MASS
A Real-Time Activation Oriented Specification Language

Technical Report

— version 2 —

Vered Gafni

October 30, 1996

Contents

Chapter 1

Introduction

Real-time systems (RTS) are characterized by intricate behaviors that pose difficult modelling and specification problems. In recent years this issue attracted many research efforts mainly concerned with robust models of the temporal reactive behavior, expressive specification languages, and formal verification methods. This report presents an activation oriented approach for RTS specification, formalized in a language called MASS.

In this chapter we first present the notion of a real-time system, and explain the special problem of its representation. Then, we motivate the creation of MASS, and introduce the ideas underlying the language. We conclude with a survey of related work, and an outline of the report.

1.1 Real-time Systems

A real-time system is designed to control or monitor the state of an ongoing physical process (its *environment*). In general, it consists of computations that interact with the environment (and with each other) at rates, and under strict time-constraints, that are generally determined by the environment's dynamics. Hence, a real-time system must satisfy not only functional I/O relations, but a reactive behavior that relates events and computations along a time line, as well.

In general, the reactive behaviour required of a real time system is given in the form of *activation requirements*, each of which states an activating event that triggers the executions of a computation, and a (timed) responsiveness constraint bounding its termination. For instance, the following statements are simple examples of activation-requirements.

- Whenever a threat is detected send an alarm signal within 1 second.
- G_ψ, a control-loop transfer function, has to be executed periodically, every 0.1 seconds, and completed within 10 milliseconds from the start of each period.

In general, the activating event and the responsiveness constraint depend — often in a very intricate way — on the environment events, and/or events generated by executions

of (previously) activated computations. The resulting behavior of a real-time system that is driven by a number of activation requirements typically consists of merging and forking chains of reactions initiated by occurrences of environment events.

1.2 Formal Model of RTS

A real-time system is modelled by timed sequences (*runs*), each of which represents a possible behavior of the system. Specification are interpreted as subsets of runs that satisfy the causal and temporal relations implied by the activation requirements (*admissible runs*). Formally, a run is a sequence of pairs

$$\rho = (t_0, I_0), (t_1, I_1), \ldots$$

where $t_0, t_1, \ldots$ is a non-decreasing sequence of time elements (either points or intervals), and $I_i, i = 0, 1, \ldots$ are chunks of system dependent information (e.g. propositions, states, actions, events) related to the time elements t_i.

Concrete models are characterized by the sort of information they represent, and the corresponding model of time. The common approaches mainly debate on the following issues:

- The extent of system information representation (states that describe the internal structure, or just the externally observable events).
- The nature of actions' execution and scheduling (synchronous with true concurrency or interleaving vs. asynchronous).
- Continuous observation time domain (represented by the reals) vs. discrete sampling at a certain rate (time represented by integers).
- Global or multiple (synchronous or asynchronous) clocks used to time the system behavior.
- Strict or weak monotonic time progress (all models assert, however, the non-Zeno assumption).

In general, these issues are not independent. For instance, synchronous and discrete models use integer time; interleaving must make use of a weak monotonic time model; distributed systems need multiple clocks.

1.3 Specification

A specification of the reactive behavior is a description of the activation requirements in a formal language. Traditionally, systems are specified in a dual-language paradigm that consists of an abstract level *requirements* language, and a concrete level *design* language.

A requirements specification describes the reactive behavior in an abstract level of the temporal order of events and actions along a concrete time axis (e.g. the event α must occur within 5sec. after every occurrence of β). In general, requirements specification languages employ a logical formalism that consist of a propositional logic augmented with real time temporal operators (e.g. MTL[3], TPTL[2], RTTL[34], XCTL[25], ITL[32], Duration Calculus[15]), or a predicate logic with special time variables and functions (e.g. RTL[27]).[1]

At the design level, a specification operationally describes the construction of the reactive behavior. At this abstraction level, the language must be capable of referring to the execution durations and interaction timings of concrete computations (e.g. Each time the button is pressed, the average temperature is computed within 5sec.). Furthermore, some languages also include constructs describing implementation dependent primitives, such as priorities and execution speeds of CPUs. Most of the design languages employ a process algebra formalism e.g. wTCCS[38], Timed-CSP[35], ATP[33], ESTEREL[8], Signal [7] (few are based on a data flow paradigm e.g LUSTRE, TB-nets[18]). In general, such languages are executable, and some are supported by visual representation (Statecharts [24]).

As most RTS are large scale, it is desired that specification languages support gradual development by *refinement* and *composition* operators. Gradual development of a system inherently calls for a method of verification that allows one to assure that the correctness of a specification is preserved throughout refinement and composition. In contrast with informal testing methods, the capability of formal reasoning is a property of the language, and should be considered an essential part of its expressive power. In particular, one is concerned with the decidability of the language, and the existence of proof methods. In general, logical languages have powerful proof systems (decision procedures and deduction). In contrast, operational languages are executable and therefore amenable to simulation, but the associated proof systems (e.g. bi-simulation, reachability analysis) are less general comparing with logical systems.

In the dual language paradigm, formal reasoning extends beyond the scope of the language as verification must be carried out between specifications in different languages. The correctness of a design is verified either by transforming the operational specification into the logical language and employing a (deductive) proof system, or by direct application of model checking techniques.

A real-time specification language is primarily judged concerning its expressive power with respect to the reactive behavior representation. This includes the support of refinement and composition, and corresponding verification tools.

However, the usefulness of a language is determined by its *expressiveness*, in which term we mean the ease it is possible to specify the user intentions, or to understand them from a given specification. In general, expressiveness is improved by the succinctness of the language (minimal set of primitive entities and operators), the availability of structuring relations, and the employment of a declarative (or visual) represntation style.

[1] See [5] for a good survey of real time logics

1.4 MASS

In this report we present a real time specification language, called MASS, that adds to a large number of existing languages (as mentioned above). These languages are distinguished by the design decisions taken with respect to the issues raised above (in particular, primitives and structuring relations, execution environments, the communication paradigm, and the representation style.

The variety of approaches represented by these languages indicate a real problem regarding RTS specification. As stated in [24], "The problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and rigorous, sufficiently so to be amenable to detailed computerized simulation".

MASS should be considered in view of the ongoing efforts towards a better understanding and improved expressiveness of RTS specification languages.

1.4.1 Motivation

The specification approach suggested by MASS is intended to support a simplified, yet practical, formal representation of RTS. In this section we raise essential problems in existing languages. While there are languages that respond to some of the problems, MASS is motivated by the comprehensive solution it provides in a single framework.

Executable Specifications

The dual language approach raises conceptual and practical problems. At the conceptual level, there is a serious debate on what constitutes requirements or design, and the possibility to make a clear distinction in a concrete specification. At the practical level, this approach introduces a discontinuity in the system specification due to the different primitives and relations used in each domain. The synthesis of a design from requirements is carried out manually, and the verification process is impaired by the non-coherent methods required to prove that the design indeed satisfies the requirements. The effect of these problems is worsted due to the fact that in a normal development process the gap is traversed more than once.

In contrast, in MASS requirements and design are expressed by a single notation, but have different logical and operational (executable) interpretations. Consistency is guaranteed due to the (proved) fact that every execution gives rise to a model for the requirements. In MASS we show that this approach is feasible with a minimal set of primitives and relations, without sacrificing the expressive power, and expressiveness.

Reactive and Functional Behavior

Most of the existing specification languages (especially those based on a process algebra) employ a single framework to represent the reactive behavior and functional computations. A system is partitioned into processes (generally reflecting its functional structure) that

communicate through channels. Activation requirements are specified by 'signal-wait' pairs splitted over the two sides of channels in different processes.

Such a paradigm adversely affects verification and expressiveness since in order to infer causality and temporal relations it is necessary to dig into each process in search of communication constructs, identify the control path in which they might be reached, and then try to mate them over the set of processes constituting the system. (Yet, in applications that consist of a strictly ordered sequence of communications this paradigm provides the most natural representation).

The alternative of separating the reactive and functional behavior into different representations simplifies verification and increases expressiveness to a large extent. However, it must stil provide for the specification of the interaction between the different behaviours. In MASS we concentrate on the reactive behaviour specification. The desired interaction with the functional behaviour is provided in the semantics of the language, by a denotational relation between computations and the events they generate during executions.

Asynchronous vs. Synchronous Environments

Common process algebra languages assume an *asynchronous* execution environment. In this approach, concurrent processes execute independently, each progressing at its own pace. A process may be either computing, or waiting to communicate with another process (or the environment) on a certain channel.

The asynchronous model had been criticized for generating unpredictable temporal behavior because it varies with every possible scenario of events, and depends on the specific rates of the individual processes. An alternative execution model based on the *synchrony hypothesis* [8] has been suggested. In this model, processes are always ready to communicate with the environment as they respond to an external event (a set of signals) before the next one arrives. All processes share a uniform view of the present event. Output signals generated by computations become part of the input set, possibly triggering chains of reactions in parallel processes (infinite chains are prevented by proper static semantics rules). Thus, the synchrony hypothesis actually means that all the reactions to an event are executed instantaneously, in zero time.

In most of the synchronous implementations (Statecharts [24], LUSTRE [21], for instance) external events are considered only at the ticks generated by a master clock at a fixed period Δ (we call this variation Δ-synchronous).

In practice, languages based on the synchronous model are mainely intended for the specification of the reactive behavior of a system. Data driven functions are independently implemented in what appears to be an environment of the synchronous system. (The synchronous system emits signals that are used to activate the external computations, and sense the events generated by these computations as environment signals.) One should note, however, that even though the synchronous part yields a deterministic behaviour, the whole system, including the data processing environment, still behaves in a non-deterministic manner.

In the following paragraphs we examine asynchronous vs. synchronous environments regarding the representation of the response-time, and the ability to infer causality and time bounds among the occurrences of events.

Response Time The table below presents the response time of each model (i.e. the duration it takes from the occurrence of an event until the completion of the corresponding response) by three parameters:

- t_e — the delay from the occurrence of an event until it is actually handled.
- t_c — the time it takes to process (identify) the event.
- t_r — execution time of a computation activated in response to an event.

asyn'	syn'	Δsyn
$t_e \geq 0$	$t_e = 0$	$\Delta \geq t_e \geq 0$
$t_c > 0$	$t_c = 0$	
$t_r > 0$	$t_r = 0$	$\Delta \geq t_r \geq 0$

In the asynchronous model none of the parameters (t_e, t_c, t_r) is bounded. Thus the satisfaction of activation requirements cannot be controlled in this model.[2] Specifically, events are liable to be lost if the system is engaged too long with responses to previous occurrences, and there is no control whether executions exceed their deadlines (actually no deadlines are specified). Indeed, some languages (ADA, for instance) include special timeout and exception mechanisms to handle such cases, however these have no temporal semantics.

In the synchronous and Δ-synchronous models the response time is constant, zero and 2Δ respectively. This of course guarantees the satisfaction of activation requirements, however it is questionable to what extent the synchrony hypothesis is reasonable. In general, synchronous languages are compiled into finite state automata, and there are representation techniques that yield very efficient code. So, under the Δ-synchronous model many real-time systems are implementable (while introducing grave maintenance problems).

Temporal Relations Reasoning in real-time systems is mainly concerned with the causality and time bounds among the events constituting the reactive behavior.

In the asynchronous model causality is traceable. However, the complexity of the reasoning process is fairly high due to the unified representation of the functional and reactive behaviours (see Section 1.4.1). In contrast, temporal relations are not deducible in this model since there is no way to know the duration a process is suspended until a requested communication is established.

In a system that is entirely specified within the synchronous model both, causality and temporal relations, are deducible. Indeed, this was a major reason for the introduction of this model [8]. However, in systems that combine synchronous reactions with activations

[2] We do not consider pre-run scheduling approaches [41].

of asynchronous computations (see Section 1.4.1), the capability of tracing causality and computing durations between occurrences of events is impaired to a large extent. This is since events participating in a synchronous computation are not formally related with the asynchronous computations that generate them. For instance, a computation that is responsible for closing a gate (say in a railroad crossing) will be represented in a synchronous specification by the events *close* and *closed* that designate respectively the activation and termination of this computation. However, there is no formal way to infer that the event *closed* is necessarily the result of a previous occurrence of the event *close*.

1.4.2 The Framework of MASS

MASS is a language intended for the specification of the reactive behavior in real-time systems, both at requirements and design levels.

At the requirements level a real time system is represented by a set of primitive events, called *tasks*. Standard logical and special temporal operators enable the expression of compound scenarios of task occurrences. A specification in MASS at this level is a set of events of the form "if α then q within t time instants", where α is a general event expression, and q is a task.

At the design level, we associate every task with a concrete function, and interpret the termination of the function execution as an occurrence of the task. The events in the specification are interpreted at this level as "if α then execute q within t time instants".

The specification of the functions associated with the tasks is assumed to be given separately within a proper framework (a conventional programming language). Thus, MASS enables the representation of a real time system by separate specifications of the reactive and functional behaviors.[3] Note that the separation is complete in the sense that the reactive behavior does not rely on the content of the functions, and vice versa. However, the interaction between the separate specifications is well established by the time constrained 'event $\rightarrow$ function' activation relation given in the reactive part, and the inverse 'function $\rightarrow$ event' generation relation provided in the semantic level of the model. Hence, causality and time bounds are completely controlled along the operation of a system.

The Execution Environment

Operationally, we suggest an hybrid execution system where the reactive behavior is carried out through a Δ-synchronous model, and the computations are run concurrently in an asynchronous model. Every clock tick it is decided what computations should be activated and aborted in response to events occurring so far. The asynchronous executive senses the activation commands (although not necessarily instantaneously) and reacts by scheduling of the proper computations. Executing computations cannot communicate with the environment;

[3] MASS is acronym of 'Marionettes Activation Scheme Specification language', a metaphor suggesting the separation of the activation mechanism from the activated puppets.

only the termination of an execution (including abortion) is observable as an event by the activation mechanism.

This execution paradigm reflects an assumption that the computation of the required reactive behavior is short enough with respect to the operational rate in which the system needs to be controlled. On the other hand, the execution time of a data driven function is not determined a-priori, however it is controlled not to exceed a specified deadline. The following table compares the response time implied by the model with the other execution models (δ is a deadline explicitly declared in the specification).

asyn'	syn'	Δsyn	MASS
$t_e \geq 0$	$t_e = 0$	$\Delta \geq t_e \geq 0$	$\Delta \geq t_e \geq 0$
$t_c > 0$	$t_c = 0$		$t_c = 0$
$t_r > 0$	$t_r = 0$	$\Delta \geq t_r \geq 0$	$\delta \geq t_r > 0$

Note that the synchronous executive continuously processes events while computations are executed under the asynchronous executive. Thus, events are not lost, furthermore it makes possible that a number of incarnations of a single computation are run concurrently.

In this model causality and time bounds relations are deducible including the effect of durational computations. Also, the separation of concerns, taken to its extreme in this paradigm, allows a clear representation of the overall system specification, and mainly eases formal verification and simulation processes of the reactive behavior.

1.5 Related Work

There is much related work concerning requirements and design specification languages. In appendix A we survey a number of languages representing the main approaches in RTS specification. In this section, we refer to languages that have direct relation to the issues that motivated MASS creation.

At the abstract level, MASS is classified as an Interval Temporal Logic (ITL), however the logic underlying MASS more resembles TPTL[4] rather than the classical ITL[32]. In general, real time logics has been studied intensively in the recent years. Current research is mainly concerned with the issues of decidability of continuous models of time, and the specification of hybrid systems systems (see [30] for selected papers). These extensions are neccearry in order to extend the scope of a specification to the controlled process as well. MASS, in comparison, is rather conservative being confined to the scope of the controller, solely. However, MASS emphasizes the issue of *causality* not handled by present specification logics. We introduce a special 'caused-by' operator allowing the relation of events to their causes, as defined in a specification.

Also, there is much work concerning executable logics [32, 19]. However, the notion of execution in these works is of model checking rather then the operational sense used in MASS.

Regarding the programming framework suggested by MASS, it must be mentioned that in fact all the synchronous languages were mainly devised for the reactive behavior specification. However, only few allow formal relation with the functional specification.

- Statecharts[24] includes actions (output events) like **start(f), suspend(f), stop(f)** etc', that refer to the activations of a function **f**. However, there is no formal relation among events and the functions that generate them.

- CRP[9] is a procedural synchronous language augmented with a special construct that specifies the activation of asynchronous computations. A program executing this construct, waits for a signal designating the termination of the computation, or otherwise it may decide to abort the execution. This capability indeed remedies the problem of tracing causality with respect to durational computations.

