

The Algebra of Multi-Tasking

C. J. Fidge

Software Verification Research Centre, The University of Queensland,
Queensland 4072, Australia
cjf@it.uq.edu.au

Abstract. Modelling multi-tasking behaviour is an important phase of real-time system design. It is shown how task scheduling principles can be captured in a CCS-based process algebra via extensions for both asymmetric interleaving, to model intraprocessor scheduling decisions, and for asynchronous communication, to model interprocessor precedence constraints. Examples are given of task preemption, blocking on shared resources, and multi-task transactions.

1 Introduction

Real-time scheduling theory has advanced considerably in recent years, thanks to a simplified, abstract computational model of multi-tasking behaviour [2, 6]. Here we exploit this model to show how task scheduling can be captured in a process algebraic formalism.

We begin with an existing timed, causal version of the Calculus of Communicating Systems (CCS). This formalism provides both a notion of real time, by allowing each atomic action to be stamped with its time of occurrence, and true concurrency semantics, by allowing each parallel agent to evolve in its own independent local context [1, 10]. We then extend this calculus further by introducing prioritised choices between eligible actions, via an asymmetric interleaving operator, and by introducing precedence constraints between parallel agents, via an asynchronous message-passing mechanism. It is then shown how the result can be used to model both multi-tasking on a single processor, and multi-task transactions occurring across several processors.

2 Previous Work

Multi-tasking systems have been modelled in a variety of formalisms. For instance, Yuhua and Chaochen used the Duration Calculus to prove properties of a dynamic-priority scheduling policy for non-communicating tasks [19]. Corbett showed how to translate tasks written in the Ada programming language into an analysable timed automata model [7]. Jacky used operations in the Z specification language to model the actions of a multi-tasking run-time environment [13]. Dong et al. presented an abstract specification of multi-processor multi-tasking in the Object-Z notation [8]. Liu et al. used the Temporal Logic of Actions to

show how applying a scheduling policy to a set of tasks can be represented as a formal program transformation [14]. Bornot et al. used the Timed Automata with Deadlines formalism, plus a notion of ‘urgency’, to model non-preemptive task scheduling [4].

More relevant to this paper are algebraic approaches. It has long been recognised that standard algebraic formalisms are ill-equipped to model multi-tasking systems, so many attempts have been made to extend them. Jackson showed how a set of tasks and a non-preemptive scheduler can be represented in the CSP process algebra, using a special ‘tick’ event to simulate the passage of time. He then showed that the resulting model can be analysed using the FDR model checker (although he was forced to modify the checker itself to recognise event priorities) [12]. Van Glabbeek and Rittgen noted the inadequacy of (some) process algebras for modelling timing and delays, and went on to devise an entirely new algebra for representing scheduling of atomic actions of varying durations [17]. Hsiung et al. used timed automata to represent tasks as ‘clients’ and a timer-driven scheduler as their ‘server’ and applied model checking techniques to the result [11]. Breviglieri et al. noted the central role played by task queues in making scheduling decisions and used a syntactic approach to model operating-system scheduling policies by extending a formal grammar with queues. Each preemptible task was represented as a sequence of distinct atomic actions, each taking one quantum [5]. Finally, and closest of all to our approach, Ben-Abdallah et al. used the CCS-based timed process algebra ACSR-VP to specify and reason about a multi-tasking system. In their formalism each action incorporates a priority and resource usage requirements. To define the asymmetric choices characteristic of priority-driven scheduling, they introduced a special ‘preemption relation’ between actions and then defined a prioritised transition relation [3].

All of the above-cited work is relevant to the current paper. However, the limitations and complexity of these models inspired us to instead seek a straightforward extension of the already well-known CCS notation. In particular, our approach differs from all of the above work by providing distinct operators for both intra and inter-processor task composition, and by introducing asynchronous communication to model precedence constraints between tasks.

3 Definitions

In this section we define the notations, some laws, and the operational semantics of our multi-tasking algebra. It is based on Milner’s CCS language [15], extended with timestamped actions, local agent contexts, a new scheduling operator, and asynchronous communication.

3.1 Notations

As usual, we assume a set of names representing the atomic actions that CCS agents can perform [15, §2.4]. However, we also distinguish a set of names for use as *asynchronous communication* actions and, since asynchronous *sends* and *receives* behave differently, we sometimes further distinguish these actions. Let

- a, b, \dots be standard, synchronous CCS actions (with conames \bar{a}, \bar{b}, \dots denoting synchronous outputs) [15, p. 43],
- i, j, \dots be asynchronous communication actions (with conames \bar{i}, \bar{j}, \dots denoting asynchronous sends),
- x, y, \dots be either synchronous or asynchronous actions,
- m, n, \dots be synchronous actions or asynchronous send actions,
- r, s, \dots be synchronous actions or asynchronous receive actions,
- f, g, \dots be renaming functions on (synchronous or asynchronous) action names [15, p. 43], and
- L, M, \dots be sets of (synchronous or asynchronous) action names [15, p. 43].

