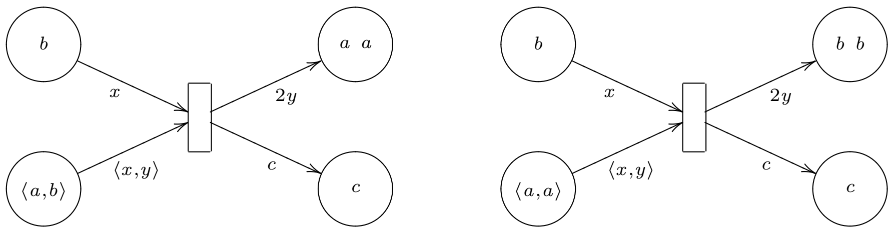


Figure 16: The results if the transition in the net in Figure 15 fires with substitution $\{x/a,\ y/a\}$ or $\{x/a,\ y/b\}$.

Timed nets

The most interesting Petri net extensions when specifying real-time systems are those that add an explicit notion of time. In its simplest version, this is done by assigning to each transition a *delay value* from a discrete or dense temporal domain. When a transition fires, the affected tokens are reserved, and can not be used to enable another transition, but the result of the firing occurs after the delay. A similar method assigns delays to places instead of transitions, restricting the time between the arrival of a token and the time at which it can participate in another transition.

If the delay values can be given as intervals rather than single points in time, more flexible specifications can be formulated. In most methods an enabled transition must fire within the given interval. This violates an important principle of Petri nets, namely that firing can be determined locally, since a transition t may be unable to fire because there exists another transition somewhere in the net whose interval forces it to fire before t.

A more general time extension, that also addresses the locality issue, is *time basic nets* [GMMP89]. Here, a timestamp is attached to each token, indicating its age. In addition, transitions are given time conditions that restrict the time when it may fire. When a transition fires, the new tokens that are created are timestamped with the time of the firing.

Time conditions of transitions are given as intervals whose end points are defined by expressions. The expressions may refer to the timestamps of tokens in the places with edges leading to the transition. For example, the net in Figure 17 describes a synchronization timeout behaviour where one process waits (represented by a token in p_1) at most D time units for another process to reach the synchronization phase (p_2). If the token stays in p_1 for more than D time units, the synchronization action t_s is no longer possible. At the same time, the timeout action t_t is enabled.

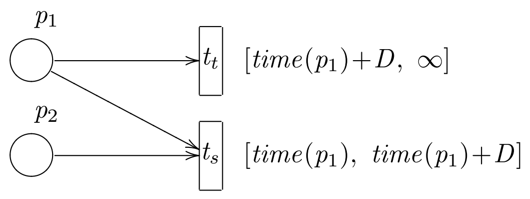


Figure 17: Timeout behaviour modeled by a time basic net.

Two different semantics are suggested for times basic nets. The *weak time semantics* states that if a transition fires, the firing time must be within the specified interval, but transitions are never forced to fire. In the timeout example above, this would mean that the waiting process is not forced to synchronize if the other process reaches the synchronization phase in time. In order to express that the timeout transition should only be allowed if the synchronization is not possible during the specified time, *strong time semantics* have to be used. It states that once a transition is enabled, it is forced to fire during the given interval.

Stochastic nets

Rather than providing nets with concrete temporal information, it could be given as stochastic distributions. In a *stochastic Petri net*, each transition t_i is associated with a random delay with an exponential probability density function of rate λ_i [Bal01].

A stochastic Petri net can be analysed with respect to, for example, the expected number of tokens in a place, the probability of a particular condition (such as a failure), or the mean number of firings of a transition per unit of time. This stochastic information is especially suited for performance evaluation.

8 UML

The recent interest in the object oriented programing paradigm is reflected in the development of similar methods and languages for analysis and design. From a large number of languages, all with their own strengths and weaknesses, the first draft of UML (Unified Modeling Language) emerged in 1995, and eventually became a de facto standard. In 1997, UML 1.0 was offered for standardization to the Object Management Group (OMG).

UML is a modeling language aimed at analysis, design, specification, communication and documentation of (primarily) software systems. When evaluating

UML as a specification language, it should be remembered that this is not the only domain it covers. For a substantial description of UML, see [BRJ99].

