Real-time specification and modeling with joint actions

Reino Kurki-Suonio and Kari Systä
Tampere University of Technology, Software Systems Laboratory, Box 553, SF-33101 Tampere, Finland

Jüri Vain
Estonian Academy of Sciences, Institute of Cybernetics, Akadeemia tee 21, EE-0108 Tallinn, Estonia

Abstract

The notion of joint actions provides a natural execution model for a specification language, when temporal logic of actions is used for formal reasoning. We extend this basis with scheduling, the role of which is to enforce liveness properties and to introduce real-time properties. This is done in a way that agrees with the partial-order view of computations and can be applied already in the early stages of specification and design. This leads to distinguishing between schedulings that are totally correct, partially correct, or incorrect with respect to liveness properties. A general scheduling policy of durational actions is formulated from which any reasonable scheduling can be obtained by reducing its nondeterminism. When this policy is totally correct for a system and gives the required real-time properties, no special limitations are imposed on the implementation. The approach also leads to a general classification of real-time models according to the permitted interactions between the computational state and real time.

1. Introduction

Specification needs levels of abstraction to allow separation of concerns in dealing with different properties. The selection of one’s first concerns depends on what is considered to be of primary importance, and on what one’s languages allow to be
conveniently expressed. In this paper we address the problem of providing a rigorous basis for tools with which executable models of distributed real-time systems can be developed in a well-structured manner.

In general, real-time requirements can be classified into statistical properties that must be satisfied with a given probability, and absolute requirements that must be satisfied by all computations. The latter are essential for safety-critical systems and should be verified with proof techniques. For a general discussion on verifying properties of safety-critical systems, see [28].

It is not possible in this paper to give a comprehensive survey of different approaches to real-time specification and verification; representative examples can be found in [11, 17, 25, 30, 32, 36], for instance. We distinguish, however, between two main possibilities for the first abstractions in real-time specification. One can start with a simple approximation of logical computations and concentrate on timing and scheduling issues. Alternatively, one can first abstract away timing and start with a specification of logical computations. The expression logical computation is used here to denote a reactive computation without any measure of time but with some ordering relation between its events.

Pure timing and scheduling specification, as a rule, largely ignores state information. (For a survey and list of references, see [36].) Therefore, it can only approximate causal relations between events, and is at its best when the tasks to be scheduled are independent and periodic. Introduction of metric time requires, however, evaluation of several aspects concerning the topology of the time domain, such as continuity, boundedness, linearity, etc., and answering the question of how the events in logical computations are related to this domain.

The alternative view of concentrating on logical computations is taken, for instance, in temporal logic [26, 33] and in process algebraic approaches [10, 16, 31]. One's first concern is then that computations be logically correct under all possible timings. Also in this case the formalism may enforce a particular kind of approximation of system state. Absolute real-time properties can be imposed on logical computations by extending system state with time [15, 27], but statistical properties are usually inexpressible; an exception to this rule is [14], for instance. In general, this group of approaches seems to have advantages over scheduling approaches when the tasks to be scheduled are "sporadic", and behavioral complexity is the main issue.

Within this wide spectrum, this paper falls in the latter category. We use the notion of joint action systems [6-8, 22], for which temporal logic of actions [26] is a natural logic for expressing and proving properties. Here we keep, however, to an intuitive, operational model of computations. For specification the underlying paradigm has the advantage that explicit process communication mechanisms are replaced by an abstract notion of distributed cooperation. For real-time modeling it is also significant that joint action systems are closed systems, in which both the system itself and its environment are included. Timing and expression of real-time properties then relate to both in a symmetric manner.
The motivation of the paper can be characterized as follows. Our main concern is to specify computations in an executable modeling formalism that allows realistic description of an arbitrarily complex system state, and which is suited for

- expressing and proving non-statistical properties in logic [26],
- modularity that supports stepwise refinement and modular proofs [20], and
- graphical visualization and validation with animation tools [35].

Rather than dealing with these issues here, we start with a basic formalism that has proved useful in such respects for logical computations in distributed systems, and address the additional problem of time, extending the approach from logical to real-time computations. In contrast to previous approaches where state-modifying actions are instantaneous [15, 27], our approach leads to durational actions.

The structure of the rest of the paper is as follows. Joint action systems are introduced in Section 2. In Section 3 we extend them to real-time modeling, define what we mean by scheduling, and analyze how liveness requirements can be implemented. Section 4 is devoted to real-time properties; a comparison to more conventional ideas of scheduling is also given there. Section 5 gives some concluding remarks, and introduces a hierarchy of real-time models with increasing complexity in dealing with real time.

The paper is a revised and extended version of [23]. Its main contribution is in providing foundations for a design method that covers both logical and real-time properties and has a simple and uniform logical basis. The basic ideas have been presented also in [24] from the viewpoint of scheduling principles, and their consequences on design methodology have been discussed in [21]. The joint action basis presented in Section 2 has been applied in an experimental specification language Disco (for Distribute Cooperation) [18, 19, 35]; the associated tools are currently being extended with additional support for dealing with real-time properties.

2. Joint action systems

2.1. Basic definitions

The operational model of joint action systems is an *interleaving model* that is directly based on transition systems as presented in [33], for instance. Such a model is state-based, as opposed to the event-based approach of process algebras; the current state will be denoted in the following by $s$.

A joint action system involves a set $X$ of *objects*. Each object $x \in X$ has a *local state* denoted by $s.x$. The local state $s.x$ consists of *program variables* that are local to object $x$ and whose values may change during a computation. For any collection of objects $Y \subseteq X$, $s.Y$ stands for the combination of their local states $s.x$, $x \in Y$. Similarly to programming notations, $s.Y$ is used both for the variables in objects $Y$ and for their values, depending on context.
Transitions in a joint action system are given in terms of actions; the set of all actions is denoted by $A$. Each action $a \in A$ has a fixed collection of participants $pa \subseteq X$, a guard $ga$, which is a boolean expression depending on $s.pa$, and a deterministic body $ba$, which expresses how variables in $s.pa$ are updated by the execution of $a$.\footnote{Nondeterministic actions are often needed in specification. The DisCo language provides for this by parameterized actions, where the parameter values are generated nondeterministically when an action is executed, and the bodies can still be kept deterministic.} Action $a$ can be executed in state $s$ only if $ga(s)$ is true, and its execution always leaves program variables outside $s.pa$ intact. Letting $ba$ also stand for the associated state mapping, the operation of a joint action system can be understood as a guarded iteration statement:

$$
\begin{align*}
\text{do} & \quad ga_1 \rightarrow s.p_a, := ba_1(s.p_a) \\
\text{do} & \quad ga_2 \rightarrow s.p_a, := ba_2(s.p_a) \\
\text{. . .} \\
\text{do} & \quad ga_n \rightarrow s.p_a, := ba_n(s.p_a) \\
\text{od.}
\end{align*}
$$

For simplicity we assume that $A$ always contains a special stuttering action $i \in A$ for which $pi$ is empty, $gi$ is identically true, and $bi$ does not affect the state.

An action is said to be enabled in state $s$ when $ga(s)$ is true. A set of actions is said to be enabled when at least one of them is enabled.

**Definition 1.** A joint action system is a quadruple $S = (X, \Sigma_0, A, F)$ where $X$ is a set of objects, $\Sigma_0$ is an initial condition for the state $s = \Sigma_0$, $A$ is a set of actions, and $F$ is a fairness family, which is a collection of fairness sets $f \subseteq A$.