Since we are constructing a real-time model, we require each occurrence of an action x to be *timestamped* with a particular absolute time t , denoted ‘ $x@t$ ’ [1, 10]. The time domain is assumed to start at 0 and may be either discrete (the natural numbers) or dense (the non-negative reals). Let

- t, u, \dots be (absolute) times,
- $x@t, y@u, \dots$ be timestamped events (action occurrences), and
- I, J, \dots be sets of timestamped, asynchronous events.

To support true concurrency, parallel agents must carry their own local *contexts*. For an agent expression E , this usually consists of the time t at which E ’s causal-predecessor finished, since this defines the earliest time at which E can begin [1, 10]. Here we extend this mechanism to implement asynchronous communication. Let the context also contain a set I of timestamped events representing messages currently in transit to E . We denote agent expression E , starting no earlier than time t , and with a set I of incoming messages waiting to be received, as ‘ ${}_{(t,I)}E$ ’. Let

- P, Q, \dots be agents [15, p. 44] in our extended CCS,
- A, B, \dots be named agent constants [15, p. 44], and
- ${}_{(t,I)}E, {}_{(u,J)}F, \dots$ be agent expressions [15, p. 43] with explicit contexts.

As usual, agent expressions can be constructed from the standard CCS operators for prefixing ‘ \cdot ’, summation ‘ $+$ ’, restriction ‘ \backslash ’, and relabelling ‘ $[\cdot]$ ’ [15, §2.4]. In our true concurrency model the CCS “composition” operator ‘ $|$ ’ actually represents interprocessor *parallelism*, and we refer to it as such here. Our multi-tasking model also introduces an entirely new compositional operator for intraprocessor *scheduling*, denoted ‘ \uparrow ’.

To allow some activity a that consumes q quanta of computation time to be preempted, we model it as a sequence of atomic ‘ a ’ actions, each consuming one quantum [5]. To avoid prefixing q copies of primitive action a , we allow the following syntactic shorthand [3].

$$a^q \cdot E \stackrel{\text{def}}{=} \overbrace{a \cdot (a \cdot \dots (a \cdot E))}^{q \text{ times}}$$

In the examples below we model static-priority scheduling by associating a priority with each action name. Let the relation ‘ \prec ’ define a partial order of action priorities. Expression ‘ $a \prec b$ ’ tells us that action b has higher priority than action a , and ‘ $a \asymp b$ ’ states that a and b have equal priority.

3.2 Equivalence Laws

Since each agent expression now has an associated time and incoming-message context, we introduce the following laws to show how these contexts distribute through the various agent constructors.

- (1) $(t,I)(P + Q) = (t,I)P + (t,I)Q$
- (2) $(t,I)(P \mid Q) = (t,I)P \mid (t,I)Q$
- (3) $(t,I)(P \upharpoonright Q) = (t,I)P \upharpoonright (t,I)Q$
- (4) $(t,I)(P \setminus L) = ((t,I)P) \setminus L$
- (5) $(t,I)(P[f]) = ((t,I)P)[f]$
- (6) $(t,I)\mathbf{0} = \mathbf{0}$
- (7) $(t,I)((u,J)(x.P)) = (\max(t,u),I \cup J)(x.P)$
- (8) $(t,I \cup \{i@t\})P = (t,I)P$ if i can never appear in P

Laws (1) to (5) tell us that contexts distribute in the obvious way through the main agent constructors [1]. Law (6) tells us that contexts are redundant for the nil agent $\mathbf{0}$ since it cannot perform any actions in any context. Law (7) relates to the prefix operator, which is the only one that directly performs actions in CCS, and shows how nested contexts are merged. If agent ' $x.P$ ' has two earliest starting times, t and u , then its earliest starting time is the later of the two. If the agent has two sets of incoming messages, I and J , then its complete set of incoming messages is the union of the two. Lastly, Law (8) states that if agent P is incapable of ever receiving some incoming asynchronous message i , then we can remove i from P 's context. (For this to be applicable, i must not appear in P , any possible relabelling of P , or any agent into which P can evolve.)

So that it is not always necessary to explicitly provide a complete context for every agent expression, we allow the following syntactic equivalences.

- (9) $E \stackrel{\text{def}}{=}_{(0,\emptyset)} E$
- (10) ${}_t E \stackrel{\text{def}}{=}_{(t,\emptyset)} E$
- (11) ${}_I E \stackrel{\text{def}}{=}_{(0,I)} E$

Law (9) tells us that an agent expression E with no explicit context is equivalent to the agent in a context where the earliest starting time is 0 and there are no messages in transit. (Expression E may contain one or more contexts 'internally'.) Recall from Law (7) that adding context ' $(0,\emptyset)$ ' to an agent has no effect. Law (9) allows us to write agent expressions without explicit contexts below. In the same way, Law (10) allows us to provide just a starting time t , and Law (11) allows us to provide just an incoming-message set I , when desired.

We conclude this section with two frequently-used equational laws [15, Ch. 3] for our parallel and scheduling operators.