The language combines the structural-, process- and data type views presented in Section 3.3. Data types are modeled by class- and statechart diagrams, and the behaviour of the system, or parts of it, is captured in interaction diagrams. Structural aspects are considered in component- and deployment diagrams.

A UML model is built from a collection of graphical elements including classes, relationships, states, actors and diagrams. The syntax and semantics, defined by a UML meta-model, define how these elements can be combined, and to some extent the meaning of such combinations.

The fact that the semantics of UML is defined semi-formally restricts the possibility to compare and verify models. Recently, much effort has been put at developing formal semantics of UML. Some parts of UML are fairly easy to define formally, e.g. statechart diagrams, while other parts rely heavily on descriptions written in natural language. The main challenge is to provide a semantics that covers all aspects of the language, connecting the aspects of different types of diagrams. For a discussion of the issues that have to be dealt with when formalizing UML, see [vEB98].

Using UML for real-time systems

UML does not provide any means of defining explicit temporal constraints or properties, other than temporal ordering of events. The language allows constraints to be associated with any graphical element, but without further describing the syntax and semantics of these constraints. A number of extensions have been proposed that allows formal treatment of temporal information, for example by providing a well defined format for the constraints.

8.1 The Object Constraint Language

The Object Constraint Language (OCL), developed in 1995, is an expression language suited to define constraints on UML models [WK98]. It is declarative, meaning that it defines restrictions, e.g. on the possible values an attribute can take, rather than describing what to do if the restriction is violated. The language provides a number of built in types, ranging from basic ones like integers and strings, to collections such as sets and sequences. Further, types can be constructed from the entities of a UML model, e.g. classes.

Typed expressions are built from constants, UML entities, and operations defined for the types. Examples of operations are equality and disjunction of booleans, and addition of integers. Collection types typically have more complex operations, e.g. intersection and for-all-elements-in.

Expressions of type boolean are called constraints, and can be used in three different ways in a UML model. They can define *invariants* of classes, types or interfaces, that describes a property, e.g a relationship between attributes of a class, that must always hold. They can also be used as *preconditions* and *postconditions* of operations and methods, stating that whenever the postcondition holds when the method is called, the postcondition must hold once the method is finished. Finally, conditions can be used in transition diagrams as guards that define under what conditions a transition is allowed to occur. Additionally, expressions of any type can be used to identify a specific object, or set of objects, in actions or events of a transition diagram.

The syntax and semantics of OCL are fairly straightforward, but some consideration has to be made when fitting the OCL concepts with those of UML. One such UML aspect is inheritance, and thus OCL defines how constraints are inherited between objects in a UML model. Invariants and postconditions may be strengthened, but not weakened, by a subclass. Preconditions, on the other hand, may be weakened, but not strengthened, by a subclass. The reason for these rules are the substitution principle, that wherever an instance of a class is expected, it must be possible to use an instance of a subclass instead. This is a general principle of UML inheritance, and it must still be valid when OCL is used.

8.2 UML-RT

UML for Real-Time (UML-RT) extends the basic UML with constructs to facilitate the design of complex embedded real-time software systems [Sel98, Lyo98]. The constructs origin from the real-time specific modelling language ROOM [SGME92], and have been modified to fit UML.

The language focuses primarily on specifying the architecture of software systems, i.e. the major components, the externally visible properties of these, and the communication between them. It is argued that since decisions made during the architectural design have a very high impact on later design, it is also the phase that profits the most from a good modeling language.

Basic concepts

UML-RT adds four new building blocks to the standard UML meta-model. Three of them (*capsules*, *ports* and *connectors*) are used to model the structure of the system, and the fourth (*protocols*) models communication within the system. The behaviour of system components is modeled using standard UML statecharts, containing state variables and changes to these expressed in some programming language.

Capsules model complex software components that might be concurrent and physically distributed. The internal structure of the components is described by sub-capsules and the connections between these. A component interacts with the surroundings, and with its sub-capsules, through a set of ports which are the only parts of a component that are visible to other objects.