**Definition 2.** A computation of a joint action system $S$ is a non-empty, finite or infinite, alternating sequence of states and actions (ending in a state in the finite case), $c = (s_0, a_1, s_1, a_2, s_2, \ldots)$, where $s_i$ are states and $a_i$ are actions such that

(i) the initial state $s_0$ satisfies $\Sigma_0$, and

(ii) for each occurrence $a_i$ of an action, $ga_i(s_{i-1})$ is true, and states $s_{i-1}$ and $s_i$ differ only by variables $s.pa$, being updated in $s_i$ by the values $ba_i(s_{i-1}.pa_i)$. A computation $c$ is fair (with respect to all sets $f \in F$), if it satisfies the following condition:

(iii) no fairness set $f \in F$ is enabled in the finite state of a finite $c$, and if $f \in F$ is enabled in infinitely many states of $c$, then $c$ contains infinitely many occurrences of actions in this $f$.

The fairness notion of (iii) is strong fairness; we could have another fairness family for weak fairness (also called justice) [33], but for simplicity we omit this. The set of all fair computations of $S$ will be denoted by $C(S)$, or simply by $C$ when $S$ is known from context. Using $pcl$ to denote prefix closure, $pcl(C)$ is the extension of $C$ with all finite prefixes of computations in $C$. 
Since no real time is involved in S, its computations will also be called \textit{logical computations}. When the identity of actions has no significance, computations can be given as sequences of states only, $c = (s_0, s_1, \ldots)$. Similarly, when the initial state is known from context, it is sufficient to indicate only actions, $c = (a_1, a_2, \ldots)$.

The model thus defined is an interleaving model with actions as atomic units of execution. Temporal logic of actions (TLA) is an appropriate logic for formulating and proving formal properties of such computations [26]. (For the limitations of TLA in expressing properties of the intuitive model, see [20].) System S is said to possess a property in this logic iff all fair computations $c \in C(S)$ have this property.

Each property of a computation is a combination of a \textit{safety property} and a (pure) \textit{liveness property} [2]. Informally, a safety property expresses that “nothing bad ever happens”, and its nonvalidity for a computation is always apparent in some finite prefix of it. Correspondingly, a liveness property expresses that “something good eventually happens”, and its nonvalidity for a computation cannot be checked from any finite prefix. In our formalism, actions give only safety properties. The only liveness properties that can be explicitly enforced are \textit{fairness properties}, expressed by the fairness sets. No liveness properties are implicitly assumed, not even fundamental liveness, i.e., fairness with respect to the set of all non-stuttering actions. Therefore, fairness is the only force that can force something to happen in a system.

The significance of indicating the participants $pa$ of an action is in guaranteeing syntactically that the local states of all other objects $x \notin pa$ remain unaffected by $a$. Two actions $a$ and $b$ are said to be \textit{independent} if $pa \cap pb = \emptyset$. If $a$ and $b$ occur as consecutive actions in a computation, and a distributed implementation has allocated variables in $s.pa$ and $s.pb$ to disjoint parts of the physical system, there may be no way for an external observer to determine the mutual order of $a$ and $b$. Other parts of the computation being equal, either both or none of the two orders would give a fair computation. It is therefore reasonable to consider computations equivalent if they differ from each other in this way only. Two computations that can be transformed into each other by a finite number of such exchanges are finitely equivalent. This can be generalized as follows to situations where an infinite number of exchanges is needed.

Let two computations be called $n$-identical, if they are identical to each other as far as their first $n$ actions (and the associated states) are concerned. Two computations are \textit{partial-order equivalent} if, for any $n > 0$, each of them is finitely equivalent to a computation that is $n$-identical with the other. In particular, addition or deletion of stuttering keeps a computation within the same equivalence class.

This equivalence partitions the set of all computations into equivalence classes that can be understood as \textit{partial-order computations} [34]. For any set $C'$ of computations, the (partial order) \textit{equivalence closure} of $C'$ will be denoted by $ecl(C')$. A property is called \textit{equivalence-robust} [3], if in each equivalence class it is possessed either by all or by none of its computations. Since there need not be any objective way for an observer to distinguish between different computations in an equivalence class, all reasonable requirements for a system are equivalence-robust. Therefore,
if a specification in temporal logic of actions, for instance, determines a collection $C$ of computations, then the larger set $ecl(C)$ can be equally well used instead.

2.2. Example

As an example consider a simple situation of communication over an unreliable channel (see Fig. 1). Let $x$ be a sender object, $y$ a receiver object, and $z$ and $u$ objects that model (interfaces to) two unreliable channels. The sender $x$ is trying to send a single message through channel $z$ to receiver $y$. If $y$ receives a copy of the message, it sends an acknowledgement to $x$ through the other channel $u$. As long as no acknowledgement has been received, $x$ keeps resending copies of the message. Channels $z$ and $u$ can carry at most one copy of a message at a time. Because of unreliability, these copies can be lost, but the transmitted copies cannot be corrupted.

The local states of the objects are assumed to consist of the following variables:

- $x.sending$ indicates that $x$ is sending a message, initialized as true;
- $x.msg$ is initialized with the message to be sent;
- $y.received$ indicates that a message has been received, initialized as false;
- $y.msg$ contains the message received, if any;
- $z.busy$ indicates that $z$ is busy with a copy of a message, initialized as false;
- $z.msg$ contains a copy of the message being transmitted, if any;
- $u.busy$ indicates that $u$ is busy in transmitting an acknowledgement, initialized as false.

Six actions are given as follows.

- Action $sm$ puts a copy of a message in channel $z$:
  \[ p_{sm} = \{x, z\}, \]
  \[ g_{sm} = x.sending \land \lnot z.busy, \]
  \[ b_{sm} = (z.msg := x.msg; z.busy := true). \]

- Action $rm$ takes a copy of a message from channel $z$:
  \[ p_{rm} = \{y, z\}, \]
  \[ g_{rm} = \lnot y.received \land z.busy, \]
  \[ b_{rm} = (y.msg := z.msg; y.received := true; z.busy := false). \]

\[ \text{Fig. 1. Objects and actions modeling an unreliable channel.} \]
Action $sa$ puts an acknowledgement in channel $u$:
\[
\begin{align*}
p_{sa} &= \{y, u\}, \\
g_{sa} &= y.\text{received} \land \neg u.\text{busy}, \\
b_{sa} &= (u.\text{busy} := \text{true}; y.\text{received} := \text{false}).
\end{align*}
\]
Action $ra$ takes an acknowledgement from channel $u$:
\[
\begin{align*}
p_{ra} &= \{x, u\}, \\
g_{ra} &= x.\text{sending} \land u.\text{busy}, \\
b_{ra} &= (x.\text{sending} := \text{false}; u.\text{busy} := \text{false}).
\end{align*}
\]
Action $dm$ drops a copy of a message from channel $z$:
\[
\begin{align*}
p_{dm} &= \{z\}, \\
g_{dm} &= z.\text{busy}, \\
b_{dm} &= (z.\text{busy} := \text{false}).
\end{align*}
\]
Action $da$ drops an acknowledgement from channel $u$:
\[
\begin{align*}
p_{da} &= \{u\}, \\
g_{da} &= u.\text{busy}, \\
b_{da} &= (u.\text{busy} := \text{false}).
\end{align*}
\]
Assuming fairness with respect to each individual action, $x$ keeps sending copies of the message until it receives an acknowledgement, and $y$ sends an acknowledgement for each copy received. Similarly, each copy of a message and an acknowledgement must eventually be either dropped or received. If no acknowledgement is ever received by $x$, then fairness with respect to $ra$ means that only a finite number of them are sent by $y$, which implies that only a finite number of copies of a message are ever received by $y$. Fairness with respect to $rm$ then implies that only a finite number of copies are sent by $x$, which leads to contradiction, and hence, an acknowledgement must eventually be received by $x$.