- (12) $P \mid \mathbf{0} = \mathbf{0} \mid P = P$
- (13) $P \upharpoonright \mathbf{0} = \mathbf{0} \upharpoonright P = P$

3.3 Operational Semantics

As usual, we define the operational semantics of our algebra via a labelled transition system [15, §2.5]. Let transition $E \xrightarrow{x@t} E'$ indicate that agent expression E is capable of performing action x at time t and then evolving into agent E' . Operator semantics are then defined by giving their complete set of transition rules. Each rule has a name, optional premises defining situations under which the rule may apply, a conclusion defining the transition that can be performed, and optional side conditions [15, p. 45].

$$\mathbf{name} \frac{\text{premises}}{\text{conclusion}} (\text{conditions})$$

To define the prefixing operator in our formalism, we must consider the effect of performing an action on the passage of time, and must treat an asynchronous receive action as a special case.

$$\mathbf{Act}_1 \frac{}{(t,I)(m \cdot E) \xrightarrow{m@t} (t+1,I)E}$$

$$\mathbf{Act}_2 \frac{}{(t,I)(i \cdot E) \xrightarrow{i@v} (v+1,I \setminus \{i@u\})E} \quad (i@u \in I \text{ and } v = \max(t, u))$$

Rule **Act**₁ tells us that an agent prefixed by action m , in context (t, I) , can perform action m at time t , and thereby evolve into an agent with context $(t+1, I)$. Incrementing the time in the context reflects our assumption that each atomic action consumes one quantum.

Rule **Act**₂ covers the special case of asynchronous receives. It states that an agent in context (t, I) , prefixed by an asynchronous receive action i , can perform action i at time v , and evolve into an agent with context $(v+1, I \setminus \{i@u\})$. The side condition requires that the initial incoming-message set I must contain a timestamped event $i@u$, indicating that the corresponding send action \bar{i} was completed at time u . (The way asynchronous messages are added to contexts is defined by rules **Par**₄ and **Par**₅, below.) Furthermore, the side condition states that the time of receipt v is the later of the local time t and the time u at which the message became available. The updated context subtracts event $i@u$ from set I , since this message has now been consumed.

The derivation rules for the summation, restriction, relabelling and agent constant definition operators are identical to standard CCS [15, p. 46], except that each action x is stamped with its time of occurrence t [1, 10].

$$\mathbf{Sum}_1 \frac{E \xrightarrow{x@t} E'}{E + F \xrightarrow{x@t} E'} \quad \mathbf{Sum}_2 \frac{F \xrightarrow{x@t} F'}{E + F \xrightarrow{x@t} F'}$$

$$\mathbf{Res} \frac{E \xrightarrow{x@t} E'}{E \setminus L \xrightarrow{x@t} E' \setminus L} \quad (x, \bar{x} \notin L) \quad \mathbf{Rel} \frac{E \xrightarrow{x@t} E'}{E[f] \xrightarrow{f(x)@t} E'[f]}$$

$$\mathbf{Con} \frac{P \xrightarrow{x@t} P'}{A \xrightarrow{x@t} P'} (A \stackrel{\text{def}}{=} P)$$

Similarly, the rules for parallel composition of independent actions and synchronous message-passing correspond to the untimed case [15, p. 46].

$$\mathbf{Par}_1 \frac{E \xrightarrow{r@t} E'}{E \mid F \xrightarrow{r@t} E' \mid F} \quad \mathbf{Par}_2 \frac{F \xrightarrow{r@t} F'}{E \mid F \xrightarrow{r@t} E \mid F'}$$

$$\mathbf{Par}_3 \frac{E \xrightarrow{a@t} E' \quad F \xrightarrow{\bar{a}@t} F'}{E \mid F \xrightarrow{\tau@t} E' \mid F'}$$

Rule **Par**₃ allows a synchronous communication event only when both agents agree on its time of occurrence t . Rules **Par**₁ and **Par**₂ allow (truly) parallel agents to evolve independently, each with their own contexts [1, 10].

However, an asynchronous message send \bar{i} is treated specially to allow for the addition of the message in transit to the context of the other agent.

$$\mathbf{Par}_4 \frac{E \xrightarrow{\bar{i}@t} E'}{E \mid F \xrightarrow{\bar{i}@t} E' \mid \{i@(t+1)\}F} \quad \mathbf{Par}_5 \frac{F \xrightarrow{\bar{i}@t} F'}{E \mid F \xrightarrow{\bar{i}@t} \{i@(t+1)\}E \mid F'}$$

In rule **Par**₄, for instance, the occurrence in agent E of an asynchronous send \bar{i} at time t , results in the timestamped event $i@(t+1)$ being added to the context of agent F . This indicates that the corresponding receive action i may now occur in F at time $t+1$ or later. (If agent F was itself composed of parallel agents, the effect is to *broadcast* message i to all of these agents—we assume that only one parallel operand contains a corresponding receive action and others will ignore the message. Furthermore, we did not add i to the context of the sending agent E here on the assumption that agents do not send messages to themselves, i.e., there is no intraprocessor asynchronous communication.)