The *ports* can be connected either to the statechart diagram defining the functionality of the component, or to the port of a sub-capsule. This way, a

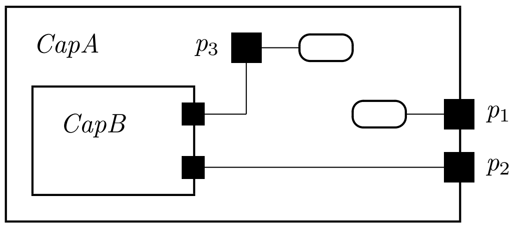


Figure 18: An example of UML-RT concepts.

message sent to a port can be handled directly by the capsule, or forwarded to a suitable sub-component.

Figure 18 shows an example of these concepts. The port p_1 of the capsule *CapA* is connected to the statechart, while p_2 is connected to the sub-capsule *CapB*. In addition, the capsule has an internal port p_3, i.e. one that is only visible within the capsule, connecting a port of the sub-capsule with the statechart.

A *protocol* defines a number of participating roles and the signals sent and received by each role. It can also contain a specification of the valid sequences of signals, encoded as a statechart. If no such specification is given, any sequence is considered valid.

Connectors are used to model communication channels between two or more ports. These ports must realize different roles of some mutual protocol. The protocols and connectors define the behaviour of the system on an architectural level.

Modeling real-time systems

Compared to standard UML, UML-RT provides some additional support when modeling the architecture of interactive systems. It does not, however, provide any support for modelling timing issues. It is possible to add structures to a UML-RT model that allows concepts like timeouts, but it is not supported by the language itself. Consequently, UML-RT does not facilitate reasoning about timeliness properties of the model.

In [GBSS98], UML-RT is provided with an alternative, formal semantics where the structural and behavioural parts of a UML-RT model are defined semantically in terms of flow-graphs. This allows a more formal treatment of models, including comparison and optimisation.

8.3 UML profile for schedulability, performance and time

In 1999, OMG issued a request for proposals regarding a new UML profile addressing specific problems related to the development of real-time systems. This resulted in a joint response from a group of OMG member companies, proposing a profile that covers for example timing concepts, analysis methods, scheduling and concurrency [OMG01]. The main aim of the profile is to define standard means to capture real-time modeling concepts, and in particular to allow exchange of models between different modeling tools, and between tools for modeling and analysis.

Since the profile should allow a wide variety of real-time systems to be modeled, it can not be restricted to a single temporal domain. On the other hand, it is also supposed to support essentially any type of analysis method, including many schedulability and performance analysis methods. These two requirements are somewhat contradictory, since analysis methods typically consider only a particular domain, thus making assumptions about the underlying models of time, concurrency, etc.

The problem is solved by defining an abstract notion of time and resources, that can be further instantiated to provide concrete notions like deadlines and timers. It also defines a minimal set of annotations for scheduling and performance analysis, that capture the common elements of different real-time specific methods, and can be extended with additional concepts needed by a particular analysis algorithm.

To include a new analysis method to the profile framework, the method provider defines the attributes that are essential for the method, and ties it to models of resources and time. When applying the method to a model, the developer might have to iteratively synthesize the model to transform it into an analysable format that contains all the information appropriate for the method. In order for the method to be useful, much of the synthesizing should be done automatically.

Modeling time

The profile distinguishes between two types of metric time, *physical* and *simulated*. Physical time is considered to be continuous, dense, unbounded and fully ordered, while simulated time models the timing concepts as visible from a system viewpoint. It can be discrete or dense, and possibly non-monotonic. Simulated time can be associated with physical time by means of periodic reference clocks that associate temporally close physical time instants with the same clock tick, at some given granularity.

To allow simulated time to be used in models, the profile contains definitions of *timers* and *clocks*. A timer generates a certain timeout event when a specified time instant is reached, while clocks periodically generate clock tick events. A clock or timer is always associated with a reference clock that provides the (simulated) time. They also have a number of attributes in common, such as resolution and drift.

Modeling schedulability

As mentioned above, the profile describes a set of common scheduling annotations that is sufficient to perform basic schedulability. It is expected that individual tool vendors provide specialized annotations to allow for more extensive analysis. The annotations defined by the profile include priority, absolute and relative deadline, and worst-case completion time.