As an example of a computation consider the sequence of actions
\[
c = (sm, rm, sm, sa, rm, da, sa, ra).
\]
Here the first attempt to transmit a copy of the message succeeds, but, before getting an acknowledgement, $x$ resends another copy. The first acknowledgement is dropped in $da$, but the second attempt succeeds. If the order of the second $sm$ and the subsequent $sa$, for instance, is reversed, another computation is obtained that is partial-order equivalent with $c$. 

The only property of \( c \) that relates to real time is the partial ordering of its actions in \( ecl(c) \). For instance, we cannot say anything about the mutual real-time order of consecutive independent actions like the second \( sm \) and the subsequent \( sa \). Properties that we would need to deal with in real-time modeling include real-time concurrency of such actions, and real-time distances between actions. In particular, one should be able to verify maximal (and minimal) delays between actions whose real-time order is uniquely determined.

2.3. *Separation of system and environment parts*

As shown by the above example, joint actions provide a model-oriented approach to specification. Without a separation of the system and its environment it is not, however, clear what such a closed system specifies. In general there are several possibilities for such a partitioning. In the above example we can adopt the interpretation that the system part consists of the sending and receiving ends of the protocol, and that the physical communication channels are the environment. (For simplicity we have not modeled how messages are given to this system, or how they are delivered from the receiving end. Therefore, such higher levels of communication cannot be taken as the environment in this model.)

In general, we assume that the objects of a joint action system \( S = (X, \Sigma_0, A, F) \) are partitioned into *internal* (system) and *external* (environment) objects, \( X = X_S \cup X_E, X_S \cap X_E = \emptyset \), modeling the state of the system and the environment, respectively. Similarly, non-stuttering actions are partitioned into *system actions* and *environment actions*, \( A = A_S \cup A_E \cup \{i\}, A_S \cap A_E = \emptyset \). Intuitively, environment actions are those for which the environment is responsible, and on whose proper execution one can rely in an implementation of the system. System actions model what the system does: they cannot have external objects as participants, and they are the ones that are to be implemented. Communication between the system and its environment takes place through internal (interface) variables that are accessed by both system and environment actions.

In the above example our interpretation of the system and environment parts is expressed by the following partitioning:

\[
X_S = \{x, y, z, u\}, \quad X_E = \emptyset, \\
A_S = \{sm, sa\}, \\
A_E = \{rm, ra, dm, da\}.
\]

Environment behavior is in this case so simple that its modeling needs no external objects.

A correct implementation of this specification would be one where all internal objects are represented, and their local states are transformed as expressed by the system actions in the model, assuming that the environment behaves as described by the environment actions. What else an implementation does (with auxiliary variables, for instance) is irrelevant for correctness. The specification itself may also
contain auxiliary (internal) variables that need not be present in an implementation, provided that all changes in the relevant variables still obey the model. The problems of correct implementation of joint action specifications have been discussed in more detail in [22].

2.4. Superposition

Superposition is a design method for stepwise development of reactive and distributed systems [13]. The execution model of joint actions is especially suited for it, and allows a specification language to support it effectively [18, 19].

In this paper we understand superposition as a transformation by which a joint action system can be modified in the following ways. Firstly, the state $s$ can be extended by additional variables, either in old objects or in new ones. Secondly, actions can be refined (by one or more separate refinements for each old action) by adding new participants to them, by strengthening their guards, and by allowing them to update the newly added program variables. New actions, which are not allowed to have any effect on old program variables, can be introduced as refinements of the implicit stuttering action.

A crucial property of superposition is that each computation of the resulting system $S'$ can be projected to a computation of the original system $S$ by deleting all new program variables and removing the effects of actions on them. Such a projection of $C(S')$ will be denoted by $C(S') \downarrow S$. The construction guarantees that all safety properties of $S$ are valid also in $S'$. However, since nothing is assumed of the fairness family of $S'$, liveness properties need not be preserved, i.e., $C(S') \downarrow S$ need not be included in $C(S)$. If also liveness properties are preserved, $S'$ is a refinement of $S$.

There are two slightly different uses for superposition. Firstly, it allows incremental specification by stepwise introduction of program variables; this also leads to modularity in specifications. Secondly, it provides an effective method to refine joint action systems towards implementation. Typically, a joint action specification can be given at a level of abstraction where direct (distributed) implementation is prevented either by multi-party actions with complex guards, or by fairness requirements that cannot be enforced in a distributed fashion. As illustrated in [6–8], a system can be simplified in such respects by systematic use of superposition. It is often essential in such refinement to reduce the set $C(S') \downarrow S$ to be a proper subset of $C(S)$. The use of superposition as a basis for design methodology has been reviewed in more detail in [21].

2.5. Concurrency in joint action models

The operational model of joint actions is sequential in the sense that actions are always executed one at a time in an interleaved fashion. Actions are considered to be instantaneous, and time is involved only in the order of their execution. This conceptual simplicity of the execution model raises the question, whether reasoning
on such a model is at all valid for any reasonable concurrent implementation. In this section we present the most obvious approach to this problem, which is to refine the granularity of actions. The idea is to introduce concurrency simply by modeling the beginnings and ends of actions as separate (instantaneous) actions. Technically we can use superposition to derive a joint action model of such concurrent implementations. Although this leads to certain problems that prevent us from adopting this approach for real-time modeling, understanding these problems is important for the rest of the paper.

Discussion of concurrency makes sense only in connection with some execution agents. Let $E$ be a set of such agents, and let $\text{agent} : X \rightarrow E$ be an allocation mapping that associates a unique agent to each object. For a set $Y$ of objects the collection of agents $\text{agent}(x)$, $x \in Y$, will be denoted by $\text{agent}(Y)$. We assume that execution agents are not shared by the environment and the system, $\text{agent}(X_\circ) \cap \text{agent}(X_e) = \emptyset$.

The intuitive idea is that all agents in $\text{agent}(pa)$ are required for the execution of an action $a$, and that no agent can be executing more than one action at a time. The execution of an action starts by all agents in $\text{agent}(pa)$ executing a handshake whereby they jointly establish that $a$ is enabled and commit themselves in its execution. Duration of time is modeled by separating the beginning (handshake) and ending of $a$ into distinct events. Since actions may require some of the participants longer than others, separate end events are given for all agents involved.

This can be described more precisely with superposition as follows. For an arbitrary system $S = (X, \Sigma_0, A, F)$ we extend $X$ by a new object $x_e$ for each agent $e \in E$, with a boolean variable $x_e.b$ indicating whether $e$ is busy in some action or free. In the initial state all agents are assumed to be free. Each action $a$ is refined as follows:

- objects $x_e$, $e \in \text{agent}(pa)$, are added as new participants to $a$,
- the guard $ga$ is strengthened to require that $x_e.b = \text{false}$ for all $e \in \text{agent}(pa)$, and
- $ba$ is refined to make all agents $e \in \text{agent}(pa)$ busy, $x_e.b := \text{true}$.