The final set of rules define the behaviour of our new ‘intraprocessor scheduling’ operator. The first two tell us that when two agents are each ready to perform an action, then the scheduling operator favours the earlier possible action. In other words, the imaginary ‘processor’ will not stand idle while an action is ready.

$$\mathbf{Sch}_1 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@u} F'}{E \upharpoonright F \xrightarrow{x@t} E' \upharpoonright_{(t+1)} F} (t < u) \quad \mathbf{Sch}_2 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@u} F'}{E \upharpoonright F \xrightarrow{y@u} F' \upharpoonright_{(t+1)} E} (u < t)$$

In rule **Sch**₂, the order of the ‘ \upharpoonright ’ operands is reversed after agent F performs an action. The rules always place the agent that has just performed an action on the left. In this way the structure of the agent expression remembers the order in which scheduled agents have executed. This ensures that preempted agents resume execution in the proper sequence. Also, when one agent performs an action, the time in the context of the other agent is increased. Unlike parallel

composition, ‘scheduled’ agents share a common time-frame, so the passage of time in one affects both.

The next two rules tell us that when two agents are ready to perform actions at the same time, the scheduling operator favours higher-priority actions. Thus we model a priority-driven scheduling policy.

$$\mathbf{Sch}_3 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@t} F'}{E \upharpoonright F \xrightarrow{x@t} E' \upharpoonright_{(t+1)} F} (y \prec x) \quad \mathbf{Sch}_4 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@t} F'}{E \upharpoonright F \xrightarrow{y@t} F' \upharpoonright_{(t+1)} E} (x \prec y)$$

The next rule defines what happens when both agents can perform actions of the same priority at the same time. In this case the scheduling operator always favours its left-hand operand. Since this will usually be the agent that last performed an action, this models a ‘no unnecessary preemption’ scheduling policy in which tasks that have started executing are allowed to continue unless preempted by a strictly higher priority task.

$$\mathbf{Sch}_5 \frac{E \xrightarrow{x@t} E' \quad F \xrightarrow{y@t} F'}{E \upharpoonright F \xrightarrow{x@t} E' \upharpoonright_{(t+1)} F} (x \asymp y)$$

All the scheduling rules above applied to cases where both operands were ready to perform an action. Lastly, we must therefore explicitly say what happens when only one operand is ready. Let ‘ $E \nrightarrow$ ’ mean that agent E cannot perform any action.

$$\mathbf{Sch}_6 \frac{E \xrightarrow{x@t} E' \quad F \nrightarrow}{E \upharpoonright F \xrightarrow{x@t} E' \upharpoonright_{(t+1)} F} \quad \mathbf{Sch}_7 \frac{E \nrightarrow \quad F \xrightarrow{x@t} F'}{E \upharpoonright F \xrightarrow{x@t} F' \upharpoonright_{(t+1)} E}$$

Rules with negative premises are not normally needed in CCS because an agent that cannot perform any action is equivalent to the nil agent $\mathbf{0}$ and can be eliminated. This is not, however, the case with asynchronous message passing. An agent that is awaiting a message may not be able to perform an action in its current context, but may later be able to perform a receive action in an updated context. Agents waiting for asynchronous messages must therefore be distinguished from the nil agent which can never perform any action, in any context. (Negative premises must be introduced with care since they can result in ill-defined, ‘circular’ transition system definitions [18].)

4 Examples

This section presents a series of examples illustrating some of the capabilities of the language defined above.

4.1 Preemption and Offsets

The asymmetry of our scheduling operator allows it to be used to model task preemption, and the time contexts can be used to model starting-time offsets.

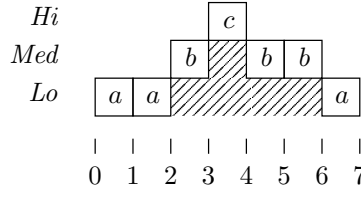


Fig. 1. Timeline for preemptive scheduling example.

Consider a system in which three tasks are required to execute on the same processor. A low-priority task requires 3 quanta, starting at time 0; a medium-priority task requires 3 quanta, starting at time 2; and a high-priority task requires 1 quantum, starting at time 3. We can model this system as three agents, assuming that action priorities obey the ordering $a \prec b \prec c$.

$$Lo \stackrel{\text{def}}{=} {}_0(a^3 \cdot \mathbf{0}) \quad Med \stackrel{\text{def}}{=} {}_2(b^3 \cdot \mathbf{0}) \quad Hi \stackrel{\text{def}}{=} {}_3(c \cdot \mathbf{0})$$

The following derivation steps show the behaviour of these three agents when composed using our scheduling operator. The sequence of timestamped events can be seen down the left. This is shown graphically in Figure 1. The derivation rules from Section 3.3 used at each step appear on the right. Laws from Section 3.2 are applied frequently and noted in interesting cases only.