An application is divided into *scheduling jobs* of some granularity that suits the scheduler and the system. A *schedule* contains the assignment of such jobs to execution entities (processors) and time intervals. The methodology used to create a schedule is referred to as a *scheduling policy*, and consists of an algorithm guided by a number of optimality criteria.

Analysing a model of an application with respect to schedulability requires knowledge of the scheduling policy and the resource allocation policy. It also requires an analysis method that fits these. If the method uses some specific annotations, a synthesizing method must be provided as well.

9 Real-time programming languages

When specification is performed as a part of a development process, it will typically act as a base for design and implementation later. This means that the usefulness of a specification language or method depends on how well its concepts correspond to constructs in the programming language that will be used. This section presents some of the features of programming languages that are required, or helpful, when implementing real-time systems [BW01, KS97].

Broadly speaking, there are two ways to implement real-time systems, with respect to the demands on programming languages. One possibility is to use any standard language combined with a real-time operating system. Each task is written as a separate program, and issues like communication and scheduling are handled completly by the operating system.

One problem with this type of programming is that the programmer has no, or little, control over the real-time related issues. The system must be designed in such a way that the temporal requirements are ensured for the given platform, for example by selecting certain periods and worst case execution times. Programs developed in this way typically suffer from bad portability since real-time issues are handled differently on different systems. Also, they tend to be difficult to maintain and extend, since platform dependent parameters are built into the design, and it is often hard to discover what, if any, original requirements that influenced them.

The tendency in recent years has been towards another way of developing real-time systems, where real-time related functionality is provided within the programming language. Examples of languages with real-time support include Ada95 [Bar95] and Real-Time Java [BGB$^+$00]. Standard languages, such as C, can be used in this way by extending them with a standardised interface to real-time primitives of the operating system, such as the one specified by the POSIX standard [IEE01].

The rest of this section focuses on the latter of these two alternative methods, i.e. languages that contain real-time related features. A comparative evaluation of such languages can be found in [HS90].

9.1 General issues

Many language and programming guidelines are especially important for real-time systems. This is partly because they are often safety-critical, but also because the additional complexity introduced by the temporal perspective stresses readability and simplicity demands.

These general issues include consistent and intuitive syntax and semantics. Further, the language should support encapsulation, e.g. through a package- or module concept, since it allows information hiding and abstract data types.

In a language with a strong type system, many minor programming mistakes can be discovered during compilation, resulting in more reliable software. Preferably, the type system should allow the programmer to define new types, both simple and structured.

9.2 Concurrency

A language suited for real-time applications should provide the programmer with explicit control over concurrency. This includes defining concurrent activities, but also dealing with synchronisation and communication between them.

If a languages requires all concurrent activities to be known at compile-time, it is said to support *static structure concurrency.* A more flexible possibility, used in for example Java, Ada and POSIX, is *dynamic structure concurrency* where new concurrent activities can be created during execution.

Using a broad definition, synchronisation means constraining the way the execution of actions from different activities may be interleaved. Sometimes, however, the term refers to the more specific task of bringing two processes simultaneously into predefined states. Synchronisation may be required because activities share common resources, such as memory or peripheral devices. Alternatively, synchronisation might be needed to ensure correct functionality when the relative order in which activities perform their actions affects the final result.

In a shared memory setting, constructs such as semaphores, critical regions and monitors can be used to achieve synchronisation and controlled communication between activities [SG98]. In Ada, semaphores can be implemented by means of message passing, and from them constructs like monitors can be constructed. A type of monitors, called protected objects, ensures mutual exclusion of critical sections. POSIX allows activities to share common memory, and provides semaphores and monitors (called mutexes). In Java, monitors are implemented as synchronised methods that can not be executed by more than one activity simultaneously.

The alternative communication style is based on messages sent and received by the activities. Message exchange can be *synchronous*, i.e. blocking the sender until the message has been received, or *asynchronous.* Further, activities may be allowed to *broadcast* messages to all other activities simultaneously, or restricted to one recipient per message. Ada uses a synchronous message style, and POSIX supports asynchronous message passing.