In addition, a new action $\text{free}_e$ is added for each agent $e \in E$, such that $x_e$ is its only participant, it is enabled whenever $e$ is busy ($x_e.b = \text{true}$), and its body makes $e$ free ($x_e.b := \text{false}$). All singleton sets $\{\text{free}_e\}$ are assumed to be included in the fairness family.

The refined old actions now stand for the handshakes, which require synchronization by all agents involved. All state changes in the original $X$ are modeled to take place in these. The role of the new actions is only to indicate releasing of agents to allow their participation in further actions. Fairness with respect to these actions guarantees that each agent is always eventually released from any action in which it participates. Potential for concurrency is present in the model in the sense that different agents may be occupied in different actions at the same time.

To illustrate this, Fig. 2 shows a diagram of one possible concurrent computation that corresponds to the computation $c$ of Section 2.2, under the assumptions that all objects are allocated to different agents, and that the real-time order of events
coincides with their interleaving order. The vertical edges of the polygons represent events in the concurrent model. Only their mutual ordering is relevant; the order of handshakes (the left ends of the polygons) is the same as that of the actions in the original c. The lengths of the polygons are arbitrary, since no measure of time is involved. In terms of such diagrams, partial-order equivalence of computations means that their diagrams can be transformed into each other by shifting the associated polygons horizontally without crossing other polygons.

Transformation by superposition guarantees that all safety properties of the original system are satisfied in this concurrent model. Unfortunately, no fairness requirements in the new model can, in general, guarantee that the liveness properties of the original model would hold. This is due to strengthening of guards, which may have the effect that actions that are continually enabled in the original model may become continually disabled in the new model. Liveness properties therefore need separate checking, or they have to be guaranteed by other means. The two models and their relationship have been considered in more detail in [7].

Because of these problems with liveness properties, this concurrent execution model cannot be directly used in the following. Still it shows the basic idea on which a reasonable real-time execution model can be built. At this stage we can make the following conclusions:

- There is no logical conflict between interleaved and concurrent execution of actions. Reasoning about state properties in the former model can be taken as an approximation where no information is available about the scheduling and durations of actions.
- In concurrent execution, atomicity of actions means that only such observations of the global state are allowed where no action is in the middle of execution. On the other hand, any observation sequence is permitted that conforms to this atomicity and is consistent with the partial ordering imposed by common execution agents in actions [12].
- Fairness requirements in an interleaved model may be stronger than what can be expressed in the concurrent model, or what could be effectively enforced in a distributed fashion in concurrent implementations.

Although we discussed here only concurrency, not metric time, the approach conforms to how real time is traditionally added to temporal-logic-based models.
(see [15, 27], for instance). The main principles of such models can be characterized as follows:

(i) all actions are instantaneous,
(ii) the interleaving order of state-modifying actions coincides with their real-time execution order,
(iii) time is advanced only by special “tick” actions that do not modify the state,
(iv) real-time constraints are imposed by appropriate conditions in the guards of actions.

Because of the problems that were encountered in the simultaneous modeling of concurrency and fairness, we will modify this approach so that, instead of (i), durations are associated directly with the actions. This leads to abandoning the intuitively natural principle (ii) also.

3. Scheduled systems

3.1. Introduction

Joint action specifications provide executable models of a system and its environment. Such models are sufficient for dealing with logical properties that do not refer to metric time. For instance, temporal eventualities and deadlocks belong to this category, while time bounds or minimum delays between actions do not.

In order to deal with real-time properties we add metric time to the model in terms of timing and scheduling. By timing we understand an association of durations with action execution; scheduling associates a start time with each action occurrence in a computation. Stuttering actions are assumed not to take time, and will therefore be ignored here.

If actions \( a \) and \( b \) need a common execution agent, their mutual order is always uniquely determined in their execution. This means that for scheduling purposes the general independence relation \( pa \cap pb = \emptyset \) of actions changes into \( \text{agent}(pa) \cap \text{agent}(pb) = \emptyset \). This leads to a stronger partial-order equivalence of computations, which we call partial-order equivalence for the allocation mapping \( \text{agent} \). In particular, if all objects are mapped to the same execution agent, all partial-order equivalence classes reduce to singleton sets.

3.2. Timing

Although the concurrent model of Section 2.5 is not applicable here as such, Fig. 2 suggests the idea of associating a duration with each agent that participates in an action. These durations consist of execution times for the participating objects, and of communication overhead that depends on the allocation mapping \( \text{agent} \).
For each action $a$ and participant $x \in pa$ we assume a duration $D_{a,x} \geq 0$ that depends, in general, on the state $s_{pa}$ in which $a$ is executed. The communication overhead for an executing agent $e \in \text{agent}(pa)$ to participate in $a$ is given by another nonnegative real number $B_{a,e}$, which may also depend on state $s_{pa}$ and on the allocation mapping $\text{agent}$. The total time $T_{a,e}$ required of agent $e$ to participate in action $a$ is then

$$T_{a,e} = B_{a,e} + \sum_{\text{agent}(x)=e} D_{a,x}.$$  

With this definition, timing is independent of scheduling, i.e., durations do not depend on when actions are executed. They may, however, depend on the current state. In many situations it is possible to work with a simpler model where each $D_{a,x}$ is constant, and all $B_{a,e}$ are 0. This makes it easier to compare the effects of different allocation mappings $\text{agent}$.

Instead of fixed timing parameters, a practical real-time model can assume only that some predicate $P$ on the allocation mapping and timing parameters is satisfied.

3.3. General principles of scheduling

In Section 2.3 we discussed the “opening” of a closed joint action system $S = (X, \Sigma_0, A, F)$ into system and environment parts $X = X_S \cup X_E$, $A = A_S \cup A_E \cup \{i\}$. Implementability requires that the fairness responsibilities of the system and the environment can also be separated. Therefore we assume that the fairness family $F$ is of the form $F = F_S \cup F_E$, where sets in $F_S$ contain system actions only, and sets in $F_E$ contain environment actions only. In addition, we assume fundamental liveness of the system part, i.e., $A_S \subseteq F_S$, and that no environment action can disable any system action. These assumptions reflect the situation that the system—unlike the environment—will not stop as long as it can do something, and that time may be needed between recognizing that an action is enabled and its actual execution.

With these assumptions we can define what we mean by scheduling of actions. We use $C$ to denote the set of all fair computations, and $C^+$ for those computations that are fair with respect to fairness sets in $\{A_S\} \cup F_E$. Obviously $C \subseteq C^+$, since the fairness requirements of $F = F_S \cup F_E$ are relaxed in $C^+$.

In accordance with [1], the generation and scheduling of computations can be interpreted as a two-person infinite game played by the system and the environment. The initial state $s_0$ is chosen by the environment, after which the environment and the system alternate moves, the environment taking the first move. In each environment move a finite (possibly empty) sequence of environment actions are executed, and a positive real number $t_i$ (start time) is associated with each. In a system move, at most one system action is executed with a unique start time. In selecting their moves, both parties are restricted by the safety properties of $S$, which means that at each stage the game has produced a finite sequence $c \in \text{pcI}(C^+)$ with an association of start times with actions. For infinite games the players are restricted by the fairness sets in $\{A_S\} \cup F_E$ to produce computations in $\text{ecI}(C^+)$. (Stuttering actions can be
assumed to be generated only in the end.) The system wins, if the produced infinite computation belongs to $ecl(C)$, i.e., satisfies the original fairness requirements. The system loses (and the environment wins), if this is not the case.