$$\begin{array}{l}
Lo \upharpoonright Med \upharpoonright Hi \\
\frac{a@0}{\rightarrow} {}_1(a^2 \cdot \mathbf{0}) \upharpoonright {}_1 Med \upharpoonright {}_1 Hi \quad [\mathbf{Con}, \mathbf{Act}_1 \text{ and } \mathbf{Sch}_1 \text{ (twice)}] \\
\frac{a@1}{\rightarrow} {}_2(a \cdot \mathbf{0}) \upharpoonright {}_2 Med \upharpoonright {}_2 Hi \quad [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_1 \text{ (twice)}] \\
\frac{b@2}{\rightarrow} {}_3(b^2 \cdot \mathbf{0}) \upharpoonright {}_3(a \cdot \mathbf{0}) \upharpoonright {}_3 Hi \quad [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_4 \text{ and } \mathbf{Sch}_1] \\
\frac{c@3}{\rightarrow} {}_4(b^2 \cdot \mathbf{0}) \upharpoonright {}_4(a \cdot \mathbf{0}) \quad [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_4 \text{ and } (13)] \\
\frac{b@4}{\rightarrow} {}_5(b \cdot \mathbf{0}) \upharpoonright {}_5(a \cdot \mathbf{0}) \quad [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_3] \\
\frac{b@5}{\rightarrow} {}_6(a \cdot \mathbf{0}) \quad [\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } (13)] \\
\frac{a@6}{\rightarrow} \mathbf{0} \quad [\mathbf{Act}_1 \text{ and } (6)]
\end{array}$$

These steps model priority-based task preemption. After performing its first two ‘a’ quanta, the *Lo* agent is preempted by the ability of the *Med* agent to perform its first *b* action at time 2. The *Med* agent is itself preempted by the *Hi* agent at time 3. The shaded areas in Figure 1 thus show where a task was ready to perform an action but was preempted by a higher-priority one. Note that when a preempting task finishes, the lower-priority tasks resume execution in the correct sequence.

4.2 Repetitive Behaviours

Scheduling theory usually assumes that each task performs an infinite sequence of invocations, with the arrivals of successive invocations separated by a known

minimum time [2]. We can model repeated task arrivals using recursive agent constant definitions. Just as standard CCS allows agent constants to be parameterised via subscripts [15, §2.8], we allow them to be parameterised with contexts.

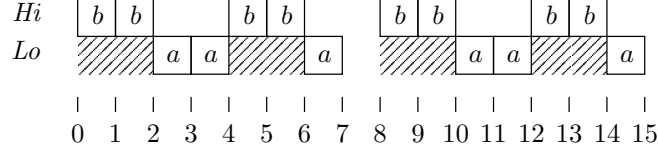


Fig. 2. Timeline for periodic scheduling example.

Consider a system consisting of two tasks. A low-priority task arrives every 8 time units and requires 3 quanta at each invocation, and a high-priority task arrives every 4 time units and requires 2 quanta at each invocation. This can be modelled via two recursive agent definitions, each parameterised with a time context t , assuming action priorities such that $a \prec b$.

$${}_tLo \stackrel{\text{def}}{=} {}_t(a^3 \cdot ({}_{t+8}Lo)) \quad {}_tHi \stackrel{\text{def}}{=} {}_t(b^2 \cdot ({}_{t+4}Hi))$$

When composed using the scheduling operator, the derivation tree for this system is infinite. Its first seven steps are as follows, after which the cycle repeats. The behaviour of the system is illustrated graphically in Figure 2.

$$\begin{array}{l}
{}_0Lo \upharpoonright {}_0Hi \\
\frac{b@0}{\longrightarrow} {}_1(b \cdot {}_4Hi) \upharpoonright {}_1(a^3 \cdot {}_8Lo) \quad [\mathbf{Con}, \mathbf{Act}_1 \text{ and } \mathbf{Sch}_4] \\
\frac{b@1}{\longrightarrow} {}_4Hi \upharpoonright {}_2(a^3 \cdot {}_8Lo) \quad [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_3] \\
\frac{a@2}{\longrightarrow} {}_3(a^2 \cdot {}_8Lo) \upharpoonright {}_4(b^2 \cdot {}_8Hi) \quad [\mathbf{Con}, \mathbf{Act}_1 \text{ and } \mathbf{Sch}_2] \\
\frac{a@3}{\longrightarrow} {}_4(a \cdot {}_8Lo) \upharpoonright {}_4(b^2 \cdot {}_8Hi) \quad [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_1] \\
\frac{b@4}{\longrightarrow} {}_5(b \cdot {}_8Hi) \upharpoonright {}_5(a \cdot {}_8Lo) \quad [\mathbf{Con}, \mathbf{Act}_1 \text{ and } \mathbf{Sch}_4] \\
\frac{b@5}{\longrightarrow} {}_8Hi \upharpoonright {}_6(a \cdot {}_8Lo) \quad [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_3] \\
\frac{a@6}{\longrightarrow} {}_8Lo \upharpoonright {}_8Hi \quad [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_2]
\end{array}$$

4.3 Blocking on Shared Resources

When tasks lock shared resources in static-priority scheduling, their active priority is temporarily raised, to hasten their execution, so that they do not unnecessarily block higher-priority tasks [2, 6]. We can model this phenomenon by allowing a low-priority agent to perform high-priority actions, to represent periods when a resource shared with a high-priority task is locked.

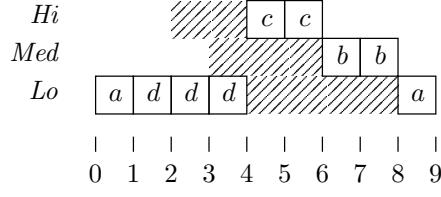


Fig. 3. Timeline for blocking example.