  This approach is basically different from ours as it engages the system control with the execution of the activated computation, disabling responses to additional occurrences of the activating event. This may be a severe restriction. For instance, a requirement like "activate P upon every occurrence of the event α" (where P is an asynchronous computation) is not expressible in this formalism.

- *eE* (*Embedded Eifel*) [13] employs an execution environment similar to MASS, however the specification framework is different. The reactive behavior is specified in the synchronous language ESTEREL augmented with a special construct **schedule(f)** where **f** is a function. The functions of a system, programmed in the object oriented language *Eiffel*, are classified into real-time and background services. Real time services are run within the synchronous executive, implicitly assumed to respect the synchrony hypothesis.

  The background services are activated by **schedule** commands issued in the reactive part. They are run asynchronously (in a non-preemptive manner) in the time slots between the termination of the real time services and the next time instant, with no hard deadline. A background service can communicate with the reactive part by a special command that adds a signal to the next time instant.

  Comparing with MASS, even though causality is traceable in eE, quantitative temporal relations are not deducible since background services are not bounded. Also, **ef** suffers the same expressiveness problems as all process based languages (see SectionP-ARCH).

In addition, we would like to mention that the *event*$\rightarrow$*action* style is widely used to describe the operation of RTS [29]. However, the extension to the *action*$\rightarrow$*event* relation is not common. (It appears in a very restricted way in [26], where the completion of an action is used to designate execution intervals, rather then as a means of communication.)

Also, the explicit separation of the activation-oriented and functional aspects of a system, can be considered as an example of a *coordination model*, a notion presented in [14] in the context of distributed systems.

1.6 Outline of the Report

Chapter 2 is an informal introduction of MASS. We presents the basic ideas and concepts underlying MASS, and provide an extended survey of the language accompanied with simple specification examples. Chapter 3 provides a formal description of MASS where the semantics of the language are presented with respect to a trace model. (For the sake of completeness, we repeat the syntax definition, so that there is some overlay with Chapter 2.) In chapter 4 we present modularization by structures of acts, and in chapter 5 a case study illustrating a system development with MASS,is worked out in details. Chapter 6 discusses the design decisions involved in the definition of the language. The report is concluded (Chapter 7) with a brief description of our practical experience with MASS.

Chapter 2

The MASS Language

This section presents an informal introduction to MASS. The language is gradually presented, where the motivation for the various constructs, and their usage, are illustrated by proper examples.

2.0.1 The Specification Paradigm

MASS has been designed for the specification of the reactive behavior of real-time systems. However, its computation model establishes a complete system specification paradigm.

The key idea underlying MASS is to represent events and computations as different aspects of a single entity. This entity, called *task*, presents by a function *signature* (an identifier and I/O domains) a common model for a function computation and the events it might generate.

From a functional point of view a task denotes a function that satisfies the signature specification. We require that function to perform pure data transformation, but not any synchronized communication with the environment (say, by *signal, wait*, or *delay* constructs).

Regarding the reactive aspect, we consider a task to represent a set of *basic events* defined by the possible pairs of the function I/O values. Each event denotes the terminations of the computation of the function with the corresponding I/O values.

In this model, a system is represented by a finite set of tasks where the reactive behavior is specified by relating the activation of functions executions to events (namely, the termination of other functions' executions). This may be accompanied by data transfer, where the results of terminating computations are passed as input to the activated function. Such a specification is interpreted as executable instructions for an executive that runs the functions associated with the tasks.

This notion of a task inevitabily suggests a layered system specification paradigm in which the reactive and computational parts evolve from a common set of tasks. The computational specification provides the function associated with each task. The reactive part specifies activation requirements for these functions in terms of tasks communication, i.e., events that activate and constrict the executions of tasks.

The functional and reactive specifications can also be considered standalone. In which case, the reactive specification is interpreted in an abstract level as logical formulae expressing requirements of causal and temporal relations between events. Our semantics ensure that the operational and logical interpretations are compatible.

This capability suggests a method of developmemt where we actually start with a logical specification of the reactive behavior, expressing the system requirements. Then, it is transformed into an executable specification only by the couplling with the functional part (the specification remains untouched).

2.0.2 Basic Notions

The primitive entities in a MASS specification are tasks. A task is declared in the form of a function prototype, like

Switch:{on, off} , Activate_furnace

The task Switch denotes the operation of a bi-state switch. It is declared with no formal input. The output domain {on,off} represents the possible outcomes of a computation. The task Activate_furnace denotes a function with neither input, nor output, intended to represent an operation of a furnace activation.

Note that in both examples the operations expected to be accomplished by the declared tasks were stated informally. Indeed, regarding the reactive behavior the only obligation imposed on the function denoted by a task, is to respect the I/O signature in the declaration. Its concrete specification is expected to be given in the computational part.

With each task declaration, the language defines a set of *basic events*. For instance, the basic events derived from the declaration of Switch are,

{ Switch=on, Switch=off }

Each denotes the terminations of a possible computation of the task Switch. For instance, the event Switch=on occurs at every time point where a computation of Switch had terminated with the value on. In addition the language defines timing events, like 2sec, and event operators that provide for the sprcification of complex scenarios involving occurrences of basic and timing events.

A specification in MASS, called an *act*, describes the reactive behavior required of, so called, *system* tasks. The description may refer, however, to occurrences of *environment* tasks whose behavior is not given in the act. In the operational sense a system task represents a function whose execution is fully controlled within the specified system. In contrast, an environment task is activated and executed outside the system, but the basic events induced by its executions are observable, and may be used to specify the system tasks' behavior.

For example, if Switch is declared as an environment task, it can be used to represent the events of turning on and off a master switch that drives the system operation. However, nothing is assumed regarding the behavior that leads to the switch operations. In contrast, the behavior of the task Activate_furnace, if declared as a system task, must be given by the act.

The concrete behavior of a system task is specified by reactions. For instance, the reaction

$$[\mathsf{Switch{=}on} \rightarrow \mathsf{Activate\_furnace}\,] \leq 2\mathsf{sec}$$

specifies the behavior required of the task Activate_furnace. As a logical statement, this reaction states that every occurrence of Switch=on should be followed, within 2 seconds, by the event Activate_furnace caused by that occurrence of Switch=on. Operationally, the reaction is interpreted as an instruction that following each occurrence of Switch=on the function denoted by Activate_furnace, should be executed within 2 seconds. The semantics of the language ensures that the operational interpretation generates a behavior that is consistent with the logical interpretation.

MASS also supports a modularization of a specification by a pair of formalisms

- A system task can be refined into another, lower level, system specification (considered an implementation of that task).
- A task can be defined as a composition of subsystems, which may communicate through common tasks.

These formalisms enable the representation of a system by a hierarchical structure of concurrent subsystems. For example, the task *Activate_furnace*, regarded atomic at the top-level specification of the furnace, might consist of a number of lower level tasks, e.g., one that opens a gas valve, another that performs the ignition, and a third that monitors the flame. These tasks form a refining system that specifies the computation corresponding to *Activate_furnace*.

2.0.3 System Specification

A specification unit in MASS, called an *act*, declares the behavior required of a real-time system. The general syntax of an act is as follows (boldfaced tokens are reserved words, and the notation $t\ldots$ means a finite list of terms of the type of t).

Act *identifier* **is**
 Tasks
  System *task* ...
  Environment *task* ...
 Reactions *reaction* ...
 TimeBase *period*
End

The identifier following **Act** provides a unique name for the specification unit. The **Tasks** section presents the *tasks* that participate in the system operation, classified the **Environment** and **System** types. A task is declared in the form:

$$task(Input) : Output$$

where *task* is the name of the task, and *Input, Output* are optional finite sets of values specified either by a type identifier, or in the form $\{v, \ldots\}$. For example,

IdentifyObject(Location):{ball, cone, cube}

is a task declaration where the input domain is given by a type identifier and the output domain is explicitly listed.

System Behavior

The **Reactions** section declares activation requirements for the system tasks declared in an act. An activation requirement specifies temporal and causal relations between *events* that are defined in terms of tasks. We first define event expressions, and then specify the behavioral relations.

Events

Basic events are terms of the form:

$$task(in)\!=\!out$$

where *in* and *out* are elements of the *input* and *output* domains, respectively (each is omitted if the task lacks the corresponding domain). It specifies terminations of those executions of the function denoted by *task* that have been activated with the value *in*, and terminated with the value *out*. Hence, a task declaration, $task(Input)\!:\!Output$, induces the set of basic events:

$$\{\ task(in)\!=\!out \mid in \in Input,\ out \in Output\ \}$$

Also, for each system task we define the terms *task*(in)! to be basic events that denote aborted executions of *task*.

General *event* expressions are inductively defined by closing the set of basic events (and additional fixed symbols) under a number of logical operators. Specifically, the set of *events* with respect to a given act consists of:

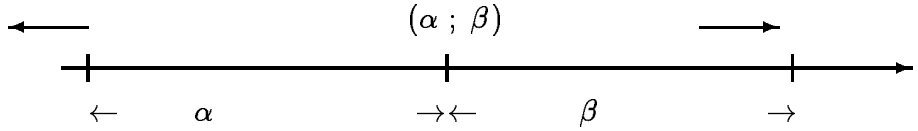
- The basic events derived from the tasks declared in the act,
- Timing events: the words *startup* and *tick*, the symbol $*$, and any term $r\mathbf{u}$ where $r \in Q^+$ (the non-negative rationals), and **u** is a time unit indication (e.g. **ms, sec**),
- For any events α, β, and a basic event e derived from a system task, the expressions

$$(\alpha \rightsquigarrow e),\ (\neg\alpha),\ (\alpha \wedge \beta),\ (\alpha \vee \beta),\ (\alpha;\beta).$$

Events are defined to *occur* at closed time intervals designated by time instants[1] (including singular intervals denoting single time instants). Informally,

[1] Time instants and time units are specified in the **TimeBase** section (see below Section 2.0.4).

- A basic event occurs at every time instant designated by a termination of a proper execution.
- Regarding timing events,
  - *startup* occurs only at the instant the system starts operating.
  - *tick* occurs on every interval designated by subsequent time instants.
  - $r\mathbf{u}$ specifies a duration of r time units $\mathbf{u}$. It occurs at any time interval of this length.
  - $*$ occurs on every time interval (specifying an arbitrary duration).
- The symbols: $\neg, \wedge, \vee$, denote logical negation, conjunction, and disjunction, respectively.
- The operator $\rightsquigarrow$ classifies those occurrences of a basic event e that were caused by the event α (causal executions will be explained shortly).
- The *sequence* operator ';' specifies an ordered occurrence of events. An event $(\alpha;\beta)$ occurs at any interval composed of an occurrence of α immediately followed by an occurrence of β.



Higher level event-operators can be defined in terms of the primitive operators. For instance, the standard temporal operator "eventually" is defined by $\Diamond\alpha \equiv (*;\alpha;*)$, and its dual "always" is defined as usual by $\Box\alpha \equiv \neg\Diamond\neg\alpha$. Also, we define the binary operator $\setminus$ such that $(\alpha \setminus \beta) \equiv (\alpha \wedge \Box(\neg\beta))$. Namely, the event $\alpha \setminus \beta$ occurs on any interval where α occurs such that β does not occur on any sub-interval.

In addition, a basic event can be partially specified in which case it designates a disjunction of basic-events satisfying the specification. In general, $task \equiv \bigvee_{u \in Input} \bigvee_{v \in Output} task(u) = v$. For example, Switch abbreviates the event (Switch=on $\vee$ Switch=off).

Reactions

A reaction describes an activation requirement for a system task in the form:

$$[\ \textit{Activating-event} \rightarrow \textit{Response-task}\ ] : \textit{Aborting-event} \leq \textit{Failing-event}.$$

where

- *Activating-event* is any event expression.
- *Response task* has the form *task-id(in)*, where *task-id* identifies a system task, and *in* is a legal input value (it is omitted if the task lacks an input domain).

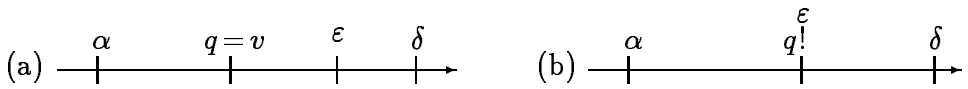


Figure 2.1: Possible scenarios due to a reaction $[\alpha \rightarrow q] : \varepsilon < \delta$

- *Aborting-event* is an event expression that does not contain basic events of the form $q!$ (see remark below).

- *Failing-event* is a timing event of the form $\delta \in Q^+$, or *tick*, or the event false $\equiv \neg *$.

The activating event must explicitly be specified in a reaction, whereas each of the *constricting* events (aborting and failing) may be omitted, in which case it is *false*.

The intended meaning of a reaction is that following each occurrence of the activating event, the termination of the an execution of the response task, that had been activated (with the input value *in*) in response to that occurrence of the activating event, must be observed within the duration designated by the failing event. The termination value is either an element of the output domain, if the task terminates no later than the occurrence of the aborting event (Fig. 2.1a); otherwise it is the value '!', designating the abortion of the execution at the occurrence of the aborting event (Fig. 2.1b).

Operationally, a reaction is interpreted as a command to schedule a computation of the response task upon each occurrence of the activating event. Executions are aborted at the occurrence of the aborting event, if they did not terminate normally earlier. If an execution exceeds the occurrence of the failing event then the system fails (the actual implementation of a system failure is left as a design decision, see Section **??**).

For example, consider the reaction

$$[\ \mathsf{Train\_out} \rightarrow \mathsf{Gate(open)}\ ] : \mathsf{Train\_in} \leq 10\mathsf{sec}$$

where Train_in and Train_out are environment tasks that indicate, respectively, the entrance and exit of a train in a railroad crossing, and the task Gate({open,close}) moves a gate up and down according to the input parameter. The meaning of this reaction is that upon each occurrence of the event Train_out the task Gate should be activated with the input *open*, and it must terminate (or be aborted) within 10 sec. The execution of Gate is aborted, and the event Gate(open)! is generated, if the event Train_in occurs before the computation is completed.

The following remarks express important properties of reactions (formalized in the next section).

- Within an act, a task can be activated independently, in a number of reactions.

- A number of occurrences of an activating event that end together will cause a single activation of the response task.

- An occurrence of an activating event will cause an activation of the response task even if the task is still executing due to a previous activation. Furthermore, it is possible that a number of executions of a single task terminate simultaneously, yielding the

same basic event. These are distinguished (in the language semantics) by relating each termination to the activating event that has caused it.

- Our interpretation does not impose any particular scheduling policy.[2]
- Constricting events play different roles. An aborting event has an operational role. Its occurrence should cause the abortion of the ongoing execution of the response task, if still running. The failing event, on the other hand, specifies a correctness condition. No execution should exceed its occurrence; otherwise the whole system execution is erroneous.
- The reason we forbid aborted executions to be used in aborting events is to avoid inconsistency as can arise, for instance, by a pair of reactions of the form

$$[\alpha \rightarrow q] : \neg p!, \qquad [\beta \rightarrow p] : q!$$

Actually, we could use a weaker constraint, but it makes no practical difference (see discussion in Section 6.1.2).

2.0.4 Executing Acts

In section 3 we describe the semantics of an act by formulae in a certain temporal logic, and operationally by a finite automaton. Here, we informally describe the execution model corresponding to the operational semantics.

The Time Domain

The *period* parameter in the **TimeBase** declaration of an act specifies a quantity $r \in Q^+$ and a time-unit indication. For instance, `0.5sec`, `3ms` are legal period specifications. The execution model assumes that the termination of task's executions are observed along the fixed rate sequence of time instants $t_i = r \cdot i, \quad i \geq 0$ interpreted in the time-base time units.

In general, occurrences of events are related to intervals $[t_i, t_j]$ defined by by the observation sequence. Thus, a time event $t \in Q^+$ (normalized to the time-base time units) occurs at every interval $[t_m, t_n]$ such that $t_{n-1} \leq t_m + t < t_n$. Operationally, the period specification. determines the resolution at which events can be observed, and therefore it affects the extent to which activation requirements can be respected. In practice, the value of r should be chosen depending on the maximal frequency of the controlled process and the allowed delays for the observation of events.

Execution Model

An act is considered to specify a synchronous executive that is periodically activated at the time instants of the observation sequence (a Δ-synchronous model). The executive

[2] It promotes, however, the earliest deadline criterion [17].

maintains a nondeterministic finite automaton that continuously identifies the occurrences of the activating and aborting events (of every reaction). At each time-instant, the observation set is fed as an input letter to the automaton. The observation at t_i consists of those tasks whose executions terminated at the period $[t_{i-1}, t_i)$. The information associated with a termination of a task execution specifies the corresponding input and output values, and for a system task it also contains the instance of the activating event that caused the current execution. However, while terminations of environment tasks are acceptable at any time instant, the automaton rejects unexpected terminations of system tasks (see below).