This process leads to sequences of the form $(s_0, (a_1, t_1), s_1, (a_2, t_2), \ldots)$ called scheduled computations. More precisely:

**Definition 3.** Scheduling is a partial mapping $\sigma$ of $ecl(C^+)$ to scheduled computations, with a non-empty domain $\text{dom}(\sigma)$, such that

(i) whenever $\sigma(c)$ is defined, $c$ and $\sigma(c)$ are identical with respect to occurrences of states $s_i$ and actions $a_i$;

(ii) if action $a_i$ precedes $a_j$ in $c$, and $\text{agent}(pa_i) \cap \text{agent}(pa_j) \neq \emptyset$, then $t_j - t_i \geq 0$ in $\sigma(c)$,

(iii) for two computations with a common prefix, $cc', cc'' \in \text{dom}(\sigma)$, the start times $t_i$ associated with actions $a_i$ in the common prefix $c$ are identical in $\sigma(cc')$ and $\sigma(cc'')$,

(iv) if $cc' \in \text{dom}(\sigma)$, and the prefix $c$ ends in a state in which an environment action $a$ is enabled, then there is some $cc'' \in \text{dom}(\sigma)$ such that $c''$ starts with $a$, and

(v) if $cc' \in \text{dom}(\sigma)$, and the prefix $c$ ends in a state in which some system action is enabled, then there is some $cc'' \in \text{dom}(\sigma)$ such that $c''$ starts with a system action.

It should be noticed that we are continually dealing with closed systems, and that our notion of scheduling therefore covers both the scheduling mechanism to be implemented for system actions, and the scheduling of environment actions, which the scheduler cannot affect. The meaning of the above conditions for scheduling can be explained as follows. Condition (i) expresses the correspondence between $c$ and $\sigma(c)$. Condition (ii) restricts the start times of actions in $\sigma(c)$ to conform to the partial order that is determined by $c$ and the allocation mapping. Condition (iii) prevents scheduling from depending on the future. Conditions (iv) and (v) reflect the fact that a scheduler cannot prevent the environment either from executing any environment action that is enabled, or from postponing such an execution arbitrarily.

A scheduled system $(S, \sigma)$ is now defined as a joint action system $S$ together with scheduling $\sigma$. For each logical computation $c \in \text{dom}(\sigma)$, $\sigma(c)$ is a scheduled computation of $(S, \sigma)$. A scheduled system $(S, \sigma)$ is a model of a possible implementation of $S$. For system actions scheduling serves for two purposes: in addition to introducing real time properties it should provide an implementation of the fairness properties of $S$. However, although all safety properties of $S$ are necessarily present also in $(S, \sigma)$, its liveness properties may be violated, since $\text{dom}(\sigma)$ is not, in general, restricted to fair computations. Properties that refer to times $t_i$ in scheduled computations are called real-time properties.
We define:

Definition 4. 
* Scheduling $\sigma$ is *sound* if, for all pairs $c, c' \in \text{ecl}(C^+)$ that are partial-order equivalent for the allocation mapping *agent* and for which both $\sigma(c)$ and $\sigma(c')$ are defined, action occurrences that correspond to each other in $\sigma(c)$ and $\sigma(c')$ have identical start times.

* Scheduling $\sigma$ is *complete*, if $C \subseteq \text{dom}(\sigma)$.

* Scheduling $\sigma$ is *totally correct*, if $\text{dom}(\sigma) \subseteq \text{ecl}(C)$, i.e., $\sigma$ allows only games where the system wins.

* Scheduling $\sigma$ is *partially correct*, if it never leads (by a finite number of steps) to a situation where the environment has a winning strategy.

* When not partially correct, scheduling $\sigma$ is *incorrect*.

Soundness expresses the intuitively natural view that computations that are partial-order equivalent for the allocation mapping are alternative observations of the same "real" computation, and, hence, there is no objective basis to schedule them differently. Centralized scheduling can, however, easily violate this property. Completeness corresponds to the situation where no additional logical properties are introduced by scheduling, and maximal freedom is therefore left for the actual scheduling that is used in the implementation. Below in Section 4.1 we shall show how sound and complete scheduling can be superposed on any joint action system $S$.

The different notions of correctness characterize the degree to which scheduling is correct also with respect to the required liveness properties. Partial correctness guarantees that, no matter how the environment behaves, it is always possible for the system to win. Total correctness requires this to happen. Although a partially correct scheduling need not exclude unfair computations, its nondeterminism allows to extend every initial prefix of a computation into a fair one. Incorrectness means an irrecoverable possibility to end up in an unfair computation. (Here we have deviated slightly from standard terminology. We have taken partial correctness to mean that a program can be refined into a totally correct one by an arbitrary refinement, which may reduce both nondeterminism and the domain of nontermination [5]. Ordinarily only the latter kind of refinement would be allowed; here only the former kind is possible.)

The above definitions lead to the following fundamental properties:

**Proposition 1.** Any complete scheduling is partially correct.

This holds, since enforcing all fairness requirements on a complete scheduling makes the system always win.
Proposition 2. If \( \text{ecl}(C) = \text{ecl}(C^+) \), then any scheduling is totally correct.

This follows directly from the definitions.

Proposition 3. If \( P \) is an equivalence-robust property of \( S \), and \( \sigma \) is totally correct, then \( P \) holds also in \( (S, \sigma) \).

This is also obvious by the definition of total correctness.

3.4. Forcing total correctness

When \( \text{ecl}(C(S)) \) is properly included in \( \text{ecl}(C^+(S)) \), an arbitrary scheduling need not be totally correct. This leaves two possibilities for deriving an implementation as a scheduled system. Either the scheduling policy is restricted so that its total correctness is guaranteed for \( S \), or \( S \) is transformed into another system \( S' \) for which \( \text{ecl}(C(S')) = \text{ecl}(C^+(S')) \), or for which a totally correct scheduling is easier to implement.

The first of these alternatives has the problem that no distributed scheduling policy can be expected to support the fairness requirements of an arbitrary joint action system \( S \). In the second alternative it is, in general, equally unrealistic to aim at a refinement \( S' \) that would exhibit all fair behaviors, i.e.,

\[
\text{ecl}(C(S')) \downarrow S = \text{ecl}(C^+(S')) \downarrow S = \text{ecl}(C(S)).
\]

(For theoretical constructions of this kind, see [4, 9].) For such reasons fairness has sometimes been considered to be an unworkable notion. In executable specifications it is, however, a fundamental concept, although engineering insight may be needed in transforming a given system into a form where its fairness assumptions allow direct implementation.

Fairness conditions that can be associated with the concurrent model of Section 2.5 are somewhat weaker and more realistic. Therefore, an interesting but still very general class of schedulings is obtained by assuming that \( \sigma \) is able to support these. In [7, 8] it was shown that these can be realistically supported in a distributed fashion, provided that a minimal centralized facility, such as a broadcast channel, is available. Sufficient conditions were also derived under which scheduling with such liveness properties is totally correct. For the case that these conditions do not hold for \( S \), it was shown how \( S \) can be refined by superposition into \( S' \) that does satisfy them. Such refinements actually introduce explicit restrictions on scheduling into the guards of the actions. This is done by adding auxiliary control variables by which the enabling of some actions is controlled; these control variables are updated both in the refined old actions and in additional control actions.