Concurrent languages

Real-time systems are often developed in general imperative, or possibly object-oriented, programming languages extended to permit concurrency. However, there exist languages especially designed to fit this type of applications.

Pure functional languages are inherently concurrent, since the absence of side effects ensures that expressions can be evaluated in any order. ERLANG is a concurrent functional language, developed by Ericsson to suit the implementation of large and highly dynamic systems such as telephone switches [AWV93].

Esterel is a synchronous language, i.e. built on a model where concurrent activities are able to exchange information and perform computation in zero time [BG92]. This conceptual assumption ensures that the system behaves as intended, independently of how concurrent activities are interleaved during execution. No implementation can of course satisfy the assumption, but as long as any input can be processed, and possible results being output, before any new input can occur, the system will still function correctly. Another example of a synchronous language is Lustre [HCRP91].

9.3 Real-time facilities

As discussed above, in some cases systems can be ensured to satisfy the given temporal constraints through carefully selected task parameters. However, the level of portability and reusability is increased if the language provides real-time related primitives. Also, a wider range of systems, with respect to the types of temporal constraints, can be developed.

Interfacing with time

In some systems, time is not only relevant in terms of temporal restrictions, but also directly affects the functional behaviour. For example, pressing a button twice may give different results depending on the amount of time between the two events. This requires that the program can access a clock, either using primitives in the language, or through device drivers for the internal or an external clock. Both Ada, POSIX and Java provides clock primitives.

Another useful facility is the possibility to delay an activity, either until a certain time in the future, or for a relative period of time. Explicitly delaying an activity, rather than simply making it do nothing for a period of time, allows the scheduler to distribute unused resources to other activities during the delay. Some languages also provide primitives for waking delayed activities prematurely.

A related concept is that of timeouts, i.e. restrictions on the time an activity is prepared to wait for communication. The term is also used to denote restrictions on the time allowed for execution of a certain part of an activity. Figure 19 shows an example of how timeouts can be used in Ada. The activity waits for either MessageA or MessageB, but if no message arrives within ten seconds, the timeout alternative is chosen. Alternatively, the timeout construct **delay until** could have been used to define an absolute timeout.

```
select
    accept MessageA(message data) do
        Perform some actions
    end MessageA;
or
    accept MessageB(message data) do
        Perform some other actions
    end MessageB;
or
    delay 10.0;
    Perform timeout actions
end select;
```

Figure 19: Message timeout in Ada.

```
select
    delay 1.0;
    Perform timeout actions
then abort
    Perform some actions
end select;
```

Figure 20: Action timeout in Ada.

As an example of timeouts on actions, consider the Ada code in Figure 20, that states that if the execution of the actions takes more than one second, it is aborted and the timeout actions are performed instead. As in the previous example, action timeouts can also be specified using absolute time.

Representing temporal properties

Most real-time languages, e.g. Ada and C with POSIX, do not allow temporal properties such as periods and deadlines to be explicitly defined. Rather, the intended behaviour is achieved using timeouts and delays. An example of a language that includes such constructs is Real-Time Euclid [KS86], where activities can be defined to restart with a certain periodicity, or in response to certain events.

In Real-Time Java, activities can be assigned scheduling parameters such as deadline and execution time. Through subclassing, more specific task models can be used to define for example periodic or event-triggered activities.

Satisfying temporal requirements

Many methods for scheduling and schedulability analysis is highly dependent on the execution times of the tasks. However, extracting this information from an implementation is very difficult for most languages. Another problem is that hardware structures like caches, pipelines and branch predictors, although improving the average performance, might lead to worse performance in the worst case scenario. Another consequence of such structures is that the execution time of a task depends highly on the other tasks it is interleaved with.

Since even a small underestimation of worst case execution time (wcet) can cause severe system degradation, wcet analysis is typically required to produce safe approximations, i.e. they may be higher than the actual value, but never lower. Still, in order to maximise the utilisation of resources, such as processor time, the approximations should be as tight as possible.

Programming languages can be designed to facilitate execution time analysis, for example by restricting jump and loop constructs, recursive function calls, and the use of pointers.