Based on the specified reactions the executive controls the system operation by aborting and scheduling executions of **system** tasks. A reaction for which the occurrence of the activating event has been detected starts expecting for an observation indicating the termination of the response task that is associated with the time instant of activating event occurrence. The expectation is ended either by the completion of the execution, or the occurrence of the aborting event, whichever comes first. If none of them occur prior to the occurrence of the failing event, then the whole execution is declared to fail.

This model relies on the essential assumption that the evaluation of the reactions carried out at each time instant, necessarily terminates before the next time instant (otherwise aborting and failing events would not be respected in time). Thus, MASS is internally a *synchronous* language [6]. (This should not be confused with the nature of the computations denoted by tasks, that are in general asynchronous.)

2.0.5 A Specification Example

We specify a system that controls the passage of a train in a one-way railroad crossing (Fig. 2.2). It consists of a rail segment equipped with a pair of sensors indicating the entrance and exit of a train, a traffic signal for trains, and a gate mounted at its side. Whenever a train approaches the crossing, the gate, initially open, has to be closed. After the train leaves the crossing, the gate is reopened. In addition, if the gate does not close within 4 seconds (from the moment a train enters the crossing region) the signal is turned on, in which case it must be turned off when the gate is actually closed. We also require that gate operations always terminates within 10sec., and that the signal reacts immediately.

It is assumed that trains arrive at the crossing at a sufficiently low rate, and leave it quickly enough, so that no train attempts to enter the crossing while another train is already there. However, it is possible that a train enters the crossing region while the gate is opening. In this case, the gate must reverse its movement and start closing.

The act Cross_control (Fig. 2.3) is a specification of this process. It consists of a pair of environment tasks, Train_in and Train_out, representing the entrance and exit sensors, respectively. These tasks lack both input and output domains, their termination (probably reported by an interrupt) is sufficient to indicate the detection of a train passage.

The act controls the operation of the tasks Gate, and Signal. The task Gate can be activated in one of two modes, given by its input definition: {close, open}. The activations of Gate(close) and Gate(open) are expected to terminate with the gate in a closed and open

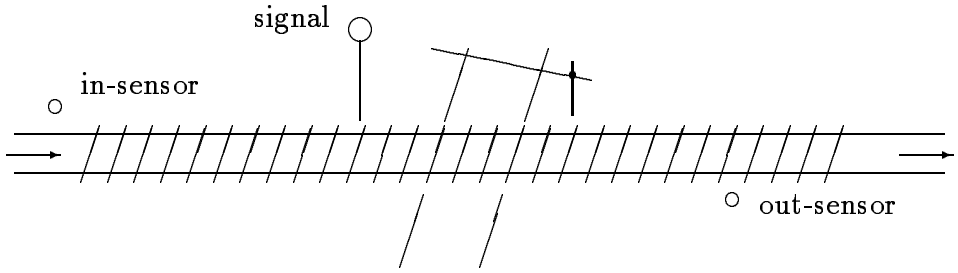


Figure 2.2: The Railroad Crossing

```
Act Cross_Control is
    Tasks
        Environment  Train_in Train_out
        System       Gate({close,open}) Signal({on,off})
    Reactions:
        [ Train_in  →  Gate(close) ] <10sec
        [ (Train_in;(4sec\Gate(close)))  →  Signal(on)] ≤tick,
        [ (startup ∨ Train_out)  →  Gate(open) ] :Train_in <10sec
        [ (startup ∨ Gate(close))  →  Signal(off) ] ≤tick
    TimeBase 2ms
End
```

Figure 2.3: A specification of the Railroad Crossing Controller

positions, respectively. Note, however, that these expectations have no formal basis; it is the functional specification of Gate that ensures the required functioning. The activations of Signal(on) and Signal(off) set the signal indication to "stop" or "pass", respectively.

The reactions describe the required behavior in case of an entrance or exit, of a train. The first reaction states that whenever a train enters the crossing, the gate must be closed. Recall, the event Train_in is instantaneously generated at a time point of the termination of an execution of the task Train_in. However, only the event, not the execution, is observable by the system.

The activating event in the second reaction, (Train_in;(4sec\Gate(close))), literally means "4 seconds passed after a train entrance and the event denoting the gate closing did not occur". The failing event tick expresses the requirement for immediate response of the task Signal. (recall that in MASS model tasks are not executed instantaneously and therefore the fastest result cannot be observed but at the next time instant).

The third reaction states that upon system start-up and following each occurrence of Train_out, the task Gate(open) should be activated, and its termination must occur within 10 seconds. However, its execution is aborted if the event Train_in occurs meanwhile (which in turn also causes the gate to start closing due to the first reaction).

The fourth reaction states that the signal should be turned off at start-up and every time the gate is closed after the signal has been activated to be turned on. Again, we use tick as a failing event to indicate the immediate response required by Signal.

The TimeBase value specifies the system to operate at a 2ms execution rate. We empha-

size that this quantity must be a result of a system analysis (cf. Section 2.0.4).

Due to the declarative nature of MASS, additional requirements are easily integrated into an act specification. For instance, suppose the gate is equipped with a special "test" button, which if pressed should cause the gate to be closed and immediately reopened. We assume that the button will be pressed only if there is no train at the crossing; however, a train may enter during a test, in which case the process must be aborted. In order to specify the additional activations of the task Gate, we add an environment task Test (representing the button), and the reactions:

[Test $\rightarrow$ Gate(close)] :Train_in $\leq$10sec
[($\neg$Train_in $\wedge$ Test$\rightsquigarrow$Gate(close)) $\rightarrow$ Gate(open)] :Train_in $\leq$10sec

The activating event in the second reaction states that the opening of the gate should be activated only if it completed closing due to a test, and no train approaches the crossing at that instant (the conjunction with $\neg$Train_in is necessary since in our semantics normal termination has priority over abortion). Specifying Train_in as an aborting event in both reactions assures that the process is aborted if a train approaches the crossing during opening or closing.

2.0.6 Language Extensions

This section presents additional constructs that enable the specification of activation relations that are not expressible by the basic constructs presented so far.

Virtual Tasks

Virtual tasks provide a formalism that allows one to define a new task whose basic events are identified with occurrences of events generated by other, previously defined, tasks. A virtual task has no executions of its own; its behavior merely reflects the executions of the tasks used to define its basic events. In particular, it cannot be declared as a response task in a reaction; its basic events, however, can be used in a specification of an activating event.

Virtual tasks are declared in a special subsection, called **Virtual**, within the scope of the **Task** section (like system and environment tasks).

Virtual *Virtual-task-declaration* ...

A *Virtual-task-declaration* is an extended form of a task declaration given as follows:

task: $\{v_1, \ldots, v_m\}$ **where** *task*=v_1 **at** α_1 ... *task*=v_m **at** α_m.

The first part is a normal task declaration (it always lacks an input domain since no activation is possible). The *where* part interprets the basic events induced by a virtual task, called *virtual events*, by events expressed in terms of previously declared tasks (α_i is called the *marking* event of *task*=v_i).

A virtual event is defined to occur at the singular time intervals which end the intervals designating the occurrences of its marking event. A marking event may itself be defined in

terms of virtual events; however, in order to avoid a circularity, a marking event may not refer to present occurrences of virtual events (cf. Section 3.2.1 for the formal definition). Also, note that a number of marking events, of a single virtual task, may happen to occur simultaneously in which case the corresponding virtual events occur as well.

Basically, virtual tasks are a useful means for abbreviating a complex event expression by a basic event. However, in the general case a recursive specification provides us with the power to represent regular expressions. For instance, a periodic event that occurs every 7ms can declared by the virtual task,

periodic_7ms where periodic_7ms at (startup ∨ (periodic_7ms ; 7ms))

In a specification of a finite automaton, the virtual events actually denote states, and the marking events define the transitions. For example, we characterize the behavior of the railroad crossing as a two-state process where the states "busy" and "free" indicate, respectively, the presence and absence of trains inside the crossing.

Xstate: { free, busy } where Xstate=busy at Train_in,
Xstate=free at (startup ∨ Train_out)

Of special interest is the case where a virtual task is used in the definition of another virtual task. For instance, suppose it is required that hoots should be sounded every second while a train occupies the crossing. For that purpose, we define,

HootTime where HootTime at (Xstate=busy ∨ (HootTime;(1sec\Xstate=free))

and add a system task Hoot, and the reaction

[HootTime → Hoot] ≤tick

The *ExcludeSets* Construct

A pair of tasks are said to be exclusive if their executions are not allowed to overlap. The capability to specify exclusive tasks is mainly useful for the representation of mutual exclusion requirements. Sets of excluding tasks are declared in MASS by,

ExcludeSets *ExcludeSet* ...

where a term *ExcludeSet* is a list of system tasks (possibly associated with input values). Note that an exclusion set may consist of a single task, in which case it means that concurrent activations of that task should be serialized.

For example, a declaration ExcludeSets { Gate } expresses the requirement that the gate in the crossing should not be opened and closed, concurrently.

The semantics of the *ExcludeSets* construct relies on the notion of *starting* events that enables designation of execution intervals for system tasks (cf. Section 3.2.2).

Message Passing

In a reaction, it is possible to specify data passing between a terminated execution of a task, and tasks activated by that execution. It is expressed by a free variable that appears both in

the activating event, and the input of the activated task. For instance, the intended meaning of a reaction:

$$[\ p{=}X\ \rightarrow\ q(X)\ ] \leq \delta$$

is that each termination of p should cause the activation of q with the current output value of p as an input value of q. Obviously, the output and input domains must match.

For example, assume that the entrance sensor in the railroad crossing is also capable of identifying the registration-number of the passing train, which must be further transmitted to a control center within 7sec. To specify this extension we redefine Train_in:IdNum, add a system task Send_Train_Id(IdNum), and add the reaction,

$$[\ \mathsf{Train\_in{=}X}\ \rightarrow\ \mathsf{Send\_Train\_Id(X)}\ ] \leq \mathsf{7sec}$$

Chapter 3

The Semantics of MASS

This chapter describes a formal semantics for MASS. We define an abstract semantics over a domain of timed traces, and a consistent operational semantics over the finite prefixes of runs.

In Section 3.1 we present the abstract semantics for the kernel of MASS, and Section 3.2 completes the semantics for the constructs of virtual tasks, task exclusion, and data transfer. In Section 3.3 we present the operational semantics and show the relationship with the abstract semantics.

3.1 MASS Kernel

This section presents a semantics for the kernel of MASS. We define both axiomatic and operational semantics, and show their mutual relation. We start with a simplified version where in an act declaration

Act *identifier* **is**
  Tasks
    System *task-name* ...
    Environment *task-name* ...
  Reactions *reaction* ...
End

- tasks are names of procedures with no I/O domains.
- the set of events derived from an act is the minimal set that contains the task names (basic events), the symbols: *startup*, $*$, 0, 1 (timing events), and for every events α, β, and system task q, the events: $(\alpha \rightsquigarrow q)$, $(\neg\alpha)$, $(\alpha \vee \beta)$, $(\alpha;\beta)$.
- a reaction is an expression of the form $[\alpha \rightarrow q] \leq r$, where α is an event (the activating event), q a system task (the response task), and $r \in \mathbb{N}$ (the deadline event) is defined recursively as an abbreviation for $(r-1;1)$ for $r \geq 2$.

- it contains no time base declaration.

The extension to the full syntax is of technical nature, and will be introduced at the end of the section.

3.1.1 The Underlying Logic of MASS

We present a causal logic that extends the language of MASS events, and give its interpretation into a domain of timed traces. The semantics of an act will be given by translating the reactions into formulae of the logic.

Syntax

Given a finite set of *task* names, Φ, and a set of variables, we define a language $E(\Phi)$, whose formulae are called *events*, as the minimal set that contains Φ (basic events), the symbols *startup*, 0, 1, $*$ (timing events), and

- If $\alpha \in E(\Phi)$ does not contain variable symbols, $q \in \Phi$, and t is a variable, then: $(\alpha \rightsquigarrow q)$, $(\alpha @ t \rightsquigarrow q) \in E(\Phi)$.
- If $\alpha, \beta \in E(\Phi)$ then: $(\neg\alpha)$, $(\alpha \vee \beta)$, $(\alpha; \beta) \in E(\Phi)$.
- If $\alpha \in E(\Phi)$ and t is a variable that does not appear in α, then $\alpha @ t \in E(\Phi)$.
- For every $\alpha \in E(\Phi)$ and variable t, $\forall t.\alpha \in E(\Phi)$.

Also, we define the standard temporal operator "eventually" by $\Diamond\alpha \equiv (*; \alpha; *)$, and its dual "always" is defined as usual by $\Box\alpha \equiv \neg\Diamond\neg\alpha$.

Semantics

We construct the semantic domain as a set of *runs* that are timed sequences of *causes*. The time domain is modelled by $(\mathbb{N}, \leq, \mathrm{Suc})$, the theory of linear order and successor over the natural numbers (including zero).[1] We refer to $i \in \mathbb{N}$ as a *time instant*, and use the notation $[m, n]$ to denote a closed interval over $\mathbb{N}$ (implicitly meaning $m \leq n$). The variables that appear in an event $\alpha \in E(\Phi)$ are referred to as *time* variables.

Definition 1 A *signature* is a pair $\langle \Phi, C \rangle$ where Φ is a set of tasks, and C is a finite subset of $E(\Phi)$ whose elements do not contain time variables.

Definition 2 A *trace* of a task $q \in \Phi$ with respect to a signature $\langle \Phi, C \rangle$ is a function, $tr_q : \mathbb{N} \to 2^{(C \times \mathbb{N}) \cup \{-\}}$, such that for all $j \in \mathbb{N}$ and $(\alpha, i) \in tr_q(j)$, $0 < i \leq j$. A *run* is a set of traces $\{tr_q \mid q \in \Phi\}$, that contains exactly one trace for each $q \in \Phi$.

[1] The symbols $=, <, +, -$, will be used with the standard interpretation of equality, strict linear order, addition, and (partial) substraction of constants (all are definable in terms of $\leq$ and *Suc*).

Informally, a *cause* $(\alpha, i) \in tr_q(j)$ indicates the occurrence of q at j due to a previous occurrence of α that ended at the time instant $j - i$. The condition $i \leq j$ is necessary to ensure that a trace does not refer to causes that occurred prior to time 0. The condition $i > 0$ means that a reaction does take time to occur (the case $i = 0$ would denote so-called *synchronous* reactions). The symbol $- \in tr_q(j)$ indicates an occurrence of q with unspecified reason.

Definition 3 A *model* of a signature $\langle \Phi, C \rangle$ is a pair (τ, V) where τ is a run of $\langle \Phi, C \rangle$, and V is a valuation that assigns a natural number to each time variable.

Next, we define the satisfiability relation $(\tau, V), [m, n] \models \alpha$. It states the condition for a model (τ, V) to satisfy an event α at a time interval $[m, n]$.

Definition 4 Given a run $\tau = \{tr_q\}$ of Φ, and a valuation V, we define $(\tau, V), [m, n] \models \alpha$ inductively on the structure of α, as follows:

- $(\tau, V), [m, n] \models q$ iff $m = n$ and $tr_q(m) \neq \emptyset$.
- $(\tau, V), [m, n] \models \alpha \rightsquigarrow q$, where α does not contain variables,
  iff $(\tau, V), [m, n] \models q$, and there exists i s.t. $(\alpha, i) \in tr_q(m)$,

  $(\tau, V), [m, n] \models \alpha@t \rightsquigarrow q$, iff $(\tau, V), [m, n] \models q$, and there exists i s.t. $(\alpha, i) \in tr_q(m)$
  and $V(t) = m - i$.
- $(\tau, V), [m, n] \models \mathit{startup}$ iff $m = n = 0$.
- $(\tau, V), [m, n] \models 0$ iff $m = n$.
- $(\tau, V), [m, n] \models 1$ iff $m + 1 = n$.
- $(\tau, V), [m, n] \models *$ for every interval $[m, n]$.
- $(\tau, V), [m, n] \models \neg\alpha$ iff $(\tau, V), [m, n] \not\models \alpha$.
- $(\tau, V), [m, n] \models \alpha \vee \beta$ iff $(\tau, V), [m, n] \models \alpha$ or $(\tau, V), [m, n] \models \beta$.
- $(\tau, V), [m, n] \models \alpha; \beta$ iff there exists $m \leq l \leq n$ s.t.
  $(\tau, V), [m, l] \models \alpha$ and $(\tau, V), [l, n] \models \beta$.
- $(\tau, V), [m, n] \models \alpha@t$ iff $m = n$ and $V(t) = m$ and there exists l s.t. $(\tau, V), [l, m] \models \alpha$.
- $(\tau, V), [m, n] \models \forall t.\alpha$ iff for every k, $(\tau, V[k/t]), [m, n] \models \alpha$.