The same approach can be applied with the assumption that (in addition to environment fairness, which is not on the responsibility of the implementation) only fundamental system liveness can be supported, which leads to sufficient conditions for proving that \( C = C^+ \), and to a method for refining \( S \) into a form for which this
holds. An example that illustrates such transformations will be discussed in Section 4.6.

4. Dealing with real-time properties

4.1. General undelayed scheduling

At least for system actions, any reasonable scheduling policy can be assumed to be undelayed in the following sense:

**Definition 5.** Scheduling is **undelayed**, if the start time of an action is always determined to be as early as is possible for the execution agents that it requires.

Notice that undelayed scheduling does not prevent a scheduler from decisions where an enabled action is postponed to be executed after some other action with an overlapping set of participants, even when that action is not yet enabled. Such decisions may, in fact, be crucial for the ability to guarantee some fairness properties, as will be demonstrated by the example in Section 4.6.

The definition of undelayed scheduling implies that the time when an action is started depends only on its predecessors in the partial order determined by the interleaved order and the allocation mapping. Since action durations were assumed to depend only on the local states of the participants, we then have:

**Proposition 4.** Undelayed scheduling is always sound.

The most general form of undelayed scheduling is also complete. We denote it by $\sigma_0$, and call it **general undelayed scheduling**. In the following we show how it can be imposed on any joint action system $S$ by superposition. Timing parameters and the allocation mapping $agent$ are implicitly involved in $\sigma_0$.

As in Section 2.5, the set of objects $X$ is extended by a new object $x_e$ for each execution agent $e \in E$, and all agent objects $x_e$, $e \in agent(pa)$, are added as new participants to each action $a$. As for the state, each $x_e$ is provided with a local variable $x_e.t$ to indicate the local time of agent $e$, initialized as 0. Intuitively, $x_e.t$ always indicates the time when $e$ was last involved in an action.

No new actions are added, and the guards of the old actions also remain unchanged. The body of each action $a$ is, however, refined to update the local times of the participating agents as follows. If action $a$ is to be executed in state $s$, the maximum value of $x_e.t$, $e \in agent(pa)$, indicates the earliest time when $a$ can be started. Let this value be denoted by $t_a$, $t_a = \max_{e:agent(pa)} x_e.t$. To describe that this earliest possible start time is chosen, the body $ba$ is refined to update $x_e.t$ for each $e \in agent(pa)$ by $x_e.t := t_a + T_{a,e}$. Finally, the family $\{A_3\} \cup F_E$ is taken as the fairness family of the resulting system.
Comparing to $S$, this representation of $(S, \sigma_0)$ has time in additional program variables; the local time $x, t$ always indicates when agent $e$ was last occupied in an action. Obviously $\sigma_0$ is both complete and sound. The interleaving order of actions in a logical computation $c$ may therefore deviate from the real-time order indicated by the start times in $\sigma_0(c)$.

Scheduled computations can be visualized by diagrams like the one given in Fig. 3, which illustrates a scheduled version of the computation discussed in Section 2.2. Timing gives the lengths of the polygons, and undelayed scheduling means that all polygons are pushed as far to the left as possible.

4.2. Monotonic properties

For two schedulings $\sigma$ and $\sigma'$ with $\text{dom}(\sigma) \supseteq \text{dom}(\sigma')$ we write $\sigma \succeq \sigma'$, if for any $c \in \text{dom}(\sigma')$, $t_c \geq t'_c$ holds for all corresponding start times in $\sigma(c)$ and $\sigma'(c)$ This is the case, for instance, for two undelayed schedulings with the same allocation mapping agent, if all timing parameters $T_{a,e}$ in $\sigma'$ are at most the corresponding parameters $T_{a,e}$ in $\sigma$, i.e., $T_{a,e} \geq T'_{a,e}$ for all $a \in A, e \in \text{agent}(pa)$.

We define:

**Definition 6.** A real-time property is monotonic, if its validity under scheduling $\sigma$ implies its validity under any scheduling $\sigma'$ with $\sigma \succeq \sigma'$.

Intuitively, monotonic properties are those that cannot be violated by more efficient execution of actions, provided that the order of executing the actions remains the same. Of course, all logical properties are monotonic, since by definition they are insensitive to scheduling. Although all interesting real-time properties are not monotonic, a significant part of them seems to fall in this category. We have:

**Proposition 5.** With $\sigma_0$, all monotonic properties are preserved under any shortening of execution times.

Monotonic properties that are valid for $\sigma_0$ are therefore insensitive to such changes in the underlying execution environment that intuitively only increase its efficiency.
4.3. Handling of delays

It is often the case that delays are needed between some actions. For example, Fig. 3 shows the (probably) undesirable situation where another copy of the message is sent before there even was a chance to receive an acknowledgement for the first one. Such delays are especially needed in the modeling of the environment.

In principle there are two ways to handle this problem. One is to introduce dummy delay actions that only describe passing of time for their participants. Delay statements in real-time programs can be interpreted as such actions for the associated execution agents.

For joint action specifications it is, however, more natural to introduce auxiliary delay objects to participate in the actions between which delays are needed. In Fig. 4 we show how the diagram of Fig. 3 changes when a delay object $d$ is introduced to cause a delay between consecutive $sm$ actions.

![Fig. 4. Diagram of a scheduled computation with delays.](image)

4.4. Real-time constraints

Complete scheduling introduces no further logical properties. However, for any timing some of the scheduled computations may violate the intended real-time properties. For instance, the only way to guarantee responses from a system is by fairness requirements, but with fairness we can only enforce that a response is eventually given, not when this will happen. Therefore, the conclusion is sometimes made that fairness is useless in the modeling of real-time properties. However, although fairness requirements cannot by themselves enforce timely execution of actions, they are sufficient to achieve this in systems that have been designed appropriately. For instance, if an action is enabled, and all other actions that involve the same agents are continually disabled, fairness with respect to this action will force it to immediate execution—immediate in real time, not necessarily in the interleaving order.

Achieving the desired real-time properties may therefore require that some of the logically possible computations be excluded, which can be done by the technique that was discussed in Section 3.4. This means that superposition can be effectively...
used to transform a system into a form where actions are executed within some
time bounds. An advantage of imposing real-time constraints in this way is that no
special assumptions are needed about scheduling.

As an example, consider an extension of the example of Section 2.2 by an
additional dummy action \( aa \) with

\[
\begin{align*}
p_{aa} &= \{x\}, \\
g_{aa} &= \text{true}, \\
b_{aa} &= (\text{skip}),
\end{align*}
\]
and with delays in \( sm \) as shown in Fig. 4. Fairness with respect to \( sm \) guarantees
that it will eventually be re-executed after the delay, unless \( ra \) has taken place before
that. However, execution of \( aa \) can intervene, and there is no bound for the number
of times this may happen (see Fig. 5). Therefore, no real-time bounds are valid for
the re-execution of \( sm \). Similarly, repeated involvement in \( aa \) may prevent \( x \) from
participating in \( ra \). The desired real-time constraints can, however, be imposed by
refining the system so that \( aa \) does not stay continually enabled, unless an acknowl-
edgement has already been received.