Consider a system of three tasks in which the lowest and highest priority tasks both access the same shared resource. A low-priority task arrives at time 0, requiring 5 quanta, but during its 2nd, 3rd and 4th quanta it needs access to the shared resource; a medium-priority task arrives at time 3 and requires 2 quanta; and a high-priority task arrives at time 2 and requires 2 quanta. This can be modelled using the following three agents. Assume that the priorities of actions are as follows: $a \prec b \prec c \asymp d$. Action d represents times when the low-priority task has locked the shared resource, and is effectively executing at the same priority as the high-priority task.

$$Lo \stackrel{\text{def}}{=} {}_0(a.(d^3.(a.\mathbf{0}))) \quad Med \stackrel{\text{def}}{=} {}_3(b^2.\mathbf{0}) \quad Hi \stackrel{\text{def}}{=} {}_2(c^2.\mathbf{0})$$

The behaviour of these agents when executing on the same processor is modelled by the following derivation. This is illustrated graphically in Figure 3.

$$\begin{array}{l}
Hi \upharpoonright Med \upharpoonright Lo \\
\begin{array}{l}
\frac{a@0}{\longrightarrow} {}_1(d^3.(a.\mathbf{0})) \upharpoonright {}_1Hi \upharpoonright {}_1Med \quad [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_2 \text{ and } (3)] \\
\frac{d@1}{\longrightarrow} {}_2(d^2.(a.\mathbf{0})) \upharpoonright {}_2Hi \upharpoonright {}_2Med \quad [\mathbf{Act}_1, \mathbf{Sch}_1 \text{ and } (3)] \\
\frac{d@2}{\longrightarrow} {}_3(d.(a.\mathbf{0})) \upharpoonright {}_3Hi \upharpoonright {}_3Med \quad [\mathbf{Act}_1, \mathbf{Sch}_5 \text{ and } (3)] \\
\frac{d@3}{\longrightarrow} {}_4(a.\mathbf{0}) \upharpoonright {}_4Hi \upharpoonright {}_4Med \quad [\mathbf{Act}_1, \mathbf{Sch}_5 \text{ and } (3)] \\
\frac{c@4}{\longrightarrow} {}_5(c.\mathbf{0}) \upharpoonright {}_5Med \upharpoonright {}_5(a.\mathbf{0}) \quad [\mathbf{Con}, \mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } \mathbf{Sch}_4] \\
\frac{c@5}{\longrightarrow} {}_6Med \upharpoonright {}_6(a.\mathbf{0}) \quad [\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } (13)] \\
\frac{b@6}{\longrightarrow} {}_7(b.\mathbf{0}) \upharpoonright {}_7(a.\mathbf{0}) \quad [\mathbf{Con}, \mathbf{Act}_1 \text{ and } \mathbf{Sch}_3] \\
\frac{b@7}{\longrightarrow} {}_8(a.\mathbf{0}) \quad [\mathbf{Act}_1, \mathbf{Sch}_3 \text{ and } (13)] \\
\frac{a@8}{\longrightarrow} \mathbf{0} \quad [\mathbf{Act}_1 \text{ and } (6)]
\end{array}
\end{array}$$

At time 2 the Hi agent is ready to perform action c . However, because the Lo agent is already performing equal-priority d actions, i.e., has locked the shared resource, it continues to get preference. This is achieved in the formalism by making the Lo agent the left-hand, favoured operand of the scheduling operator. (Letting the high-priority task start executing would break the lock on the shared resource.) The high-priority task thus experiences ‘direct’ blocking [6] at

time 2. At time 3 the *Med* agent becomes ready to perform action *b*, but cannot because the *Lo* agent's *d* actions have higher-priority. This is an example of 'push-through' blocking—the medium-priority task is blocked even though it does not use the shared resource [6]. Once the *Lo* agent stops executing high-priority *d* actions, i.e., unlocks the shared resource, it is immediately preempted.

4.4 Multiprocessor Scheduling

In complex embedded systems, sequences of task invocations are linked to form end-to-end transactions. The tasks comprising a transaction may reside on different processors, so interprocessor precedence constraints must be enforced to ensure that they execute in the correct sequence. We can use our (true concurrency) parallelism operator to model multiple processors, and asynchronous communication to model precedence constraints.

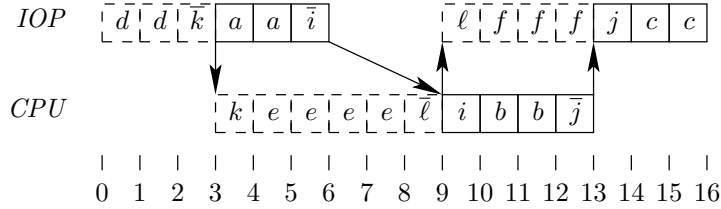


Fig. 4. Timeline for multiprocessor example.