Finally, we define the concepts of a satisfiable and valid event as follows:

- An event α is *satisfiable* iff there exist a run τ, a valuation V, and an interval $[m, n]$, such that $(\tau, V), [m, n] \models \alpha$.

- An event α is *τ-valid* with respect to a run τ, denoted by $\tau \models \alpha$, iff $(\tau, V), [m, n] \models \alpha$ for every valuation V and interval $[m, n]$.
- An event α is *valid*, denoted by $\models \alpha$, iff $\tau \models \alpha$ for every run τ.

The Semantics of an Act

We define the semantics of an act by stating events that express the behavior of its reactions. Given an act A, let Φ_S, Φ_E denote the sets of system and environment tasks, and $\Phi = \Phi_S \cup \Phi_E$. For every system task q, let $[\alpha_1 \to q] \leq \delta_1, \ldots, [\alpha_{n_q} \to q] \leq \delta_{n_q}$ be all the reactions in A in which the system task q appears as the activated task, and let $E_q = \{\alpha_1, \ldots, \alpha_{n_q}\}$ (without loss of generality, we assume $E_q \neq \emptyset$ for every $q \in \Phi_S$, and by convention we denote $E_q = \emptyset$ for every $q \in \Phi_E$). Then, we define $C = \bigcup_{q \in \Phi_S} E_q$, and take $\langle \Phi, C \rangle$ to be the signature of A. (The set of MASS events derived from A forms a subset of $E(\Phi)$.)

The intended meaning of the reactions in an act is expressed by the following events:

- **Causality-1**

$$X_q \equiv \neg \bigvee_{\alpha \in C - E_q} (\alpha \rightsquigarrow q)$$

  This event requires that no occurrence of a task can have a cause not specified in the act.

- **Causality-2**

$$C_q \equiv (q \to \bigvee_{\alpha \in E_q} (\alpha \rightsquigarrow q)) \wedge \bigwedge_{\alpha \in E_q} \forall t.((0@t; *; \alpha@t \rightsquigarrow q) \to (\alpha@t; *))$$

  This event requires that every occurrence of a system task q is related with, at least, one cause that is an activating event of q. Furthermore, every cause mentioned in a trace actually occurred at the specified time.

- **Single Activation**

$$S_q \equiv \bigwedge_{\alpha \in E_q} \forall t.\neg(\alpha@t \rightsquigarrow q; 1; *; \alpha@t \rightsquigarrow q)$$

  This event expresses the property that no two different occurrences of a system task may refer to the same cause.

- **Responsiveness**

$$R_q \equiv \bigwedge_{\alpha \in E_q} \forall t.((\alpha@t; *; \delta_q^\alpha) \to \Diamond(\alpha@t \rightsquigarrow q))$$

  This event expresses the property that every occurrence of an activating event is followed by an occurrence of the activated task caused by that event, within the specified deadline

Definition 5 The semantics of an act A is the set of runs

$$[\![A]\!] = \{\ \tau\ \mid\ \tau \models \bigwedge_{q\in\Phi} X_q\ \wedge\ \bigwedge_{q\in\Phi_S} (C_q \wedge S_q \wedge R_q)\ \}$$

3.1.2 Tasks with I/O Domains

Input Domains

A task q declared with an input domain $I = \{u_1, ..., u_k\}$ is interpreted as a set of task-declarations $\{\, q(u) \mid u \in I\}$. where $q(u)$ denotes the task q associated with a fixed parameter u. In particular, each task $q(u)$ has a separate trace. Serialization is obtained by declaring the task, as a separate exclusion set, within the scope of the **ExcludeSets** section (cf. 3.2.3).

Output Domains

The interpretation of tasks declared with output domains requires an extension of the range of the trace functions. We assume that for every task $q \in \Phi$, the output domain O^q is a finite, non-empty, set. (In order to preserve the uniformity of presentation, we associate all tasks declared without an output domain with a singleton that consists of an arbitrary value.) We redefine the basic events of the language $E(\Phi)$ to be the expressions of the form $q{=}v$ where $v \in O^q$ (instead of the symbol q alone in the base version).

Definition 6 A *trace* of a task q with respect to a signature $\langle \Phi, C\rangle$ is a function,

$$tr_q : \mathbb{N} \rightarrow 2^{(C\times\mathbb{N}\times O^q)\cup(\{-\}\times O^q)}$$

such that for all $j \in \mathbb{N}$ and $(\alpha, i, v) \in tr_q(j)$, $0 < i \leq j$.

Informally, a *cause* (α, i, v) or $(-, v)$ extends a basic cause with the information indicating the a termination of q with the value v.

The satisfaction relation is respectively redefined for the new form of basic events, as follows.

- $(\tau, V), [m, n] \models q{=}v$ iff $m = n$ and $(-, v) \in tr_q(m)$ or there exist α and i s.t. $(\alpha, i, v) \in tr_q(m)$.
- $(\tau, V), [m, n] \models \alpha \rightsquigarrow q{=}v$, where α does not contain variables, iff $(\tau, V), [m, n] \models q{=}v$, and there exists i s.t. $(\alpha, i, v) \in tr_q(m)$.

Finally, the properties defining the semantics of an act are modified in the presence of output domains as follows:

$$X_q \ \equiv\ \neg \bigvee_{\alpha\in C-E_q} \bigvee_{v\in O^q} (\alpha \rightsquigarrow q{=}v)$$

$$C_q \equiv \bigvee_{v \in O^q} (q=v \rightarrow \bigvee_{\alpha \in E_q} (\alpha \rightsquigarrow q=v)) \wedge \bigwedge_{\alpha \in E_q} \forall t.((0@t;*; \bigvee_{v \in O^q} (\alpha@t \rightsquigarrow q=v)) \rightarrow (\alpha@t;*))$$

$$S_q \equiv \bigwedge_{\alpha \in E_q} \forall t.\neg((\bigvee_{v \in O^q}(\alpha@t \rightsquigarrow q=v);1;*;\bigvee_{v \in O^q}(\alpha@t \rightsquigarrow q=v)) \vee \bigvee_{v \neq v'}((\alpha@t \rightsquigarrow q=v) \wedge (\alpha@t \rightsquigarrow q=v'))$$

$$R_q \equiv \bigwedge_{\alpha \in E_q} \forall t.((\alpha@t;*;\delta_q^\alpha) \rightarrow \Diamond \bigvee_{v \in O^q} (\alpha@t \rightsquigarrow q=v))$$

3.1.3 Reactions with Aborting Events

In order to specify the semantics of an act with reactions in the general form $[\alpha \rightarrow q] : \varepsilon \leq \delta$, we extend O^q with the special value '!'. For every system task q, we define the legal occurrences of the events $q!$ with respect to ε, as follows ($q!$ abbreviates $q=!$).

$$Ab_q \equiv \bigwedge_{\alpha \in E_q} \forall t.(\alpha@t;*;\alpha@t \rightsquigarrow q!) \leftrightarrow ((\alpha@t;\text{first}(\varepsilon)) \wedge \neg\Diamond \bigvee_{v \neq !}(\alpha@t \rightsquigarrow q=v))$$

where $\text{first}(\alpha) \stackrel{\text{def}}{=} (*;\alpha) \wedge \neg\Diamond(\alpha;1)$. The property A_q is added to the set of properties defining the semantics of an act (Definition 5).

3.2 Extensions of MASS

This section presents Virtual tasks, starting events, and the ExcludeSets construct. These entities allow the specification of additional activation relations which are not expressible by the kernel constructs alone.

3.2.1 Virtual Tasks

In general, we consider virtual tasks as environment tasks, but for every virtual task declaration

$$q : \{v_1, \ldots, v_m\} \textbf{ where } q=v_1 \textbf{ at } \alpha_1 \ \ldots \ q=v_m \textbf{ at } \alpha_m.$$

we require that the basic events $q=v_k$ to occur in the instants at which α_k ended.[2]

Thus, we augment the semantics definition of MASS with the requirement

$$V_q \equiv \bigwedge_{1 \leq k \leq m} \forall t(\alpha_k@t \leftrightarrow (q=v)@t)$$

for every virtual task declaration that appears in the act specification.

[2] Hence, virtual tasks are a kind of synchronous computations.

In case a marking event entails recursive expansion of virtual tasks, the syntactic restriction of such events to the form $\alpha \equiv \beta \vee \bigvee_{i=1}^{k}(\gamma_i; 1)$, where β is free of virtual events, ensures that every evaluation converges to a previous occurrence of α (as can be easily verified by induction on the time of of the occurrence of α).

3.2.2 Starting Events

In this section we add the starting of an execution as an observable event. Starting events are not intended to be explicitly used in an act specification, but provide for the representation of higher level constructs that specify mutual exclusion and refinement (described in the sequel).

A starting event is introduced by defining with every system task q the special basic event $q\uparrow$ that denotes the starts of the executions of q. The value $\uparrow$ does not extend O^q instead the range of tr_q is extended to $2^{(C \times \mathbb{N} \times (O^q \cup \{\uparrow\})) \cup (\{-\} \times (O^q \cup \{\uparrow\}))}$. We require that the properties X_q, C_q, S_q separately hold for the event $q\uparrow$. Instead of R_q we require for a starting event the property

$$St_q \equiv \forall t(\alpha@t\ ;\ *\ ;\ \bigvee_{v \in (O^q - \{!\}} \alpha@t \rightsquigarrow q{=}v) \rightarrow \Diamond(\alpha@t \rightsquigarrow q\uparrow)$$

which means that every normal termination of a task is necessarily preceded by a starting event, where both refer to the same cause.

Remarks:

- We do not require that every starting event is followed by a corresponding termination event, thus allowing for non-terminating computations in case both the aborting and failing events do not occur.
- Also, a termination by aborting is not necessarily preceded by a starting event. Indeed, this possibility reflects a situation where the execution had never started prior to the occurrence of the aborting event.
- In an observed trace, it is possible that the events denoting the start and termination of a computation, occur at the same time-instant. However, that means that our time base is not refined enough, rather then real simultaneous occurrence.

3.2.3 The *ExcludeSets* Construct

The semantics of a declaration **ExcludeSets** $M_1, \ldots, M_n$ where M_i, is a list of system task identifiers $\{q_1, \ldots, q_m\}$, excludes runs in which tasks listed in a common exclusion set M_i have overlapping executions. Formally, we characterize this requirement by the property

$$\bigwedge_{1 \le i \le n} \bigwedge_{p,q \in M_i} (\alpha@t \rightsquigarrow q\uparrow; *; (\bigvee_{v \in O^q} \alpha@t \rightsquigarrow q{=}v \vee \alpha@t \rightsquigarrow q!) \rightarrow \neg\Diamond p\uparrow)$$

Remarks

- The executions of each task in an exclusion set are serialized, as well.
- Aborted executions, which are not preceded by proper starting events, may occur within an execution interval of another task in the same exclusion set.
- Mutual exclusion is not enforced with respect to non-terminating executions.

3.3 Operational Semantics

In this section, we represent the semantics of an act by a regular language. We first define the corresponding alphabet Σ, and construct a regular language for every event expressible in the act vocabulary of tasks. Then, we show how to compose a regular expression according to the semantics of the act reactions.

Next, we show that the language accepted by the regular expression is actually all the finite prefixes of admissible runs. Thus we prove the operational semantics is consistent with the logical semantics in the sense of *realizability* defined by Abadi and Lamport [1].

In practice, the regular expression is transformed into a finite automata that monitors the occurrences of events, and reacts synchronously by proper activation and abortion of system tasks.

3.3.1 The Alphabet Σ

In general, we denote by Φ_E, Φ_S, the environment and system tasks respectively. For every $q \in \Phi_S$, E_q denotes the set of the activating events of q, and for every $\alpha \in E_q$, d_q^α denotes the deadline of q as specified by the reactions in the act.

For every $q \in \Phi_S$, and $\alpha \in E_q$, we define the symbols: $q_\alpha^1, \ldots, q_\alpha^{d_q^\alpha}$. Our alphabet is

$$\Sigma = 2^U \quad \text{where } U = \Phi_E \cup \{q_\alpha^i \mid q \in \Phi_S, \alpha \in E_q, 1 \le i \le d_q^\alpha\} \cup \{\text{startup}\}$$

Also, for every $u \in U$ we denote by $\hat{u}$ the set of letters that contain u. Namely, $\hat{u} = \{a \in \Sigma \mid u \in a\}$.

3.3.2 Interpreting MASS Events by Regular Languages

For every event α in MASS, we define a regular language $L(\alpha)$ over Σ.

- For $q \in \Phi_E$: $L(q) = \{a \in \hat{q}\}$.
- For $q \in \Phi_S$: $L(q) = \{a \in \bigcup_{\alpha \in E_q} \bigcup_{i=1}^{d_\alpha^q} \hat{q}_\alpha^i\}$.
- For $\alpha \rightsquigarrow q$: $L(q) = \{a \in \bigcup_{i=1}^{d_\alpha^q} \hat{q}_\alpha^i\}$.
- For timing events:

  $L(\text{startup}) = \{a \mid a \in \Sigma, \text{startup} \in a\}$, $L(0) = \Sigma$, $L(1) = \Sigma^2$, $L(*) = \Sigma^*$.

- For events α, β:
  - $L(\neg\alpha) = \overline{L(\alpha)}$.
  - $L(\alpha \vee \beta) = L(\alpha) \cup L(\beta)$.
  - $L(\alpha;\beta) = \{axb \mid a, b \in \Sigma^*, x \in \Sigma, ax \in L(\alpha),\ xb \in L(\beta)$.

3.3.3 The Interpretation of an Act

Given a reaction

$$r = [\alpha \rightarrow q] \leq d$$

we define the following languages.

- The language $L_C(r)$ is the complement of the set of scenarios where there is an occurrence of q such that its activation time does not coincide an occurrence of α.

$$L_C(r) = \overline{\overline{L(*;\alpha)}(\bigcup_{l=1}^{d} \Sigma^{l-1}\hat{q}_\alpha^l)L(*)}$$

- The language $L_S(r)$ is the complement of the set of scenarios where more then a single occurrence of q is related to a certain time instant as its activation time.

$$L_S(r) = \overline{L(*)(\bigcup_{(i,j)\in D} \hat{q}_\alpha^i \Sigma^j \hat{q}_\alpha^{i+j+1})L(*)}$$

  where $D = \{(i,j) \mid 1 \leq i \leq d,\ 0 \leq j,\ 1 \leq i+j \leq d-1\}$.

- The language $L_R(r)$ is the complement of the set of scenarios where an occurrence of α is not followed by a proper reaction of q within the specified deadline.

$$L_R(r) = \overline{L(*;\alpha)(\Sigma - \hat{q}_\alpha^1)(\Sigma - \hat{q}_\alpha^2)\ldots(\Sigma - \hat{q}_\alpha^d))L(*)}$$

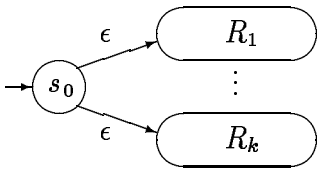
For every act, we define the language $L(\text{startup})$ that restricts the event startup to occur at (and only at) the first time instant.

$$L_{\text{startup}} = L(\text{startup};*) \cap \overline{L(1;*;\text{startup};*)}$$

Let $r_1, \ldots, r_k$ be all the reactions declared in an act. then we define the language of an act A as

$$L_{act}(A) = L_{\text{startup}} \ \cap \bigcap_{i=1}^{k} L_c(r_i) \ \cap \ L_s(r_i) \ \cap \ L_r(r_i)$$

Informally, the structure of the automaton corresponding to $L_{act}(A)$ has the form of a disjunction of automata, one for each reaction.



3.3.4 Runs vs. Words

In this section we show that every word accepted by A_{act} is a prefix of a run in $[\![A]\!]$, and vice versa.

Definition 7 Given $\tau = \{tr_q\}$, we define for every $i \geq 0$ the letter $a_i \in \Sigma$ as follows.

$$a_i \;=\; \{q \mid q \in \Phi_E,\, tr_q(i) = \{-\}\} \;\cup\; \{q^j_\alpha \mid q \in \Phi_S,\, (\alpha, j) \in tr_q(i)\} \;\cup\; \{\text{startup} \mid \text{ if } i = 0\}$$

Also, for every $m \leq n$, we denote by $a[m,n]$ the word $a_m a_{m+1} \ldots a_n \in \Sigma^+$.