![Fig. 5. Unbounded delay for re-execution of sm.](image)

4.5. Incomplete scheduling

The principle of letting the interleaving order of actions coincide with their
real-time order would have been intuitively natural, and this approach is usually
followed when time is added to systems of instantaneous actions [15, 27]. With joint
action systems and durational actions this would also be possible by explicit
reduction of nondeterminism, but would lead to considerable complexities in the
guards. For distributed implementation this would also cause additional problems.

To discuss such schedulings we define:

**Definition 7.** Scheduling \( \sigma \) is **order-preserving** if, for each scheduled computation
\( \sigma(c) = (s_0, (a_1, t_1), s_1, (a_2, t_2), s_2, \ldots), t_i \leq t_{i+1} \) holds for all \( i \).

In real-time execution models of programming languages it is natural to assume
an order-preserving scheduling that follows the **maximal parallelism principle** [17,
Real-time specification and modeling

30], whereby execution agents of the system part are never kept idle when something can be done with them:

**Definition 8.** An undelayed, order-preserving scheduling \( \sigma \) is a maximal parallelism scheduling if, for each scheduled computation \( \sigma(c) = (s_0, (a_1, t_1), s_1, (a_2, t_2), s_2, \ldots) \), if \( a \) is a system action that is enabled in state \( s_i \) with start time \( t_i \), then \( t_{i+1} \leq t \).

This seems to reflect truthfully how programs are executed in practice. The underlying philosophy deviates, however, significantly from temporal logic approaches, as far as the fundamental question “What makes a system tick?” is concerned. In joint action models, for instance, fairness is the only “law of nature” that can force something to happen in a closed system, while maximal parallelism postulates a kind of “abhorrence of a vacuum” for the system part. It seems that the two principles are conflicting in the sense that they cannot both be conveniently applied at the same time.

The different philosophies also lead to different attitudes toward nondeterminism. In temporal logic specifications external and internal nondeterminism (i.e., nondeterminism in the environment and in the system, respectively) are treated symmetrically, and it is natural not to restrict nondeterminism unnecessarily. Maximal parallelism, on the other hand, is intended more for the modeling of implementations, where only external nondeterminism is significant, and internal nondeterminism is considered even harmful. As for the consequences for the modeling of real-time scheduling, fairness allows (and eventually also forces to) even drastic measures, if the required deterministic resources are not otherwise obtained for an enabled action, while the more deterministic nature of maximal parallelism may preclude those.

As an example consider again the example of Section 2.2 as extended in Section 4.4. With the timings of Fig. 4, computation \( c \) would no longer be possible under maximal parallelism. Action \( aa \) would be forced to start after \( sm \), since its only participant then has no other alternatives. Under \( \sigma_0 \) the execution of \( aa \) would be possible but not obligatory.

Except for environment actions, nondeterminism is involved in maximal parallelism only when several alternative actions with overlapping participant sets could start at the same time. On the programming language level it is customary to assign priorities to processes (i.e., to actions executed by them), and to always force an enabled action with the highest priority to be executed next. Therefore we define:

**Definition 9.** Maximal parallelism scheduling \( \sigma \) is a priority scheduling if, for each scheduled computation \( \sigma(c) = (s_0, (a_1, t_1), s_1, (a_2, t_2), s_2, \ldots) \), if \( a_{i+1} \) is a system action, and if another system action \( a \) is also enabled in state \( s_i \) with the same start time \( t_{i+1} \), then the priority of \( a_{i+1} \) is at least that of \( a \).

Maximal parallelism is an example of incomplete undelayed scheduling. Because of incompleteness we have:
**Proposition 6.** There are systems for which maximal parallelism is incorrect, no matter how priorities are assigned to actions.

This can be shown as follows. Incorrectness, i.e., conflict with partial correctness, implies the existence of situations where the environment has a winning strategy. Due to "conspiracy" by other actions, a scheduled computation may treat an action unfairly even when the participants are never simultaneously available for this. Examples can easily be constructed where continual execution of such conspiring actions is forced by maximal parallelism. A concrete example will be discussed in Section 4.6 below.

Another consequence of incompleteness is that shortening of execution times may have drastic effects on properties that have been proved under maximal parallelism. Also monotonic and non-real time properties can be violated, since the logical computations to be selected by the scheduling may totally change.

Using general undelayed scheduling in design does not prevent from using an incomplete scheduling policy in the implementation. Any undelayed scheduling \( \sigma \) is obtained from \( \sigma_0 \) (with the same allocation mapping and timing parameters) by reducing its domain, i.e., \( \sigma(c) = \sigma_0(c) \) for all \( c \in \text{dom}(\sigma) \). This gives us:

**Proposition 7.** If \( \sigma_0 \) is totally correct for \( S \), then any undelayed scheduling \( \sigma \) is also totally correct for \( S \).

This follows directly from the definition of total correctness.

Let \( \text{Dom}(\sigma) \) and \( \text{Fair} \) denote the properties that a computation belongs to \( \text{dom}(\sigma) \) or satisfies the fairness conditions of \( S \), respectively. We then also have:

**Proposition 8.** For any logical property \( P \), if \( P \) holds in \( S \) and \( \text{Dom}(\sigma) \Rightarrow \text{Fair} \), then \( P \) holds also in \( (S, \sigma) \).

This is straightforward, as \( \text{Dom}(\sigma) \Rightarrow \text{Fair} \) is equivalent to \( \text{dom}(\sigma) \subseteq C \).

**Proposition 9.** For any real-time property \( P \), if an assumption \( Q \) on the allocation mapping and timing parameters implies that \( P \) holds in \( (S, \sigma_0) \), then for any undelayed scheduling \( \sigma \), \( Q \) implies that \( P \) holds also in \( (S, \sigma) \).

This is a consequence of the fact that, whenever \( \sigma(c) \) is defined, \( \sigma(c) = \sigma_0(c) \).

**4.6. Example**

As an example of dealing with real-time properties, consider the following formulation of the dining philosophers problem. Let there be \( n \) objects, \( n > 1 \), called philosophers, \( P_0, \ldots, P_{n-1} \), and \( n \) objects called forks, \( F_0, \ldots, F_{n-1} \), each object
being allocated on an execution agent of its own. Fork \( F_i \) is said to be on the left of philosopher \( P_i \), and \( F_{i+1} \) is on his right, counting modulo \( n \). The local state of each philosopher contains a boolean variable \( P_i.h \) that indicates whether he is hungry or not. For forks no local variables are introduced at this stage. The system is considered as a closed system without any particular partitioning into system and environment parts.

A high-level model of the system needs two kinds of actions for each philosopher, one for thinking, and one for eating:

\[
\begin{align*}
p \text{ think}_i &= \{P_i\}, \\
g \text{ think}_i &= \neg P_i.h, \\
b \text{ think}_i &= (P_i.h := \text{true}), \\
p \text{ eat}_i &= \{P_i, F_i, F_{i+1}\}, \\
g \text{ eat}_i &= P_i.h, \\
b \text{ eat}_i &= (P_i.h := \text{false}).
\end{align*}
\]

Obviously, no deadlocks can arise since both of the required forks are always taken in one action. To guarantee the expected liveness properties we take the fairness family to consist of all singleton sets of actions. With this assumption each non-hungry philosopher will eventually think and get hungry, and each hungry philosopher will also eventually eat.