Consider part of an embedded avionics system [9, §4.3.2.1]. A low-priority transaction consists of three task invocations. The first task performs sensor input operations and requires 2 quanta; the second task processes the data and requires 2 quanta; and the third task performs actuator output operations and requires 2 quanta. To ensure that the three tasks synchronise correctly, when the first task finishes it sends a message *i* to the second, and when the second finishes it sends a message *j* to the third. This can be modelled by the following three agents.

$$LoIn \stackrel{\text{def}}{=} a^2 . (\bar{i} . \mathbf{0}) \quad LoProc \stackrel{\text{def}}{=} i . (b^2 . (\bar{j} . \mathbf{0})) \quad LoOut \stackrel{\text{def}}{=} j . (c^2 . \mathbf{0})$$

A high-priority transaction is structured similarly. Its input task requires 2 quanta; its processing task requires 4 quanta; and its output task requires 3 quanta. Asynchronous messages *k* and *l* are used for task synchronisation.

$$HiIn \stackrel{\text{def}}{=} d^2 . (\bar{k} . \mathbf{0}) \quad HiProc \stackrel{\text{def}}{=} k . (e^4 . (\bar{l} . \mathbf{0})) \quad HiOut \stackrel{\text{def}}{=} l . (f^3 . \mathbf{0})$$

We assume that all actions in the high-priority transaction have priority over all actions in the low-priority one: $\{a, b, c, i, j\} \prec \{d, e, f, k, l\}$.

Next the tasks are assigned to processors. All the input and output tasks are scheduled on a dedicated I/O Processor, and the two data processing tasks

$$\begin{array}{ll}
\frac{i@9}{10(b^2 \cdot (\bar{j} \cdot \mathbf{0}))} \{ \ell@9 \} ({}_6LoOut \upharpoonright {}_6HiOut) \mid & [\mathbf{Con}, \mathbf{Act}_2 \text{ and } \mathbf{Par}_2] \\
\frac{\ell@9}{10(b^2 \cdot (\bar{j} \cdot \mathbf{0}))} ({}_{10}(f^3 \cdot \mathbf{0}) \upharpoonright {}_{10}LoOut) \mid & [\mathbf{Con}, \mathbf{Act}_2, \mathbf{Sch}_7, \mathbf{Par}_1 \text{ and } (8)] \\
\frac{b@10}{10(f^3 \cdot \mathbf{0}) \upharpoonright {}_{10}LoOut} \mid {}_{11}(b \cdot (\bar{j} \cdot \mathbf{0})) & [\mathbf{Act}_1 \text{ and } \mathbf{Par}_2] \\
\frac{b@11}{10(f^3 \cdot \mathbf{0}) \upharpoonright {}_{10}LoOut} \mid {}_{12}(\bar{j} \cdot \mathbf{0}) & [\mathbf{Act}_1 \text{ and } \mathbf{Par}_2] \\
\frac{f@10}{11(f^2 \cdot \mathbf{0}) \upharpoonright {}_{11}LoOut} \mid {}_{12}(\bar{j} \cdot \mathbf{0}) & [\mathbf{Act}_1, \mathbf{Sch}_6 \text{ and } \mathbf{Par}_1] \\
\frac{\bar{j}@12}{11(f^2 \cdot \mathbf{0}) \upharpoonright ({}_{11}, \{j@13\})LoOut} & [\mathbf{Act}_1, \mathbf{Par}_2, (12) \text{ and } (8)] \\
\frac{f@11}{12(f \cdot \mathbf{0}) \upharpoonright ({}_{12}, \{j@13\})LoOut} & [\mathbf{Act}_1 \text{ and } \mathbf{Sch}_6] \\
\frac{f@12}{({}_{13}, \{j@13\})LoOut} & [\mathbf{Act}_1, \mathbf{Sch}_6, (6) \text{ and } (13)] \\
\frac{j@13}{14(c^2 \cdot \mathbf{0})} & [\mathbf{Con} \text{ and } \mathbf{Act}_2] \\
\frac{c@14}{15(c \cdot \mathbf{0})} & [\mathbf{Act}_1] \\
\frac{c@15}{\mathbf{0}} & [\mathbf{Act}_1 \text{ and } (6)]
\end{array}$$

Initially only the I/O Processor's agents can make progress, and the *HiIn* agent is given preference. Note that while the *HiIn* agent is performing actions locally, time does not advance in the parallel *CPU* agent. At time 2 the *HiIn* agent performs action \bar{k} which sends message k . This results in event $k@3$ being added to the context of the *CPU* agent, which enables the *HiProc* agent. In the particular interleaving shown above, event $k@3$ in the *CPU* agent occurs *after* event $a@4$ in the *IOP* agent. This is permissible because the true-concurrency timing model requires that only causally-related events must appear in chronological sequence [1, 10]. However, when all the timestamped events are displayed as in Figure 4, it can be seen that the overall derivation is causally consistent. After the asynchronous send action \bar{i} occurs at time 5, the *LoProc* agent is enabled, but it does not accept this message and begin executing until time 9, because the *HiProc* agent is preempting it. This separation between sends and receives demonstrates the need for asynchronous communication to model multi-processor precedence constraints.