Lemma 1 *For every τ and MASS event α, $a[m,n] \in L(\alpha)$ iff $\tau, [m,n] \models \alpha$.*

Proof. We show the lemma by induction on the structure of α

- For $q \in \Phi_E$:

$$\begin{array}{lcl} a[m,n] \in L(q) & \Leftrightarrow & m = n \text{ and } a_m \in \hat{q} \\ & \Leftrightarrow & q \in a_m \qquad\qquad \Leftrightarrow \;\; tr_q(m) = - \;\; \Leftrightarrow \;\; \tau, [m,n] \models q \end{array}$$

- For $q \in \Phi_S$:

$$\begin{array}{lcl} a[m,n] \in L(q) & \Leftrightarrow & m = n \text{ and exist } \alpha \in E_q \text{ and } 1 \leq i \leq d^\alpha_q \text{ s.t. } a_m \in \hat{q}^i_\alpha \\ & \Leftrightarrow & q^i_\alpha \in a_m \\ & \Leftrightarrow & (\alpha, i) \in tr_q(m) \\ & \Leftrightarrow & \tau, [m,n] \models q \end{array}$$

- For $\alpha \rightsquigarrow q$:

$$\begin{array}{lcl} a[m,n] \in L(\alpha \rightsquigarrow q) & \Leftrightarrow & m = n \text{ and exist } 1 \leq i \leq d^\alpha_q \text{ s.t. } a_m \in \hat{q}^i_\alpha \\ & \Leftrightarrow & q^i_\alpha \in a_m \\ & \Leftrightarrow & (\alpha, i) \in tr_q(m) \\ & \Leftrightarrow & \tau, [m,n] \models \alpha \rightsquigarrow q \end{array}$$

- For the timing events:

$$a[m,n] \in L(\text{startup}) \;\Leftrightarrow\; m = n = 0 \;\Leftrightarrow\; \tau, [m,n] \models \text{startup}$$

$$a[m,n] \in L(0) \;\Leftrightarrow\; m = n \;\Leftrightarrow\; \tau, [m,n] \models 0$$

$$a[m,n] \in L(1) \;\Leftrightarrow\; n = m + 1 \;\Leftrightarrow\; \tau, [m,n] \models 1$$

$$a[m,n] \in L(*) \;\Leftrightarrow\; m \leq n \;\Leftrightarrow\; \tau, [m,n] \models *$$

- For $\neg\alpha$

$$a[m,n] \in L(\neg\alpha) \;\Leftrightarrow\; a[m,n] \notin L(\alpha) \;\overset{ind}{\Leftrightarrow}\; \tau, [m,n] \not\models \alpha \;\Leftrightarrow\; \tau, [m,n] \models \neg\alpha$$

- For $\alpha \vee \beta$:

$$\begin{array}{lcl} a[m,n] \in L(\alpha \vee \beta) & \Leftrightarrow & a[m,n] \in L(\alpha) \text{ or } a[m,n] \in L(\beta) \\ & \overset{ind}{\Leftrightarrow} & \tau, [m,n] \models \alpha \text{ or } \tau, [m,n] \models \beta \\ & \Leftrightarrow & \tau, [m,n] \models (\alpha \vee \beta) \end{array}$$

- For $\alpha;\beta$:

$$\begin{array}{rcl} a[m,n] \in L(\alpha;\beta) & \Leftrightarrow & \text{there exists } m \le k \le n \text{ s.t.} \\ & & \quad a[m,k] \in L(\alpha) \text{ and } a[k,n] \in L(\beta) \\ & \overset{ind}{\Leftrightarrow} & \tau,[m,i] \models \alpha \text{ and } \tau,[i,n] \models \beta \\ & \Leftrightarrow & \tau,[m,n] \models (\alpha;\beta) \end{array}$$

In what follows, we assume that every run is faithful in the sense that it does not contain causes that are not expressible in MASS. Namely, for every system task q, $(\alpha,i) \in tr_q(m)$ iff α is an activating event of q, and i is not greater then the corresponding deadline. Note that every run can be made faithful, simply by throwing away the bad elements in every trace. Also, this operation does not influence the content of a_m since by definition of Σ it records only the good elements from every trace. Thus, henceforth we assume that runs are necessarily faithful.

Theorem 1 *Given an act A, $\tau \in [\![A]\!]$ iff for every $m \ge 0$, $a[0,m] \in L_{act}(A)$.*

Proof. We show first that if $\tau \in [\![A]\!]$ then for every $m \ge 0$, $a[0,m] \in L_{act}(A)$.

- By Definition 7 $a_0 \in L(\text{startup})$, and for $i > 0$, $a_i \notin L(\text{startup})$. Therefore, $a[0,m] \in L(\text{startup};*)$ and $a[0,m] \notin L(1;*;\text{startup};*)$. Hence, $a[0,m] \in L_{\text{startup}}$.

- If there exist $m \ge 0$ and a reaction $r = [\alpha \to q] \le d$ such that $a[0,m] \notin L_C(r)$ then:

  $\Rightarrow$ there exists $1 \le l \le d$ s.t. $a[0,m] \in \overline{L(*;\alpha)}\Sigma^{l-1}\hat{q}^l_\alpha L(*)$.

  $\Rightarrow$ there exist k s.t. for every $j \le k$, $a[j,k] \notin L(\alpha)$, and $a_n \in \hat{q}^l_\alpha$, where $n = k + l$.

  $\Rightarrow$ for every $j \le k$, $\tau,[j,k] \not\models \alpha$ (by Lemma 1), and by Def. 7 $(\alpha,l) \in tr_q(n)$.

  $\Rightarrow$ $(\tau, V[k/t]),[k,n] \models (0@t;*;\alpha@t \rightsquigarrow q)$, and for every $j \le k$, $(\tau, V[k/t]),[j,k] \not\models \alpha$.

  $\Rightarrow$ $(\tau, V[k/t]),[k,n] \models (0@t;*;\alpha@t \rightsquigarrow q)$, and $(\tau, V[k/t]),[k,n] \not\models (\alpha@t;*)$.

  In contradiction with $\tau \models C_q$ (implied by $\tau \in [\![A]\!]$). Thus, for every $m \ge 0$,

$$a[0,m] \in \bigcap_{i=1}^{k} L_C(r_i)$$

- If there exist $m \ge 0$ and a reaction $r = [\alpha \to q] \le d$ such that $a[0,m] \notin L_S(r)$ then:

  $\Rightarrow$ there exist i,j such that $1 \le i \le d$, $0 \le j$, $1 \le i + j \le d - 1$, and

$$a[0,m] \in L(*)\hat{q}^i_\alpha \Sigma^j \hat{q}^{i+j+1}_\alpha L(*).$$

  $\Rightarrow$ there exist l,k such that $l < k \le m$ and $a[l,k] \in \hat{q}^i_\alpha \Sigma^j \hat{q}^{i+j+1}_\alpha$.

  $\Rightarrow$ $(\alpha,i) \in tr_q(l)$ and $(\alpha,i+j+1) \in tr_q(k)$.

  $\Rightarrow$ $(\tau, V[j-l/i]),[l,l] \models \alpha@t \rightsquigarrow q$, $(\tau, V[j-l/i]),[k,k] \models \alpha@t \rightsquigarrow q$ (by Definition 7 and the semantics of events), and $\tau,[l,k] \models (1;*)$ (by Lemma 1).

$\Rightarrow$ $(\tau, V[l-i/t]), [l,k] \models (\alpha@t \rightsquigarrow q; 1; *\alpha@t \rightsquigarrow q)$

In contradiction with $\tau \models S_q$ (implied by $\tau \in [\![A]\!]$). Thus, for every $m \geq 0$,

$$a[0,m] \in \bigcap_{i=1}^{k} L_S(r_i).$$

- If there exist $m \geq 0$ and a reaction $r = [\alpha \rightarrow q] \leq d$ such that $a[0,m] \notin L_R(r)$ then:

  $\Rightarrow$ $a[0,m] \in L(*;\alpha)(\Sigma - \hat{q}_\alpha^1)(\Sigma - \hat{q}_\alpha^2)\ldots(\Sigma - \hat{q}_\alpha^d))L(*)$.

  $\Rightarrow$ there exist k, l such that $k < l \leq m$, $a[0,k] \in L(*;\alpha)$, and

  $$a[k+1,l] \in (\Sigma - \hat{q}_\alpha^1)(\Sigma - \hat{q}_\alpha^2)\ldots(\Sigma - \hat{q}_\alpha^d)).$$

  $\Rightarrow$ there exists $i < k$ such that $a[i,k] \in L(\alpha)$ and for every $1 \leq j \leq d$, $(\alpha, j) \notin tr_q(k+j)$.

  $\Rightarrow$ $(\tau, V[k/t]), [k,k] \models \alpha@t$ (by Lemma 1), and for every $0 \leq j \leq d$,

  $$(\tau, V[k/t]), [k+j, k+j] \not\models \alpha@t \rightsquigarrow q$$

  (by Definition 7 and the semantics of events).

  $\Rightarrow$ $(\tau, V[k/t]), [k, k+d] \models (\alpha@t; d)$, and $(\tau, V[k/t]), [k, k+d] \not\models (*; \alpha@t \rightsquigarrow q; *)$ (by the semantics of events).

  In contradiction with $\tau \models R_q$ (implied by $\tau \in [\![A]\!]$). Thus, for every $m \geq 0$,

  $$a[0,m] \in \bigcap_{i=1}^{k} L_R(r_i).$$

In the other direction, assume that $a[0,m] \in L_{act}(A)$ for every $m \geq 0$. We show that $\tau \in [\![A]\!]$.

- $\tau \models \bigwedge_{q \in \Phi} X_q$ since τ is faithful.

- If $\tau \models \neg \bigwedge_{\alpha \in E_q} \forall t.((0@t; *; \alpha@t \rightsquigarrow q) \rightarrow (\alpha@t; *))$ then there exists q such that $\tau \models \neg \forall t.((0@t; *; \alpha@t \rightsquigarrow q) \rightarrow (\alpha@t; *))$. Let $[m,n]$ be an interval, and V a valuation, such that $(\tau, V)[m,n] \models \neg((0@t; *; \alpha@t \rightsquigarrow q) \rightarrow (\alpha@t; *))$.

  Then, $(\tau, V)[m,n] \models (0@t; *; \alpha@t \rightsquigarrow q)$ and $(\tau, V)[m,n] \models \neg(\alpha@t; *)$. Therefore, by the semantics of events $(\alpha, n-m) \in tr_q(n)$ and for every $k \leq m$, $(\tau, V)[k,m] \models \neg\alpha)$.

  Hence, by Def. 7 and Lemma 1 $a_n \in \hat{q}_\alpha^l$ where $l = n - m$, and for every $k \leq m$, $a[k,m] \in \overline{L(\alpha)}$. Therefore, $a[0,m] \in \overline{L(*;\alpha)}$, and $a[m+1,n] \in \Sigma^{l-1}\hat{q}_\alpha^l$ (note that by the semantics of events $n > m$). Thus, $a[0,n] \in \overline{L(\alpha)}\Sigma^{l-1}\hat{q}_\alpha^l L(*)$ in contradiction with $a[0,n] \in L_C$ (since $a[0,n] \in L_{act}$).

  Also, $\tau \models (q \rightarrow \bigvee_{\alpha \in E_q}(\alpha \rightsquigarrow q))$ since τ is faithful. Hence $\tau \models \bigwedge_{q \in \Phi_S} C_q$.

- If for some $q \in \Phi_S$, and $\alpha \in E_q$, $\tau \models \neg\forall t.\neg(\alpha@t \rightsquigarrow q; 1; *; \alpha@t \rightsquigarrow q)$ then there an interval $[m,n]$, and a valuation V, such that $(\tau,V)[m,n] \models (\alpha@t \rightsquigarrow q; 1; *; \alpha@t \rightsquigarrow q)$. Therefore, by the semantics of events there exists $1 \leq i \leq d$ and $0 \leq j$ s.t. $i+j \leq d-1$, $(\alpha,i) \in tr_q(m)$, and $(\alpha,i+j) \in tr_q(n)$. Therefore, by Def. 7 $a[m,n] \in \hat{q}^i_\alpha \Sigma^j \hat{q}^{i+j+1}_\alpha$ in contradiction with $a[0,n] \in L_S$ (since $a[0,n] \in L_{act}$). Hence $\tau \models \bigwedge_{q\in\Phi_S} C_q$.

- If for some $q \in \Phi_S$, and $\alpha \in E_q$, $\tau \models \neg\forall t.((\alpha@t; d^\alpha_q) \rightarrow \Diamond(\alpha@t \rightsquigarrow q))$ then there is an interval $[m,n]$, and a valuation V, such that $(\tau,V)[m,n] \models (\alpha@t; d^\alpha_q)$, and $(\tau,V)[m,n] \models \neg\Diamond(\alpha@t \rightsquigarrow q))$.

  Therefore, by the semantics of events there exists $k \leq m$ such that $(\tau,V)[k,m] \models \alpha$, $n-m=d$, and for every $1 \leq l \leq d$, $(\alpha,l) \notin tr_q(m+l)$.

  Hence, by Def. 7 and Lemma 1 $a[0,m] \in L(*;\alpha)$, and

  $$a[m+1,n] \in (\Sigma - \hat{q}^1_\alpha)(\Sigma - \hat{q}^2_\alpha)\ldots(\Sigma - \hat{q}^d_\alpha)).$$

  Therefore, $a[0,n] \in L(*;\alpha)(\Sigma - \hat{q}^1_\alpha)(\Sigma - \hat{q}^2_\alpha)\ldots(\Sigma - \hat{q}^d_\alpha))L(*)$ in contradiction with $a[0,n] \in L_R$ (since $a[0,n] \in L_{act}$). Hence $\tau \models \bigwedge_{q\in\Phi_S} R_q$.

□

Chapter 4

Structures of Acts

In order to support large software system development, a specification language must provide for the partition of a system into a hierarchical tructure of modules that can be independently specified and composed with other modules while retaining their basic properties.

In our framework, an act provides the scope of a module. It clearly satisfies the basic properties required of a module, namely: a closed representation framework for the behavior of a set of tasks, and a clear interface that allows its composition with other acts.

In this chapter we present constructs that allow the partition of a system into a hierarchical structure of acts, and describe the semantics of a structure in a global clock environment.

4.1 Task refinement

Task refinement associates a system task declared in an act (the *refined* task) with a separate act (the *refining* act), that bears the name of the refined task. Thus, a task considered an atomic entity at a certain level of abstraction is represented by an internal structure of tasks and reactions at a lower level of abstraction.

Syntactically this relation is expressed by extending the refining act with a **where** section that maps the basic events induced by its signature (which are also the basic events induced by the refined task) to internal events of the act. We call this form a *closed* act. Specifically, it is represented as follows:

Act $q(\{u_1, \ldots, u_m\}) : \{v_1, \ldots, v_n\}$ **is**
body
where $q(u_1) = v_1$ **at** $\alpha_{1,1}$ $\ldots$ $q(u_m) = v_n$ **at** $\alpha_{m,n}$.
End

The body consists of the normal sections (**Tasks, Reactions**,...), and $\alpha_{i,j}$ are events expressed in terms of tasks specified in the body.[1] We call $\alpha_{i,j}$ the *marking* event of $q(u_i) = v_j$.

[1]Note the resemblance between a task refinement and a virtual task declaration. Indeed, conceptually, these constructs can be considered as specialization and abstraction, respectively.

```
Act Gate({close,open}) is
    Tasks
        System  Motor_cmd({down,up})
                Gate_state:{down,moving,up}.
    Reactions   [ startup(open)  →  Motor_cmd(up) ] < 20ms
                [ startup(close)  →  Motor_cmd(down) ] < 20ms
                [ Periodic(30ms)  →  Gate_state ] < 10ms
    where       Gate(open) at (startup(open)≻Gate_state=up)
                Gate(close) at (startup(close)≻Gate_state=down)
End
```

Figure 4.1: A refinement of the task Gate

A specification of a task refinement must obey the following rule concerning tasks shared by the host — the act where the refined task is declared — and refining acts.

- Every system task, other then the refined task, declared in the host (either system or environment) may also be declared as an environment task in the refined task.

Shared tasks declared under this rule are considered to denote the same object, and therefore will be required to exhibit a consistent behavior (see the semantics section, below). In contrast system tasks in the host and refining acts that bear the same name are considered as different objects.