Scheduling

Some languages allow the programmer to influence the scheduling algorithm. The scheduling in Ada is priority based, and the programmer assigns priorities to the tasks. Further, the programmer can choose among different policies for resource locking and interrupt handling.

POSIX allows dynamically assigned priorities, and supports a number of different priority-based scheduling algorithms.

Similar features are provided by Real-Time Java, but in addition the language includes a mechanism for online feasibility analysis. When a schedulable object is created dynamically, it can be tested whether it can be admitted with respect to the current resources, or not. The object can also be assigned an appropriate priority.

10 Summary

The appropriateness of a specification method, or language, is strongly influenced by the intended purpose of the specification. The specification might be used to guide design and implementation, or for reasoning about some important properties of the system, or simply as a way to capture the intuition behind the system. A specification is typically an abstraction, focusing on some aspects of the system while ignoring other aspects to simplify reasoning. When specifying real-time systems it is often important to focus on aspects like concurrency, causal and temporal behaviour, and properties of liveness and safety.

Most specification methods suggested for real-time systems do not contain an elaborate notion of time in their basic form. Instead, they typically focus on concurrency and synchronisation, as process algebras and Petri nets, or functionality and structure, like UML and Z. In order to reason about temporal behaviour, these methods are extended with some temporal domain that is integrated with the concepts of the method. This can be done either with specific temporal operations and constructs, or by adopting a suitable temporal logic.

Today, most real-time systems are implemented in a standard imperative language, where most of the real-time related aspects are handled by the operating system. The developer achieves a higher degree of control if the programming language supports concurrency and direct access to time, e.g. through delay and timeout constructs. In addition, some languages provide functionality by which the programmer can influence scheduling and resource policies dynamically.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994. Fundamental Study.

[AH92] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 74–106, Berlin, Germany, June 1992. Springer.

[AWV93] J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[Bal01] Gianfranco Balbo. Introduction to stochastic petri nets. *Lecture Notes in Computer Science*, 2090:84–155, 2001.

[Bar95] John Barnes. *Programming in Ada'95.* Addison-Wesley, 1995.

[BBR95] J.-M. Bruel, A. Benzekri, and Y. Raymaud. Z and the specification of real-time systems. In H. Habrias, editor, *Z Twenty Years on – What is its Future?*, pages 77–91, Université de Nantes, France, 1995. IRIN (Institut de Recherche en Informatique de Nantes).

[BCN95] G. Bucci, M. Campanai, and P. Nesi. Tools for specifying real-time systems. *Real-Time Systems*, 8(2-3):117–172, 1995.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87 – 152, 1992.

[BGB$^+$00] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[BMN00] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Computing Surveys*, 32(1):12–42, March 2000.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[But97] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 1997.

[BW01] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages:*. Addison Wesley, 3rd edition, 2001.

[Cha99] Z. Chaochen. Duration calculus, a logical approach to real-time systems. *Lecture Notes in Computer Science*, 1548:1–7, 1999.

[Cle93] Rance Cleaveland. RTSL: A language for real-time schedulability analysis. In *Proceedings of the Real-Time Systems Symposium*, pages 274 – 283, Raleigh-Durham, North Carolina., December 1993.

[DK94] Jeffrey Douglas and Richard A. Kemmerer. Aslantest: A symbolic execution tool for testing aslan formal specifications. In *International Symposium on Software Testing and Analysis*, pages 15–27, 1994.

[DS89] Roger Duke and Graeme Smith. Temporal logic and Z specifications,. *Australian Computer Journal*, 21(2):62–66, 1989.

[EPO99] H. Ehrig, J. Padberg, and F. Orejas. From basic views and aspects to integration of specification formalisms. *Bulletin of the European Association for Theoretical Computer Science*, 69:98–108, October 1999. Columns: Formal Specification Column.

[GBSS98] R. Grosu, M. Broy, B. Selic, and Gh. Stefanescu. Towards a calculus for UML-RT specifications. *Seventh OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications, Vancouver, Canada, Monday, October, 19th, 1998.*, 1998.