5 Discussion and Future Work

The formalism presented above can be enhanced in a number of ways. For instance, rather than assuming each task always requires its worst-case quanta of computation time, we can use summation to define an abbreviation which allows tasks to have both minimum and maximum execution times [3, 10].

We assumed static-priority scheduling above since this is the default in programming languages such as Ada. However, we could easily adopt Ben-Abdallah et al.'s approach for modelling dynamic-priority scheduling in which each action incorporates an expression defining its current dynamic priority [3].

We also assumed that interprocessor precedence constraints were enforced via asynchronous message-passing, but in some applications it may be preferable to use interrupts. This can be modelled by changing our asynchronous communication mechanism to treat receives as very high-priority actions that always occur as soon as they can.

More significantly, our algebra must be completed by defining its notion of weak equivalence and a full set of equational laws [15]. Many such laws are straightforward modifications of their standard CCS counterparts, but entirely new laws are needed for the scheduling operator and asynchronous communication. This work is ongoing.

In practice, the full potential of a formalism is achieved only when support tools become available for it. Model checking timed automata has been the subject of much research, but has been hindered by semantics in which parallel transitions are linked by the passage of time [16]. The true concurrency model used here allows non-causally-related transitions to occur independently, and even out of their chronological sequence [1, 10], and is thus particularly amenable to efficient partial order reduction model checking techniques.

6 Conclusion

We have shown how multi-tasking behaviour can be modelled in a process algebra. This was done by starting with a timed, causal CCS variant and then introducing both an asymmetric interleaving operator, to model intraprocessor scheduling decisions, and asynchronous communication, to model interprocessor precedence constraints. The resulting formalism was shown to be capable of representing a wide variety of scheduling behaviours.

Acknowledgements Thanks to Antonio Cerone, Padmanabhan Krishnan and the anonymous referees for their comments. This work began while the author was visiting the University of York's Real-Time Systems Research group, courtesy of a University of Queensland Travel Award. Thanks to Andy Wellings for hosting the visit and Iain Bate for helpful discussions. This research was funded by the Information Technology Division of the Australian Defence Science and Technology Organisation.

References

1. L. Aceto and D. Murphy. Timing and causality in process algebra. *Acta Informatica*, 33:317–350, 1996.
2. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
3. H. Ben-Abdallah, J.-Y. Choi, D. Clarke, Y. S. Kim, I. Lee, and H.-L. Xie. A process algebraic approach to the schedulability analysis of real-time systems. *Real-Time Systems*, 15:189–219, 1998.

4. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer-Verlag, 1998.
5. L. Breveglieri, S. Crespi-Reghizzi, and A. Cherubini. Modeling operating systems schedulers with multi-stack-queue grammars. In G. Ciobanu and G. Păun, editors, *Fundamentals of Computation Theory (FCT'99)*, volume 1684 of *Lecture Notes in Computer Science*, pages 161–172. Springer-Verlag, 1999.
6. G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.
7. J. C. Corbett. Timing analysis of Ada tasking programs. *IEEE Transaction on Software Engineering*, 22(7):461–483, July 1996.
8. J. S. Dong, N. Fulton, L. Zucconi, and J. Colton. Formalising process scheduling requirements for an aircraft operational flight program. In *Proc. IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 161–169. IEEE Press, November 1997.
9. J. D. G. Falardeau. Schedulability analysis in rate monotonic based systems with application to the CF-188. Master's thesis, Department of Electrical and Computer Engineering, Royal Military College of Canada, May 1994.
10. C. J. Fidge and J. J. Žic. An expressive real-time CCS. In *Proc. Second Australasian Conference on Parallel and Real-Time Systems (PART'95)*, pages 365–372, Fremantle, September 1995.
11. P.-A. Hsiung, F. Wang, and Y.-S. Kuo. Scheduling system verification. In W. R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag, 1999.
12. D. M. Jackson. Experiences in embedded scheduling. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 445–464. Springer-Verlag, 1996.
13. J. Jacky. Analyzing a real-time program in Z. In *Proc. Z User's Meeting (ZUM'98)*, 1998.
14. Z. Liu, M. Joseph, and T. Janowski. Verification of schedulability for real-time programs. *Formal Aspects of Computing*, 7(5):510–532, 1995.
15. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
16. M. Minea. Partial order reduction for model checking of timed automata. In J. C. M. Baeten and S. Mauw, editors, *CONCUR'99: Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 431–446. Springer-Verlag, 1999.
17. R. van Glabbeek and P. Rittgen. Scheduling algebra. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 278–292. Springer-Verlag, 1999.
18. R. J. van Glabbeek. The meaning of negative premises in transition system specifications II. In F. Meyer auf der Heide and B. Monien, editors, *Automata, Languages and Programming, 23rd International Colloquium*, volume 1099 of *Lecture Notes in Computer Science*, pages 502–513. Springer-Verlag, 1996. Extended abstract.
19. Z. Yuhua and Z. Chaochen. A formal proof of the deadline driven scheduler. In H. Langmaack, W.-P. de Roever, and J. Vytupil, editors, *Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 756–775. Springer-Verlag, 1994.