The operation of the refining act is dominated by the activations of the refined task (occurring due to a proper reaction in the *host* act Each activation would cause a separate run of the refining act which is ended by the first occurrence of a marking event. In which case, it also denotes the termination of the refined task execution with the value designated by the marking event.

For example, Fig 4.1 presents a possible refinement of the task Gate (declared in the act Cross_control, see page 21). It specifies an implementation of the gate operations by a pair of tasks: Motor_cmd initiates the gate-motor to move the gate towards a required position, and Gate_state reports the current status of the gate. The whole process is accomplished by activating Motor_cmd with an input corresponding to the activation value of Gate, and periodically activating Gate_state until it detects the gate has reached the required position. The termination conditions are specified by the marking events. (Note the extended syntax of the event startup that binds it to the activation value of the refined task. For instance, the event startup(open) occurs whenever the task Gate is activated with the input open.) In Cross_control any termination of the act Gate is interpreted as a termination of the local task Gate.

Note that the TimeBase declaration has been omitted in the specification of Gate. Indeed, in our semantics a refining act inherits the time-base of the host act, and their observation sequences are fully synchronized. In particular, the starting events of the task Gate occur in Cross_control simultaneously with the startup events of the act Gate. Similarly, the marking

events of the act Gate occur simultaneously with the corresponding basic events of the task Gate in Cross_control.

4.2 Semantics of Refinement

In what follows, we first present the semantics of a closed act independently of the task it is intended to refine. Then, the semantic domain is generalized to support the presentation of multiple runs, and finally the semantics of a refining act is given in the context of the host act.

4.2.1 Semantics of a Closed Act

The semantics of a closed act is the semantics of the normal act obtained from the closed act by ignoring the **where** section, and transforming every reaction:

$$[\,\alpha \rightarrow q\,] : \varepsilon < \delta \quad \text{into the form:} \quad [\,(\mathit{startup}; *; \alpha) \setminus \text{ter} \rightarrow q\,] : (\varepsilon \vee \text{ter}) \leq \delta.$$

The event ter is given by

$$\text{ter} \equiv (\alpha_1 \vee \ldots \vee \alpha_m \vee \mathit{abort})$$

where α_i are the marking events, and *abort* is an environment task that is implicitly added to the specification of a closed act. Informally, *abort* is assumed to be generated by the environment whenever the refined task is aborted within the host act.

Note that due to the transformation of the activating event α into $(\text{startup}; *; \alpha) \setminus \text{ter}$ once ter occurs the reaction becomes disabled. Moreover, the extension of the aborting event to $(\varepsilon \vee \text{ter})$ will cause the aborting of all ongoing executions of tasks q, respectively generating the events $q!$.

Note also that every normal act can be considered as a closed act where $\text{ter} \equiv \mathit{abort}$. Namely, a normal act employed as a refinement terminates only due to the abortion of the refined task.

4.2.2 System of Runs

A trace of a refined task may specify multiple executions of the refining act that start at various time points, and may even overlap. Thus, in order to express the behavior of a refined task we need to associate with every trace runs of the refining act, and represent thier synchronised behvior.

The semantics given below assumes a global clock that drives all runs, both of the host and the refining acts, in a synchronous manner.[2] Thus, reactions are evaluated at the same time instants in all active acts. However, the durations with respect to the starting time instant differ, in general, from one act to another.

[2] See discussion of asynchronous runs in Section 6.5.2.

Consistency of Runs

We define the term of consistent behavior for traces of tasks shared by a pair of acts. (Recall, we allow an environment or system task of the host act to be declared as an environment task in the refined act.) A consistent behavior refers only to the externally observed behavior exhibited by a trace. Therefore, we define the free form of a trace as follows.

Definition 8 The *free* form of a trace tr_q is the trace tr_q^f defined by,

$$tr_q^f(m) = \begin{cases} \{(-,v) \mid \text{for some } \alpha \text{ and } i, (\alpha,i,v) \in tr_q(m)\} & q \text{ is a system task} \\ tr_q(m) & q \text{ is an environment task} \end{cases}$$

Definition 9 Let A, A', be acts. A run $\tau' \in [\![A']\!]$ is *consistent* with respect to a run $\tau \in [\![A]\!]$ in an interval $[m,n]$ iff for every environment task q of A' that is also declared in A (either as a system, or as an environment task)

$$tr_q(j)' = tr_q^f(m+j) \quad \text{for every} \quad 0 \leq j \leq n-m$$

4.2.3 Admissible Runs

In general, we present the behavior of a refinement by a run of the host act, and a set of runs of the refining act, one for each execution (in the host run) of the refined task. This set of runs must satisfy a pair of relations that express the idea of refinement.

In the following definition we assume that q is a system task in an act A, and A_q a refining act of q.

Definition 10 Let $\tau \in [\![A]\!]$. An *implementation* of the trace $tr_q \in \tau$ is a minimal set of runs $I \subseteq [\![A_q]\!]$ such that:

1. For every $m, n \in \mathbb{N}$, there exists a valuation V such that

   $$(\tau, V), [m,n] \models (q \uparrow @t; *; q \uparrow @t \rightsquigarrow q = v_k)$$

   iff there exists $\tau' \in I$ that is consistent with τ in $[m,n]$ and such that

   $$(\tau', V), [0, n-m] \models (*; \alpha_k) \wedge \neg(\bigvee_{1 \leq i \leq m} \Diamond(\alpha_i; tick)).$$

2. For every $m, n \in \mathbb{N}$, there exists a valuation V such that

   $$(\tau, V), [m,n] \models (q \uparrow @t; *; q \uparrow @t \rightsquigarrow q!)$$

   iff there exists $\tau' \in I$ that is consistent with τ in $[m,n]$ and such that

   $$(\tau', V), [0, n-m] \models \text{first}(\text{abort}_q) \wedge \neg(\bigvee_{1 \leq i \leq m} \Diamond \alpha_i).$$

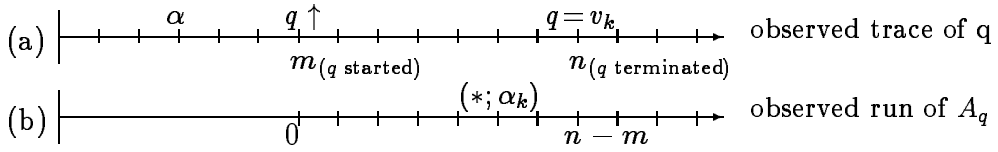


Figure 4.2: Self termination of the refining act: (a) the task q (b) the act A_q.

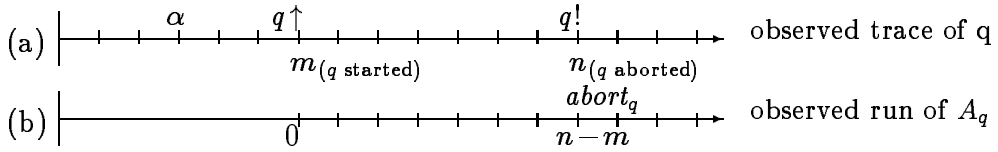


Figure 4.3: Abortion of the refining act: (a) the task q (b) the task abort_q (in A_q).

Informally, the above definition express the following constraints:

- The first requirement specifies a case of normal execution of the refined task (Fig. 4.2). It states that the marking event corresponding to the termination of the refined task occurs just once at an interval that ends the execution of the refined task, and no marking event occurs within the execution of the refined task.
- The second requirement specifies an execution of the refined task that is terminated by abortion (Fig. 4.3), indicated in the refining act by an occurrence of the environment task $abort_q$.

Finally we define the semantics of refinement as follows.

Definition 11 An admissible run of a pair of acts (A, A_q), where A_q is a refinement of the task q in A, is a pair $\langle \tau, I \rangle$ such that $\tau \in [\![A]\!]$, and I is an implementation of $tr_q \in \tau$.

4.2.4 Plays

In general, task refinement enables a hierarchical construction of a system. One starts with an act specifying the operation of a system at a top level view. For instance, a tracking system can be represented at this level by a pair of reactions,

$$[\ (\mathsf{startup} \vee \mathsf{track}{=}\mathsf{fail}) \ \rightarrow \ \mathsf{search}\ ]\ , \qquad [\ \mathsf{search}{=}\mathsf{target} \ \rightarrow \ \mathsf{track}\ ]$$

Then, each of the system tasks – search and track – is separately specified by a refining act that elaborates its operation in terms of lower level (implementation) tasks. The process can

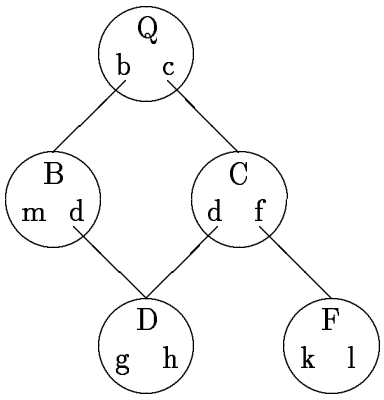


Figure 4.4: A play (**B** is the name of the act refining the task declaration **b** etc.)

be iteratively applied to acts in the second level, and so on to every generated level, until all system tasks get to a level where they are represented by functional computations.

For instance, in a video tracking system the task search could be represented by a task that is responsible for the mechanical scanning of an area, and another task which continuously processes the video signals. These in turn, are further refined depending on the special equipment and system algorithms. Note however that the top level specification equally holds for any other tracking system (radar based, for instance), thus supporting an hierarchy of abstraction levels, and separation of concerns as well.

We call a set of acts that establish an hierarchical structure of refinements, a *play*. Formally, it is presented by a graph whose nodes are acts, and the edges indicate refinements.

Definition 12 Given a pair of acts (A, A'), we say that A' *refines* A, denoted by $\rho(A, A')$, **iff** A' is a closed act whose signature is declared as a system task of A.

Definition 13 (play) A set of acts P is called a *play* **iff** the graph (P, E), where

$$(A, A') \in E \text{ iff } \rho(A, A')$$

is a connected directed a-cyclic graph, with a single root.

For instance, Fig 4.4 illustrates a play where $\rho = \{(Q, B), (Q, C), (B, D), (C, D), (C, F)\}$

Semantics of a Play

In order to define the semantics of a play we first extend the notion of an implementation to a whole run.

Definition 14 Let P be a play, and $A \in P$. An *implementation* of $\tau \in [\![A]\!]$ is a minimal set that contains an implementation of $tr_q \in \tau$ for every task q in A that has a refinement in P.

Finally, an admissible run of a play is defined as a set of runs that consists of a run of the root, and implementations that are added recursively.

Definition 15 Let P be a play with a root R. An admissible run of P is a minimal set of runs that contains a run $\tau \in [\![R]\!]$, and an implementation of every run in this set.

4.3 Composition of Acts

A *composition* is a structure of (possibly interacting) concurrent acts. It is specified in an act of a special form, as follows.

Act A **is**
composition $A_1, \ldots, A_k$
TimeBase Δ_A
End

Here, A and A_i are act signatures that lack input and output domains. (The **composition** section can be followed with the normal sections, in which case these are considered as a specification of an additional act in the composition list.)

Operationally, the acts A_i become active simultaneously with each activation of A (the host act). The acts of a composition can interact by referring to each other events. Syntactically, a system task declared in an act of a composition may be declared as an environment task in any of the other acts, enabling these acts to react to the events generated by that task. Also, the acts declared in a composition inherit the TimeBase declaration of the host act.

For instance, the following act specifies the behaviour of a train that obeys the commands of a signal.

```
Act Train_control is
   Tasks
      Environment Signal(on,off)
      System Train(move,stop)
   Reactions
      [ (startup ∨ Signal(off)) → Train(move) ] :Signal(on)
      [ Signal(on) → Train(stop)] :Signal(off) <0.5sec
End
```

Given Cross_control and Train_control, an act which controls the entire railway system can be defined as follows:

```
Act Railway_Control is
   Composition Train_control, Cross_control.
```

```
    TimeBase 0.1sec
End
```

Note that Signal declared in Train_control as an environment task, refers to the same task classified in Cross_control as a system task. Thus its activations by the crossing control would affect the train movement control.

4.3.1 Semantics of a Composition

Under the global clock activation policy, we consider a composition as an act each of whose sections is composed of the union of the corresponding sections in the acts that comprise the composition. However, environment tasks that have counterpart system declarations in some other constituents, are considered just once, as system tasks. Also, the compound act employes the host TimeBase declaration.

Formally, let Sys_i, Env_i, and Γ_i, denote, respectively, the sets of system tasks, environment tasks, and reactions in the act A_i. Then, the composition is defined as the act represented by

$$\{ \bigcup_{i=1..k} Sys_i\,, \quad \bigcup_{i=1..k} Env_i - \bigcup_{i=1..k} Sys_i\,, \quad \bigcup_{i=1..k} \Gamma_i\,, \quad \Delta_A \}$$

Thus, in fact composition is nothing but 'syntactic sugar' used to afford modular representation. Also, note that a composition can participate in a play as it can be considered a closed act that is terminated only by abortions of the refined task (as shown in Section 4.2.1).

We would like to point out that in a distributed environment, where each act is driven asynchronously by a local clock, composition is no more 'syntactic sugar' but an essential structuring means (see discussion in Section 6.5.2).

Chapter 5

System Development with MASS

In this section we illustrate a top-down activation oriented specification methodology. Basically, it suggests to start with a refinement of what seems to be the main course of the system behavior. Everything else, even though known to be part of the system, is considered part of the environment. Then, in succeeding iterations the structure is broadened by transferring environment activities into the system, and respectively refining their behaviors. At any stage of the development the specification is amenable to a simulation where the unspecified behavior is considered to be part of the environment.

An activity identified during a specification is represented in MASS by a *task*. In a development step a task is refined, either into a composition of tasks which represent a partition into concurrent activities, or by an act which specifies its design in terms of lower level activities represented by tasks as well.

In the case of refinement by a composition, we can independently proceed with any one of the constituting tasks. This is possible since MASS requires that all the information needed for interaction should be presented at this level. However, the partition need not be complete, as it will be shortly explained in the context of environment tasks.

In the case of refinement by a substantial act, the constituting tasks are classified into system and environment subsets, representing the activities which are meaningful at this level of abstraction. Respectively, we use *reactions* to specify the activation requirements for the system part.

We classify a task to be of a system type in case its activation is under the responsibility of the specified subsystem. Environment tasks, for which no activation requirements are specified, are either system tasks of another act (at the same, or higher, level of specification), or otherwise tasks whose execution is entirely external to the specified system. However, this observation is merely conceptual. Thus one may declare environment tasks, known to be part of another subsystem, even though the corresponding act has not been specified yet.

Each system task constitutes an abstraction level which can be further refined to a single, or a composition of acts. We proceed with that process until all system tasks are decomposed into basic components.

In succeeding iterations we go over each specification level, and refine it horizontally.

We can either expand unrefined system tasks, or define new acts where tasks, classified as environment in existing acts, are classified to be of system type. Thus, by composing the new acts with the already defined part of the system, we transfer them from the environment into the system.

In the following subsections, we illustrate this method by working out the specification of a car automatic cruise-control system.

5.1 Automatic Cruise Control

The Automatic Cruise Control (ACC) is intended to control the speed of a car according to driver instructions. The interface with the driver consists of a master switch (on/off), a 3-state command lever (decrease, maintain, increase), a 'resume' button, and the gas and brake pedals. The ACC takes over the speed control whenever the master switch is turned on, provided the car engine is working. The control is released either if the engine goes off, or the master switch is turned off.

Being activated, the ACC operates to maintain the car speed. It starts increasing (decreasing) the speed only when the command lever is moved from the 'maintain' to the 'increase' ('decrease') position (thus if the lever is not initially in the 'maintain' position, it must first be moved there). A return of the lever to the 'maintain' position will cause the system to maintain the new acquired speed. The whole control operation is immediately suspended in case a press, either on the brake or on the gas pedal, is sensed. In this case, upon a press of the 'resume' button the control is resumed to maintain the suspended speed, (provided the brake and gas pedals are not pressed).

This example has been chosen since it is extensively worked out in the literature concerning real time system specification [39, 10, 11, 12, 20, 37, 40]; thus it can serve as a case study for a comparison of different approaches.

5.2 Specification in MASS

At the top level, the act **Control_ACC_Operation** considers the control operation, denoted by the task **CruiseCrl**, as an atomic activity.

Act Control_ACC_Operation is
  Tasks
    Environment Engine:{ off, on }
    System CruiseCrl:{}
  Reactions [Engine=on $\rightarrow$ CruiseCrl] : Engine=off
  TimeBase 0.01sec
End

CruiseCrl is designed as a non-terminating operation (indicated by using empty braces, {}, to denote its output domain), thus it must be externally aborted in order to be stopped.