[GMMP89] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezzè. A general way to put time into petri nets. In *ACM SIGSOFT Enginneering Notes; Proceedings of the Fifth International Workshop on Software Specification, 1989, Pittsburgh, Pennsylvania, USA*, pages 60–67, May 1989.

[Gup94] Vineet Gupta. *Chu Spaces: A Model of Concurrency*. PhD thesis, Stanford University, September 1994.

[Han94] H. Hansson. *Time and Probability in Formal Design of Distributed Systems*. Elsevier, 1994.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HJN93] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the differences between VDM and Z. Technical Report UMCS-93-8-1, University of Manchester, Computer Science Department, August 1993.

[HMP91] Tom Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for real-time systems. In ACM, editor, *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages, January 21–23, 1991, Orlando, FL*, pages 353–366, New York, NY, USA, 1991. ACM Press.

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

[HS90] W. Halang and A. Stoyenko. Comparative evaluation of high-level real-time programming languages. *Real-Time Systems Journal*, 2(3), 1990.

[IEE01] IEEE. *1003.1-2001 IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) – Rationale (Informative).* IEEE, New York, NY, USA, 2001.

[JMS88] Farnam Jahanian, Aloysius K. Mok, and Douglas A. Stuart. Formal specification of real-time systems. Technical Report CS-TR-88-25, University of Texas, Austin, June 1, 1988.

[Kem85] Richard A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, January 1985.

[KS86] E. Kligerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Trans. on Software Eng.*, 12(9):941–949, September 1986.

[KS97] C. M. Krishna and K. G. Shin. *Real-Time Systems.* McGraw-Hill, 1997.

[Lyo98] Andrew Lyons. UML for real-time overview, April 1998. RATIONAL Software Corporation Whitepaper, *http://www.rational.com/products/whitepapers/100463.jsp.*

[Mil85] Robin Milner. Lectures on a calculus for communicating systems. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197, pages 197–221, Berlin, 1985. Springer-Verlag. Lecture Notes in Computer Science Vol. 197.

[MPW92] R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[MT90] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415, Amsterdam, The Netherlands, 27–30August 1990. Springer-Verlag.

[Mur89] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[OMG01] OMG. Response to the OMG RFP for schedulability, performance, and time, June 2001. OMG document number: ad/ 2001-06-14, *http://www.omg.org/cgi-bin/doc?ad/2001-06-14*.

[OS95] E.-R. Olderog and M. Schenke. Design of real-time systems: The interface between Duration Calculus. In J. Desel, editor, *Structures in Concurrency Theory*, Workshops in Computing, pages 32–54. Springer-Verlag, 1995.

[Par88] Joachim Parrow. Verifying a CSMA/CD-protocol with CCS. In Sabnani Aggarwal, editor, *Proceedings of the 8th IFIP Symposium on Protocol Specification, Testing and Verification*, pages 373–384. North-Holland, June 1988.

[Par01] Joachim Parrow. An introduction to the pi-calculus. In Jan Bergstra, Alban Ponse, and Scott Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.

[Sel98] B. Selic. Using UML for modeling complex real-time systems. *Lecture Notes in Computer Science*, 1474:250–??, 1998.

[SG98] A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison Wesley, 5 edition, 1998.

[SGME92] Bran Selic, Garth Gullekson, Jim McGee, and Ian Engelberg. ROOM: An object-oriented methodology for developing real-time systems. In Gene Forte, Nazim H. Madhavji, and Hausi A. Müller, editors, *5th Int. Work. Computer-Aided Software Engineering*, pages 230–240, 6–10 July 1992.

[Som96] I. Sommerville. *Software Engineering.* Addison-Wesley, Reading, MA, 5 edition, 1996.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[SZ96] Arcot Sowmya and John Zic. State-transition based techniques for specifying real-time system: a review. In *Proceedings of the Australasian Conference on Parallel and Real-time Systems (PART'96)*, 1996.

[vEB98] Peter van Emde Boas. Formalizing UML: Mission impossible? In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[vG97] R. J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177(2):329–349, 15 May 1997.

[vV93] Hans van Vliet. *Software Engineering: Principles and Practice.* John Wiley & Sons, Chichester, 1993.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.