Let some durations now be associated with the actions. For simplicity we assume that an eating action always takes the same duration for all its three participants. Figure 6 then illustrates general undelayed scheduling of a (logical) computation where a number of \( \text{eat}_0 \) actions precede \( \text{eat}_2 \) and \( \text{eat}_1 \), with the assumption that all philosophers are initially hungry. This demonstrates how a non-order-preserving

![Fig. 6. Undelayed scheduling of a dining philosophers computation.](image-url)
scheduling may allow arbitrarily long real-time intervals in which all the participants for an action (eat) are repeatedly idle and ready for the action, but this is still not started. The scheduling of eat, on the other hand, does not depend on the preceding eat, actions, provided that it precedes eat,1. For thinking actions it is always true that they are scheduled to take place immediately after the previous eating action by the same philosopher.

Obviously, similar idle periods would not arise with maximal parallelism. However, maximal parallelism could easily force a philosopher to starve by leading to continual “conspiracy” by the two neighbors. This demonstrates a conflict with the given fairness requirements. Not only does the domain of maximal parallelism scheduling include unfair computations, but these cannot even be removed by any restriction of nondeterminism, which means that the scheduling policy is incorrect for this model. Furthermore, this incorrectness cannot be straightened out by any assignment of priorities (cf. Proposition 6).

There are, however, two essential problems with the model. Firstly, its fairness assumptions are unrealistic to be directly enforced in a distributed implementation. Secondly, it gives no real-time bound for how long a hungry philosopher may need to wait for eating. In order to achieve such a bound in scheduled computations, consider the following simple policy to restrict nondeterminism. Let \([n/2]\) of the forks be marked, and let us allow eating only with a marked left fork and an unmarked right fork, after which the two forks should be exchanged. Obviously, this policy implements the quite restrictive rule that no philosopher can eat again, until both of his neighbors have also eaten. For simplicity we assume that initially exactly those philosophers are hungry for whom the forks are ready for eating.

Introducing boolean variables \(F_i,m\) to indicate the marking of forks, this policy can be imposed on the original model by superposition, resulting in the following modified eating actions:

\[
\begin{align*}
p\text{eat}_i &= \{P_i, F_i, F_{i+1}\}, \\
g\text{eat}_i &= P_i.h \land F_i.m \land \neg F_{i+1}.m, \\
b\text{eat}_i &= (P_i.h := \text{false}; F_i.m := \text{false}; F_{i+1}.m := \text{true}).
\end{align*}
\]

Obviously, freedom from starvation is preserved by this transformation, if we still assume fairness with respect to all individual actions. Furthermore, this fairness assumption is now also implementable, as the eating actions of two neighboring philosophers are never simultaneously enabled. An example of a scheduled computation in such a system is illustrated in Fig. 7, with marked and unmarked forks indicated by + and −.

Consider now the corresponding scheduled system with general undelayed scheduling. Let \(P_i.t\) and \(F_i.t\) denote the auxiliary variables for the local times of the philosophers and the forks, respectively, and let \(\text{Think}_{\text{min}}, \text{Think}_{\text{max}}, \text{Eat}_{\text{min}},\) and \(\text{Eat}_{\text{max}}\) be the minimum and maximum durations of thinking and eating actions, respectively. What we are now interested in is the smallest bound \(b\) for which the
Fig. 7. Four dining philosophers with marked forks.

invariant

$$P_i \cdot h \Rightarrow \max(F_i \cdot t, F_{i+1} \cdot t) - P_i \cdot t \leq b$$

would always hold. This bound would obviously be the maximum waiting time for a hungry philosopher.

For fixed values of $n$, $\text{Think}_{\text{max}}$, and $\text{Eat}_{\text{max}}$, such a bound $b$ obviously exists. Under the simplifying assumption that eating always takes longer than thinking, this bound can easily be shown to be $[n/2] \cdot \text{Eat}_{\text{max}} - \text{Think}_{\text{min}}$ [24]. For the purposes of this paper it is interesting to notice that this bound depends on $n$ but is independent of scheduling, provided that it is undelayed. On the other hand, it seems that with an approach that is based on some incomplete scheduling policy, the bound would not need to depend on $n$. However, distributed implementation of such a scheduling policy might then be a major problem.

It is also interesting to notice that the real-time property that was needed above was a safety property. This is true also more generally for our approach: practical real-time properties are combinations of safety properties and logical liveness properties. For proof techniques this is an advantage, since the proof rule for safety properties is simple: check that the property is satisfied initially, and that it is preserved by each action. (Auxiliary invariants may, however, be needed, and these need not be trivial to find. Such auxiliary invariants are also needed in the above example.) An example of real-time liveness properties is non-Zenoness, i.e., that an infinite number of actions is not executed in finite time [27].

5. Concluding remarks

An approach has been presented where logical properties are addressed by developing a joint action system on which real-time properties can be imposed by
a scheduling \(\sigma\) at any stage of development. We end up with a few general remarks of this approach.

Sound scheduling removes possible conflicts between the underlying interleaving model and a partial-order view of computations. The fact that fairness conditions are not, in general, equivalence-robust, causes no conflicts either, since all reasonable requirements can be assumed to be equivalence-robust.

The notion of scheduled systems allows modeling of real-time properties already at a high level of abstraction, where multi-party actions with complex guards may prevent a straightforward distributed implementation. With such modeling one can check, whether the required real-time properties seem realistic for implementation, but one cannot impose arbitrary real-time requirements on a system. When the joint action system is transformed into a form that is closer to implementation, new timing estimates are needed to recheck the real-time properties. This means that real-time properties cannot be refined in superposition similarly to logical safety properties.

Enforcing real-time properties with general undelayed scheduling makes any totally correct scheduling sufficient. There is, however, a trade-off between transformations that are then needed and the use of general-purpose scheduling policies that are incomplete in general, like maximal parallelism.

Finally, the technique of auxiliary time variables that we used for describing durational actions leads to the following classification of joint action models:

- **Type 0**: Basic models with no time variables.
- **Type 1**: Models where time variables have no effect on state properties.
- **Type 2**: Models where time variables may be used to strengthen the guards of actions, but not to affect how state variables are updated.
- **Type 3**: Models where time variables can be used freely in both guards and bodies of actions.

Although our focus was here on type 0 and type 1 models, type 2 and type 3 models can also be derived with superposition from their type 0 approximations.

As discussed in this paper, complete scheduling leads in a natural way from type 0 to type 1 models. In spite of the simplicity of type 1 models, and their potential usefulness in safety-critical applications, this class seems not to have drawn much attention previously. The special case of cyclic systems seems, however, to be the predominant approach for hard real-time systems [29].

In dealing with durational actions, the class of type 2 systems is often tacitly assumed. Maximal parallelism, for instance, leads from type 0 to type 2 models. Guard formulas become, however, quite complicated to be explicitly given in this case, their distributed implementation becomes more difficult, and guards also become interdependent in the sense that changes in one guard may affect the guards of other actions.

Finally, type 3 models are the most general class where explicit interaction is allowed between state variables and time variables. In general, models should be kept as simple as possible. Therefore, it is suggested that the models to be used in
practical specification and design of real-time systems be kept as low in this hierarchy as possible. For this reason the new class of type 1 models seems to deserve further research and evaluation.

Acknowledgement

The joint action approach was developed together with Ralph Back. Research on its use as a basis for a practical specification language was carried out in project DisCo, which was part of the FINSOFT programme of the Technology Development Centre of Finland, and was supported by four industrial partners. We are also indebted for discussions in the Real-Time track at the Sixth International Workshop on Software Specification and Design, and we also wish to thank the anonymous referees of this paper for their careful comments.

References