The associated reaction specifies this task to be activated whenever the engine is turned on, and respectively aborted when the engine is turned off; namely the task is active as long as the engine is operating. The engine, represented by the task **Engine**, is declared as an environment task, since at this stage we are not concerned with how exactly the engine is monitored. Only its events which influence the control operation are of interest. At a later stage, we can independently (in a different act) specify the activation requirements for the engine (as indeed we do), and compose them with the control operation.

As this is the top level act in the system specification, the time-base is explicitly specified. (Recall it also serves as the time-base for all descendent acts, therefore the TimeBase declaration will be omitted in their specifications.)

We continue the specification by a refinement of **CruiseCrl** given as follows.

Act CruiseCrl
  Tasks
    Environment DriverCmd : { start, stop }
    System AutoCruiseCrl:{}
  Reactions [DriverCmd=start $\rightarrow$ AutoCruiseCrl] :DriverCmd=stop
End

At this level the control operation is represented by a pair of tasks.

- An environment task, **DriverCmd**, which identifies the driver commands to start and stop the control process (again its concrete specification is deferred).
- A system task, **AutoCruiseCrl**, which manages the automatic speed control in response to a proper driver command.

The task **AutoCruiseCrl** is further refined into the following tasks:

- A system task **ActiveCrl** which accomplishes the actual control operation.
- A system task **SaveSpeed** that upon activation computes the current speed. It is intended to record the car speed to be maintained by **ActiveCrl**.
- The environment tasks **Brakes, Gas, Resume** which determine the conditions for activation and abortion of **ActiveCrl**. These conditions are expressed by the virtual task **CruiseState** that represents the whole process as a two-state (active, suspend) automaton.

The first reaction handles the need to record the car speed that would be used as the command for the control process whose activation is monitored by the second reaction. The operator BT (pronounced *becomes-true*) is defined as follows:

$$BT(\alpha,\beta) \stackrel{def}{=} (\alpha \succ \beta) \vee (\beta \succ \alpha)$$

In our case, we need it to guarantee that the speed is recorded even if the process is reactivated immediately after being suspended.

Note that this specification assumes that the brakes are not pressed as the process starts, or resumed. We could however make it explicit in the specification, or design the reaction to consider such occurrences.

```
Act AutoCruiseCrl
  Tasks
    Environment  Brakes  Gas  Resume
    System  ActiveCrl:{}  SaveSpeed
    Virtual
      CruiseState : { active, suspend }
      where CruiseState=suspend  at  (CruiseState=active ≻ (Brakes ∨ Gas))
            CruiseState=active   at  (Startup ∨ (CruiseState=suspend ≻ Resume))
  Reactions
    [ (startup ∨ CruiseState=suspend) → SaveSpeed ] <0.1sec
    [ BT( SaveSpeed, CruiseState=active ) → ActiveCrl ] :CruiseState=suspend
End
```

Next we refine **ActiveCrl**. It consists of the environment task **CruiseCmd** which designates changes in the lever position, and system tasks **MaintainSpeed, DecreaseSpeed, IncreaseSpeed** which implement the control operations corresponding to the current lever position. Note that upon returning to **MaintainSpeed** after the speed was changed we need first to update the reference speed (by **SaveSpeed**). We specify this requirement in a special reaction (the first). Alternatively, we could use the reaction

```
[ CruiseCmd=maintain → (SaveSpeed;MaintainSpeed ] :ChangeSpeed
```

but in this case we cannot specify a specific deadline for the execution of **SaveSpeed**.

```
Act ActiveCrl ( Speed )
  Tasks
    Environment  CruiseCmd : { decrease, maintain, increase }
    System  SaveSpeed  MaintainSpeed  DecreaseSpeed  IncreaseSpeed
    Virtual ChangeSpeed
            where  ChangeSpeed  at  (CruiseCmd=decrease ∨ CruiseCmd=increase)
  Reactions
    [ CruiseCmd=maintain → SaveSpeed ] <0.1sec
    [ (startup ∨ SaveSpeed) → MaintainSpeed ] :ChangeSpeed
    [ CruiseCmd=decrease → DecreaseSpeed ] :CruiseCmd=maintain
    [ CruiseCmd=increase → IncreaseSpeed ] :CruiseCmd=maintain
End
```

The concrete behavior of the control operations is given by the refinements below. Note that **MaintainSpeed** hides a number of data flow connections. For instance, the speed

recorded by **SaveSpeed** is probably used in the computation of **ComputeSpeedError**. Furthermore, the error produced by **ComputeSpeedError** is used in **SetThrottle**. Although we can specify data transfer in MASS (see Section 2.0.6) it seems to be not suitable for situations like these.

Also, **MoveThrottle** is declared to be of system type both in **IncreaseSpeed** and **IncreaseSpeed**. This is legal and should cause no confusion, even if it could be activated concurrently due to reactions in separate acts.

Act MaintainSpeed
  Tasks
    System
      ComputeSpeedError SetThrottle
  Reactions
    [periodic(1sec) → ComputeSpeedError] ≤20ms
    [periodic(100ms) → SetThrottle] ≤20ms
End

Act DecreaseSpeed
  Tasks
    System MoveThrottle(Steps)
  Reactions [periodic(100ms) → MoveThrottle(-3)] ≤20ms
End

Act IncreaseSpeed
  Tasks
    System MoveThrottle(Steps)
  Reactions [periodic(100ms) → MoveThrottle(3)] ≤20ms
End

At this stage, after the main course of the system operation had been specified, we elaborate declarations of environment tasks, and combine them into the entire system specification. Starting at the top level, we specify an act **EngineMon** which defines the activation requirement for the task **Engine**.

Act EngineMon
  Tasks
    System EngineState:{ off, on }
    Virtual Engine:{ off, on }
        where Engine=off *at* (Startup ∨ EngineState=on≻EngineState=off)
            Engine=on at (EngineState=off≻EngineState=on)
  Reactions
    [periodic(20ms) → EngineState] < 10ms
End

This specification presents a possible implementation based on polling. The task **EngineState** continuously samples the state of the engine, and **Engine** is a virtual task intended to detect the changes in the state of the engine operation. Note that, once we decide to modify the implementation, for instance by using interrupts instead of polling (see below), this would not affect the acts using the task **Engine** (in fact it would not be known there).

In order to compose **EngineMon** with **Control_ACC_Operation** we add an upper level declaration,

Act ACC is *composition* Control_ACC_Operation EngineMon *End*

To complete the specification we define the act **DriverCmdMon** which establishes the activation requirement for the task **DriverCmd**.

```
Act DriverCmdMon
  Tasks
    Environment MasterSwOff, MasterSwOn
    Virtual DriverCmd : { start, stop }
          where DriverCmd=stop at MasterSwOff
                DriverCmd=start at MasterSwOn
End
```

Here, we prefer an implementation by interrupts (mentioned above). The environment tasks **MasterSwOff** and **MasterSwOn** represent hardware interrupts generated upon turning the master switch off and on, respectively. **DriverCmd** is declared as a virtual task which provides an abstraction of the interrupts which represent the master switch as an object. Practically, the entire act is superfluous since we could directly use **MasterSwOff** and **MasterSwOn** in **CruiseCrl**. However, it does have considerable benefits in the context of a whole system development and maintenance, as it supports good software engineering practice.

Finally, the composition of **CruiseCrl** and **DriverCmdMon** is explicitly stated by extending the specification with a declaration,

Act CruiseCrl is *composition* DriverCmdMon *End*

Note that we have used **CruiseCrl** to name both the act representing the composition and a constituent of the composition. That is since we have no proper 'composition' stub to host **DriverCmdMon**, and it is absolutely undesirable to modify the whole structure in order to insert one.

This situation represents a conflict which often occurs in system development through hierarchical decomposition. In general it is characterized by having a number of subsystems which formally should be presented at the same abstraction level. However, they are not equally significant in the specification of the system behavior at this level. Thus using a single stub where the system is described as a composition of all of its subsystems, and then refining each one separately seems an unnatural representation which blurs the whole picture. Instead we suggest a form of the *composition* construct which emphasizes the subsystem which seems to be central (in the developer view), and does not force a designer

to have a complete view of the system structure at an initial stage.

Chapter 6

Discussion

In this section we discuss design decisions made throughout the definition of MASS, and provide their rationale.

6.1 Synchronous Activation of Asynchronous Tasks

Operationally, a specification in MASS is executed periodically – at a user defined rate — computing the events occurring at each cycle and required reactions (activation and abortion of tasks' executions).

The semantics of MASS relies on the essential assumption that the computation, carried out at each cycle, necessarily terminates within the cycle period (the period corresponding to the execution rate). This assumption characterizes MASS as a *synchronous* language [6]. However, in contrast with the synchronous model, the activated tasks are considered to behave asynchronously, as their executions may take a non-zero (albeit bounded) duration.

Comparing with the pure synchronous model, we believe our approach is more realistic since it restricts the validity of the synchronous hypothesis to a very special class of computations, for which it seems reasonable (see below). On the other hand, it preserves asynchronous computations as the specification primitive, and keeps them under the control of the system.

6.1.1 Design Decisions

MASS employs a pair of design decisions that make the synchrony assumption reasonable.

- Basic events occurring during a cycle are considered only at the next cycle. Consequently, they do not affect the ongoing computation of reactions. This approach also avoids endless circularity, a problem often encountered in the synchronous model.
- We restrict failing events to denote finite durations, or no bound at all. A bounded reaction ensures that the number of ongoing executions of the activated task (and

therefore the number of executions that must be monitored for abortion) is bound, as well.

Unfortunately, the bound does not exist for unbound reactions. However, unbound reactions improve the expressiveness of a specification to a large extent (for instance, consider the task **ActiveControl** in the Cruise Control example). Fortunately, we were able to observe that in all worked out examples such reactions never lead to concurrent activations. This is not a formal claim, but we could require that whenever the reaction is unbound the activating event should be formulated to prevent concurrent executions, even if they could occur. For instance, it can be achieved by transforming a reaction $[\alpha \ \rightarrow \ q] : \varepsilon$ into the form:

$$[(\text{startup}; \text{first}(\alpha)) \vee ((\alpha; \text{first}(\alpha)) \wedge \Diamond(1; q)) \ \rightarrow \ q] : \varepsilon$$

6.1.2 Resolving Inconsistency

In order to avoid inconsistency of events identification (as illustrated in Section 2.0.3), we disallow abortion events $q!$ to participate in aborting events. This problem is well known in synchronous languages [23]. In fact, we could replace the general constraint by a compilation time analysis that would detect inconsistent specifications (as done in ESTEREL, for instance), but we preferred the convenience of a general constraint since it seems to have no practical difference.

6.2 Responding Activating Events

By our semantics a new incarnation of a task is activated with each occurrence of an activating event. Thus it is possible that different incarnations of a task are executing concurrently (if an activating-event occurs while a task is executing in response to a previous occurrence). However, in order to allow an external observer to pair an activating-event with its effect (the concrete response), we assume that with every execution of a task. the system scheduler records the information concerning the occurrence of the activating event, and returns it together with the result of the task execution. A number of alternative design decisions could be taken:

- Ignoring further occurrences of the activating events while a task is still executing. Such a policy would simplify the semantics and implementation of MASS to a great extent. However, it conflicts with the basic definition of a real time systems, since it allows activation requirements not to be respected. Therefore, it must be rejected.

- Occurrences of the activating events while a task is still executing are respected by the result of the ongoing computation. This approach might lead to erroneous implementations since the computation of a real-time function must responds to the state of the environment at the activation time. Therefore, the output of an executing function

can not be considered as a correct result with respect to activations requested during its execution.

- Further activations are delayed and serialized due to a certain criterion. This policy is a special case of the semantics taken in MASS. As a general approach, it might lead to inefficient scheduling, as computations are not preempted.

Note that in a pure synchronous approach the problem does not exist since computations take a zero duration.

6.3 Missing Deadlines

Recall that MASS allows the specification of two kinds of deadlines with respect to an activated task, an aborting deadline and a failing deadline.

Missing an aborting deadline results in a special 'abort' event. It is assumed that the execution of the activated task is indeed aborted. (Our semantics rejects a run in which a task responds after the occurrence of the aborting event.

On the other hand, a failing deadline is considered as a safety property, something that should never happen. Therefore, no reaction is specified in response to its occurrence. In practice, since this event is detectable, various strategies can be taken. For instance, it can be used to cause a partial or general system reset.

It is worthwhile mentioning that the mechanism of aborting events would be superfluous in a pure synchronous approach. However, even in the asynchronous model, it could be eliminated if one assumes a complete pre-run (feasible) scheduling is possible [41], and computations always terminate. In practice, this is rarely the case, and we do need to handle exceptions. In addition, this mechanism allows to design a system with (intentionally) non self-terminating functions. Thus, it is possible to express a requirement like "the train keeps moving until a 'stop' signal is observed".

6.4 Minimal delays

Most languages allow the specification of minimal and maximal delays for the occurrence of an event. MASS, in comparison, supports maximal delays, solely (by constricting-events).

We claim that there is no point in the specification of a minimal delay, in the case of asynchronous operation, since it imposes strict constraints on the implementation of the generating function (even the speed of the CPU must be taken into consideration).

Instead, we support a more realistic interpretation of a minimal delay, as a delay for activation of the function which generates the event; a requirement which can be explicitly expressed in MASS by an event of the form $(\alpha; r)$. However, note that in *synchronous* languages [6], where a computation takes zero time, the two interpretations coincide.

6.5 Acts Synchronization

The semantics given for refinement is based on the notions of a global clock and synchronized observation sequences. This approach arises a number of questions that are discussed below.

6.5.1 Single TimeBase

As emphasized in Section 4 it is desirable that subsystems composed into a whole system retain their basic properties. In a play the refining act inherits the time-base of its host which may turn to be different from its local declaration. So the question arises, what properties of the act are changed when the time-based is modified.

We did not examined this problem formally, however it is clear that increasing the time-base would change time-oriented responsiveness properties. Namely, since the time-base mainly determines the delay in which events are observed, a property like "β is always observed 20ms after each occurrence of α" may be proved true if the time-base is 2ms, but is certainly not satisfiable if the time-base is increased to 30ms.

On the other hand, shorting the time-base probably improves responsiveness, but does not falsify interesting properties proved for the original time-base. (We rule out properties concerning minimal delays that are achieved by employment of a coarse time-base.) It seems to us that given a property it is possible to determine a maximal timebase such that every shorter time-base will not affect the validity of the property. (One can define the maximal frequency, and employ Shanon theorem [].) Therefore, constructing a play, it seems a good practice to fix the play time-base to the minimal time-base of the constituting acts.

Alternatively, using the *multi-clock* approach employed in LUSTRE [21], we could compose acts under refinement while retaining the local time-bases. In fact, this is a private case of the asynchronous clock semantics discussed in the next subsection.

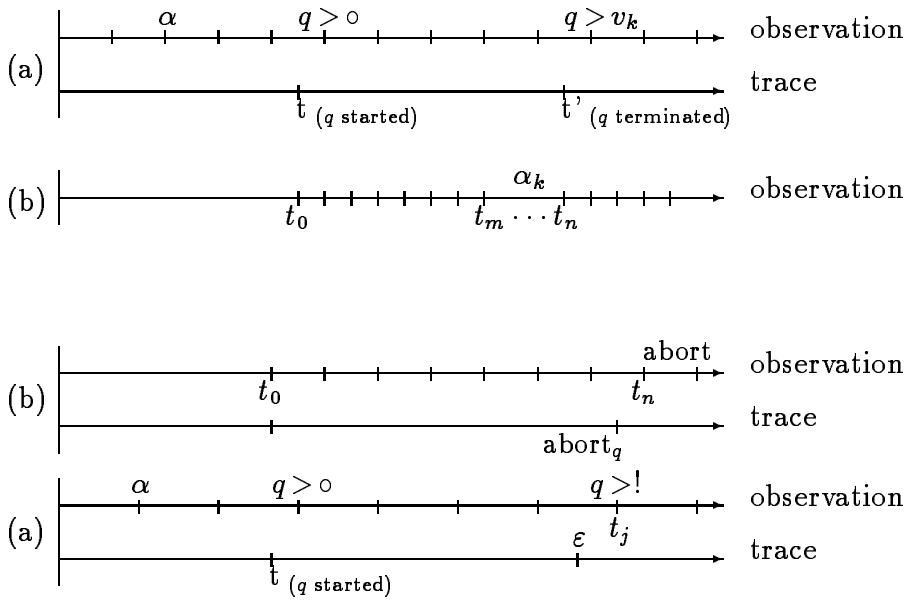
6.5.2 Distributed Systems

We discuss two essential properties of distributed systems, regarding their representation in MASS.

The property of *events propagation* refers to the fact that in a distributed system an event is observed with different delays at the various subsystems (asynchronous message passing). This effect can be easily expressed in the global clock semantics of MASS, by simulating the propagation of events with tasks.

The property of *asynchronous clocks* indicates that each site in the system is driven by a local clock. In general, the clocks are not synchronized, each running at a different rate. In order to represent this property, we have defined a distributed semantics for a play. This semantics allows each act to be interpreted according to its local time-base, and the interaction between acts does not assume any synchronization of execution steps. However, it ignores the problem of clocks drift which is an essential phenomena in distributed systems. The distributed semantics will be presented in a future report. However, the figures below

present the flavor of runs under the distributed model. These should be compared with Figures 4.2 and 4.3.



It is worthwhile mentioning that in this case *composition* is no more syntactic-sugar but becomes an essential primitive concerning structures of acts.

6.6 Expressiveness

MASS suggests an alternative software architecture. A process in the traditional sense is represented in MASS as a set of computations, each corresponding to a sequential chunk in the original process. The various synchronization constructs that are dispersed within a traditional process are replaced by acts that are separately specified, and not necessarily structured according to the original partition into processes.

In such an architecture the system functions are actually represented twice, in two independent specifications where modules cohesion is determined due to a data-oriented and a temporal scope of operation criteria, respectively.

In general, due to the separation of concerns, this approach improves a specification expressiveness to a large extent (in comparison with CSP-based architectures). Nevertheless, it must be mentioned that there are certain applications for which the CSP paradigm provides a much more natural representation. Mainly, when real distributed systems are considered (in the sense of cooperating agents), but also in particular applications where a process consists of a strictly ordered sequence of communications.

A second expressiveness issue is concerned with the declarative representation style employed in MASS. In general, declarative representation is considered a higher level of abstrac-

tion than procedural representation. Our experience indicates that MASS specifications are especially expressive regarding systems in which a large number of events are likely to occur non-exclusively, and activations depend on temporal patterns of a number of events. Typical applications of this kind such as an automatic assembly line (see below 7.2), a multi-sensor tracking system, an automatic pilot, and subsystems management in a communication satellite, were successfully specified in MASS.

However, we realize that MASS is not the natural formalism to express all parts of an application. For example, for state-based logical processes, state-diagrams are probably much more communicative.

Chapter 7

Practical Experience

7.1 Development System

We have constructed a basic development environment for an earlier version of MASS (without refinement). The environment consists of a compiler, a run-time system, and a simulation package.

The run-time system is designed in an object-oriented fashion (implemented in C++). The top level class is "reaction". Its state consists of instances of class "event" corresponding to the events that control a reaction operation. Thus an act is compiled into an initialization procedure that generates instances of class "reaction". The run-time skeleton and the initialization procedure are then compiled into the target code, and linked with the object code of the functions that implement the computational part of the tasks.

The active part in the run-time system consists of two parts: a synchronous executive, and a multitasking kernel. The executive, excited by ticks of a real-time clock, activates the reaction objects (a class method). Each reaction is responsible for the control of the executions of its response-task . It identifies the occurrences of relevant events in accordance with the execution model (cf. Section **??**), and sends corresponding scheduling and aborting messages to the multitasking kernel. All the reactions operate on a common input of basic events that is generated between ticks by the multitasking kernel, and external interrupts. The multitasking kernel operates asynchronously with the executive. It is driven by scheduling messages, and terminations of task executions that are in general scheduled according to an earliest-deadline scheduling algorithm [17].

The simulation package allows one to include in a system tasks whose computation has not been specified. Such tasks are identified at the link phase, and the simulation package is directed to generate basic events corresponding to the their declarations, at random delays with respect to each activation. (For environment tasks it provides a simulation of the physical environment.) We use the simulation package to develop a system by "incremental specification", where one starts with all tasks unspecified and then gradually replace them by corresponding procedures, or refining acts (which in turn add new unspecified tasks).

7.2 Case Study

In order to study the adequacy of MASS for practical specifications we carried out a case study of a large scale system. We specified the operation of an automatic assembly line that consists of trays moving between stations on a multi-way conveyor. Each station is equipped with one or more general purpose (4DOF) robot cells, which communicate and work together to retrieve parts from an automatic storage facility and assemble them into products. (A full description is given in [42]).

This application is large enough to get a significant insight of the language performance. (It must be mentioned, however, that this is not a perfect real-time example since no timing constraints were actually required.) In general, the results were very satisfactory, and encourage the introduction of MASS in real industrial applications.

Chapter 8

Conclusions

We have presented MASS, an activation oriented specification language for real-time systems. The salient properties that impart MASS with its special expressiveness capabilities, are:

- It provides for an explicit specification of causality and temporal relations that constitute a real-time behavior.
- It semantically relates computations and the events that are likely to be generated by them.
- MASS execution model provides for maximal concurrency. It allows concurrent execution of different tasks, as well as multiple incarnations of a single task.
- MASS suggests a software architecture that supports the separation of concerns regarding the activation of computations, and the data transform algorithms they execute.

However, MASS is not equally capable for all kinds of applications. In particular, it does not seem to be the natural formalism to describe state-based logical processes, in which case Statecharts (cf. SectionA) is probably better. In the spirit of [22], we can envisage a paradigm that combines both languages by using acts to specify the operation of a system within a state. It seems likely that a Statecharts specification can be mechanically translated into a structure of acts. Thus, we might be able to establish the whole specification on a unified formalism.

The next step concerning this study is the development of a reasoning system. In general our aim is to construct a user assistant method based on the work of Rich an Feldman [36]. At present we study a number of formalisms, both automata and temporal-logic based.

Appendix A

Survey of Real-Time Specification Languages

In what follows, we survey a number of languages representing the main approaches in RTS specification. It must be mentioned that most of the languages have associated verification tools. These will not be described here since in this report we are concerned only with the language representation and expressiveness.

Statecharts

Statecharts [24] is probably the most notable example. It represents a system (visually) as a hierarchical structure of (possibly concurrent) automata, communicating by broadcasting global events.

Statecharts is a synchronous language. Asynchronous computations are controlled by special events, such as: *Start(f)* and *Suspend(f)*. However, it is not possible to relate events to the computation of f, therefore the language bears the basic disadvantage of synchronous languages in general. (It is possible to model the execution of an asynchronous computation with special states ("suspend","active" etc.), but this is obviously impractical since it would lead to an enormous number of "technical" states, making a specification unreadable.)

Our experience indicates that Statecharts is highly suitable for the top level specification of complex reactive systems, but it becomes quite unnatural for a specification of the operations that constitute a state. Also, Statecharts is often criticized for its liberal usage of global events broadcasting. This is considered dangerous from a software engineering perspective (in a large specification it is hard to locate the source of events and conditions specified to cause a certain transition). Argos [31] (a derivative of Statecharts) presents a mechanism for localizing the effect of events. Also, Statemate (a CASE tool employing Statecharts) suggests a method of integration with data-flow charts, by treating the reactive behaviour as a special computation, in the functional framework.

ESTEREL and CRP

ESTEREL [8] is an early example of a synchronous language. The basic actions in ESTEREL specify instantaneous emission of signals in response to incoming signals, and (possibly constrained) waiting for a signal occurrence. Basic actions are combined by conditional, sequential, delayed-loop (with watchdog), and parallel constructs, into a reactive behavior specification.

ESTEREL allows the specification of synchronous computations only. However, recently Berry et al.[9] introduced CRP, a paradigm that combines synchronous and asynchronous computations. Essentially, they extend ESTEREL with a special construct, "*exec L:P*", which implicitly associates the label L with the events (signals) denoting the start, termination, and abortion of an execution of P. This is an alternative to our approach, as indeed it remedies the problem of tracing causality with respect to asynchronous computations. However, its expressiveness is restricted, since once engaged with an execution, a program cannot respond to additional activations. For instance, if P is an asynchronous computation, a requirement like "activate P upon every occurrence of the event α" is not expressible in this formalism.

LUSTRE

LUSTRE [21] is an example of a synchronous declarative language. A specification is presented as a set equations '**X**=**E**' where **X** is a variable and **E** is an expression involving variables, standard operators, and a special operator which provides the previous (time-instant) value of an expression. Variables are computed every clock 'tick', representing the sequence of values they take during an execution. A variable in an expression can be expanded into an another expression or otherwise is considered as an environment (sampled) datum. A value assignment to a variable is considered an event, thus events occurrences are actually broadcast. However, it is possible to restrict the effect of an event by organizing specifications into structures of, so called, Nodes.

Time is not explicitly represented in Lustre but rather considered an environment variable (the interpretation of time is given by the user). Thus in order to reason about real-time progress one needs a special set of equations defining its nature.

ATP

There are a number of process algebra languages intended for RTS specifications. The most suitable seems to be ATP [33] which augments CSP with special time variables and the activation constructs: Timeout and Watchdog (other extensions of CSP [35] rely on a special 'delay' construct for the specification of time progress).

ATP employes a (dense) time domain to represent an execution of a process. In the language actions can be specified as instantaneous, or time consuming, and there are two time operators. One is 'P **timeout***(d)* Q' which behaves as P if P performs an action within a duration of d time units; otherwise it behaves as Q. The second is 'P **watchdog***(d)* Q'

behaves as P during time equal to d. At time d, P stops and Q is started. There is also a **cancel** action which when executed by P cancels the watchdog effect.

Communication in ATP is point-to-point (as in CSP). Broadcasting (which often occurs in RTS) must be programmed as a set of independent parallel communications, but can not be explicitly specified.

Petri-nets

Petri-nets provide for modeling, graphically, the activation of atomic actions and are associated with powerful analysis techniques. This formalism is recognized as highly suitable for the representation of special classes of synchronization problems (mutual-exclusion, handshaking etc.). However, its expressive power is limited with respect to real-time requirements since it is impossible to specify an order of tokens arrivals and dependency on past transitions. There are a number of extensions ([28, 16]) intended for the incorporation of timing constraints into a net specification; however, the most general one seems to be given by TB-nets [18].

In TB-nets tokens in a marking are all designated by their creation time. Transitions are annotated with a timings relation over the tokens in the input and output places. Such a relation determines a time-interval during which an enabled transition may fire (has to fire in a 'strong' time semantics). The output tokens are all assigned with the actual firing time. Mostly, this formalism provides an expressive representation of real-time specifications. However, the fact that petri-nets are in general history insensitive (like any other finite state automata), make solutions to history dependent problems very cumbersome or even not expressible (for instance if an action is required to be executed upon each occurrence of an event α ever after an event β had occurred). It must be noted that TB-nets are derived from a more general formalism, ER-nets, which allows specification of timings constraints which depend on functional attributes of a system. In particular, ER-nets allow transition relations associated with assignment of general variables; thus one can record history by special variables. Again, with this techniques, specifications seems to be expressed by programming solutions, rather then explicit requirements.

Bibliography

[1] M. Abadi and L. Lamport. Composing specifications. In *Stepeise Refinement of Distributed Systems, LNCS 430*, pages 1–41. Springer-verlag, 1990.

[2] R. Alur and T.A. Henzinger. A really temporal logic. In *30th Annual Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[3] R. Alur and T.A. Henzinger. Real-time logics: Complexity and expressiveness. Technical Report TR STAN-CS-90-1307, Stanford University, 1990.

[4] R. Alur and T.A. Henzinger. Real-time logics: complexity and expressiveness. In *5th IEEE LICS*, pages 390–401, 1990.

[5] R. Alur and T.A. Henzinger. Logics and models of real time: A survey. In *REX Workshop Real-time: Theory and Practice, LNCS 600*, pages 74–106, The Netherlands, June 1991. Springer-verlag.

[6] A. Benveniste and P. Le Guernic. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):192–230, August 1992.

[7] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The signal language and its semantics. Technical Report 459, IRISA, 1989.

[8] G. Berry and G. Gonthier. The ESREREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming, 19*, pages 87–152, 1992.

[9] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating reactive processes. In *20 Annual ACM Symposium on Principles of Programming Languages*, Charleston, Virginia, 1993.

[10] R. Blumofe. Executing real-time structured analysis specifications. *ACM SIGSOFT Software Engineering Notes*, 13(3):32–44, July 1988.

[11] G. Booch. Object oriented development. *IEEE Trans. on SE*, SE-12(2):211–221, February 1986.

[12] J. W. Brackett. Automobile cruise control and monitoring system example. Technical Report TR-87-06, Wang Institute of Graduate studies, 1987.

[13] R. Budde and K.H Sylla. Object oriented design for embedded systems. Technical Report TR-87-06, GMD, 1996.

[14] N. Carriero and D. Gelernter. Coordination languages and their significance. *Comm. ACM*, 35(2):97–107, February 1992.

[15] Z. Chaochen, C.A.R. Hoare, and A.P. Raven. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.

[16] J. Coolahan and S. Roussopoulos. Timing requirements for time driven systems using augmented petri nets. *IEEE Trans. on SE*, SE-9(5):603–616, September 1983.

[17] V. Gafni. A tasking model for reactive systems. In 10^{th} *Real-time Systems Symposium*, Santa Monica, California, December 1989.

[18] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A unified high-level petri-net formalism for time critical systems. *IEEE Trans. on SE*, SE-17(2):160–172, February 1993.

[19] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio: A logic language for executable specifications of real time systems. Technical Report TR 89 006, Politechnico di Milano, 1989.

[20] H. Gomma. Structuring criteria for real-time system design. In *11*th *Int. Conf. on SE*, pages 290–301, Pittsburgh, PA, May 1989.

[21] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language lustre. *IEEE Trans. on SE*, SE-18(9):785–793, September 1993.

[22] D. Harel. Biting the silver bullet: Towards a brighter future for system development. *Computer*, 25(1):8–20, January 1983.

[23] D. Harel. On the formal semantics of statecharts. In *2nd IEEE Symp. on Logic in Computer Science*, pages 54–64, New York, 1987.

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

[25] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *5th IEEE LICS*, pages 402–413, 1990.

[26] F. Jahanian and A.K. Mok. A graph theoretic approach for timing analysis in real-time logic. In *Real-time Systems Symposium*, pages 98–108, Louisiana, December 1986.

[27] F. Jahanian and A.K. Mok. Safty analysis of timing properties in real time systems. *IEEE Transactions on Software Engineering*, 12(9), 1986.

[28] K. Jensen. Computer tools for construction, modification and analysis of petri-nets. In *LNCS 255*, pages 4–19. Springer-Verlag, 1986.

[29] R. Kurki-Suonio and H.M. Jarvinen. Action system approach to the specification of distributed systems. Technical report, Tampre University of Technology, Software Systems Lab, 1989.

[30] H. Langmaack, W.P. de Rover, and J. Vytopil. *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag, 1994.

[31] F. Maraninchi. The argos language: Graphical representation of automata and description of reactive systems. unpublished.

[32] B. Moszkowski. *Reasoning about Digital Circuits*. PhD thesis, Stanford University, 1983.

[33] X. Nicolin and J. Sifakis. The algebra of timed processes atp: Theory and application. Technical Report RT-C26, LGI-IMAG, France, 1990.

[34] J.S. Ostroff. Verification of safety critical systems using ttm/rttl. In *REX Workshop Real-time: Theory and Practice, LNCS 600*, pages 573–602, The Netherlands, June 1991. Springer-verlag.

[35] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. In *LNCS 226*, pages 314–323. Springer-Verlag, 1986.

[36] C. Rich and Y. A. Feldman. Seven layers of knowledge representation and reasoning in support of software development. *IEEE Trans. Software Engineering*, SE-18(6):451–469, 1992. Special Issue on Knowledge Representation and Reasoning in Software Development.

[37] S.L. Smith and S.L. Gerhart. Statemate and cruise control: A case^2 study. In *COMPSAC '88*, pages 49–56, Chicago, IL, October 1988.

[38] C. Tofts and F. Moller. A temporal calculus of communicating systems. In *LFCS*, pages 89–104, 1989.

[39] S. Tyszberowicz. *OBSERV: A prototyping Language and Environment*. PhD thesis, Tel-Aviv University, 1990.

[40] P.T. Ward and S.J. Mellor. *Essential Modeling Techniques*, volume 2 of *Structured Development for Real-time Systems*, chapter Appendix A - Cruise Control, pages 67 – 103. Tourdon Press, Englewood Cliffs, NJ, 1985.

[41] J. Xu and D.L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Trans. Software Engineering*, SE-19(1):70–84, January 1993.

[42] G. Zehavi. Specification and implementation of robotic applications using mass, 1993.