Dynamic priorities for modeling real-time

Girish Bhat, Rance Cleaveland
Department of Computer Science, North Carolina State University
Raleigh, NC 27695-8206, USA, e-mail: {gsbhat1,rance}@eos.ncsu.edu

Gerald Lüttgen
Fakultät für Mathematik und Informatik, Universität Passau
D-94030 Passau, Germany, e-mail: luettgen@fmi.uni-passau.de

Abstract

This paper describes an approach for modeling real-time systems using *dynamic priorities*. The advantage of the technique is that it drastically reduces the state space sizes of the systems in question while preserving properties of their functional behavior. We demonstrate the utility of our approach by formally modeling and verifying aspects of the widely-used *SCSI-2 bus-protocol*. It turns out that the state space of this model is about an order of magnitude smaller than the one resulting from traditional real-time semantics.

Keywords

model checking, modeling, process algebra with priority and real-time, SCSI-2 bus-protocol, verification

1 INTRODUCTION

A variety of formal approaches have been introduced for modeling and verifying distributed systems including *process-algebraic* frameworks (Milner 1989) and *model checking* (Clarke et al. 1986, Kozen 1983). However, only with the advent of verification tools (Bengtsson et al. 1995, Cleaveland & Sims 1996, Holzmann 1991) in the last decade they have emerged as practical aids for system designers (Baeten 1990, Cleaveland et al. 1996, Elseaidy et al. 1996). This paper addresses the problem of modeling and verifying concurrent systems where *real-time* plays an important role for functional behavior. On the one hand, real-time is used to implement abstract *synchronization constraints* in distributed environments. As an example of a synchronization constraint, consider a communication protocol where the next protocol phase may be entered only if some or all components agree. On the other hand, electric phenomena like *wire glitches*, that may lead to malfunction, can be avoided using *deskew delays*. Thus, for accurately modeling such systems it is necessary to capture their real-time aspects, thereby motivating the need for implementing real-time process algebras (Moller & Tofts 1990, Yi 1991) efficiently.

Traditional implementations of real-time process algebras typically cause state spaces to explode, the reason for this being that time is considered as part of the state, i.e. a new state is generated for every clock tick. We tackle this problem by using *dynamic priorities* to model real-time. We introduce a new process algebra, called $\mathtt{CCS^{dp}}$ (CCS with dynamic priorities), which essentially extends CCS (Milner 1989) by assigning priorities to actions. Unlike traditional process algebras with priorities (e.g. Cleaveland & Hennessy 1990), actions in our algebra do not have fixed or *static* priorities; priorities may change as systems evolve. It is in this sense that we refer to $\mathtt{CCS^{dp}}$ as a process algebra with *dynamic* priorities. In contrast to traditional real-time algebras, e.g. a version of Temporal CCS (Moller & Tofts 1990), which we refer to as $\mathtt{CCS^{rt}}$ (CCS with real-time), the $\mathtt{CCS^{dp}}$ semantics interprets delays preceding actions as priority values attached to these actions. In other words, the longer the delay preceding an action, the lower is its priority. The semantics of $\mathtt{CCS^{dp}}$ avoids the unfolding of delay values into sequences of elementary steps, each consuming one time unit, thereby providing a formal foundation for *efficiently* modeling real-time. The soundness and completeness of this approach is proven by establishing a one-to-one correspondence between the $\mathtt{CCS^{rt}}$ and the $\mathtt{CCS^{dp}}$ semantics of arbitrary systems. It is important to note that our approach does not abstract away aspects of real-time. Thus, all quantitative timing constraints explicit in $\mathtt{CCS^{rt}}$ semantics can still be analyzed within $\mathtt{CCS^{dp}}$ semantics.

The utility of our technique is shown by means of a practical example, namely modeling and verifying several aspects of the SCSI-2 bus-protocol, a protocol used in many of today's computers. The protocol's model is derived from the official standard (ANSI 1994) where real-time delays are recommended for implementing synchronization constraints as well as for ensuring correct behavior in the presence of signal glitches. Accurate modeling of the SCSI-2 bus-protocol thus requires considering discrete quantitative real-time. To this end, we model our protocol in the syntax common to both $\mathtt{CCS^{rt}}$ and $\mathtt{CCS^{dp}}$. We then generate the state space according to both semantics. We show that the size of our model is an order of magnitude smaller in the $\mathtt{CCS^{dp}}$ semantics than in the $\mathtt{CCS^{rt}}$ semantics. The modeling of the protocol was carried out in the *Concurrency Workbench of North Carolina* (Cleaveland & Sims 1996), `CWB-NC`, a tool for analyzing and verifying concurrent systems. In order to verify and to prove the accuracy of our model, we extract several mandatory properties from the ANSI document and validate them for our model. We use the well-known *modal μ-calculus* as our specification language, and automatically check the formalized properties by using the *local model checker* (Bhat & Cleaveland 1996) integrated in the `CWB-NC`. Due to space constraints all proofs and part of the formalization of the SCSI-2 case study are left out and can be found in a technical report (Bhat et al. 1997).

2 PROCESS-ALGEBRAIC FRAMEWORK

In this section we introduce the process algebra $\mathtt{CCS^{rt}}$ and develop the new process algebra $\mathtt{CCS^{dp}}$, which has the same syntax but different semantics. Whereas $\mathtt{CCS^{rt}}$ is an extension of CCS (Milner 1989) in order to capture *discrete quantitative timing aspects* with respect to a single, global clock, $\mathtt{CCS^{dp}}$ extends CCS by a concept of *dynamic priorities.* Our syntax differs from CCS by associating delay and priority values with actions, respectively, and by including the *disabling* operator known from LOTOS (Bolognesi & Brinksma 1987).

Formally, let Λ be a countable set of *action labels* or *ports*, not including the so-called *silent* or *internal* action τ. With every $a \in \Lambda$ we associate a *complementary action* $\overline{a}$. Intuitively, an action $a \in \Lambda$ may be thought of as representing the receipt of an input on port a, while $\overline{a}$ constitutes the deposit of an output on a. We define $\overline{\Lambda} =_{\mathrm{df}} \{\overline{a} \mid a \in \Lambda\}$ and take $\mathcal{A}$ to denote the set of all actions $\Lambda \cup \overline{\Lambda} \cup \{\tau\}$. In what follows, we let $a, b, \ldots$ range over $\Lambda \cup \overline{\Lambda}$ and $\alpha, \beta, \ldots$ over $\mathcal{A}$. Complementation is lifted to actions in $\Lambda \cup \overline{\Lambda}$, also called *visible actions*, by defining $\overline{\overline{a}} =_{\mathrm{df}} a$. As in CCS an action a communicates with its complement $\overline{a}$ to produce the internal action τ.

In our syntax actions are associated with *delay values*, or *priority values*, taken from the natural numbers, respectively. More precisely, the notation $\alpha : k$, where $\alpha \in \mathcal{A}$ and $k \in \mathbb{N}$, specifies that action α is ready for execution after a minimum delay of k time units or, respectively, that action α possesses priority k. In the priority interpretation, smaller numbers encode higher priority values; so 0 represents the highest priority. The syntax of our language is defined by the following BNF:

$$
\begin{array}{llllllllll}
P & ::= & nil & \mid & \alpha : k.P & \mid & P + P & \mid & P \,\rangle\!\!\rangle P & \mid \\
 & & P|P & \mid & P[f] & \mid & P\backslash L & \mid & C &
\end{array}
$$

where $k \in \mathbb{N}$, the mapping $f : \mathcal{A} \to \mathcal{A}$ is a *relabeling*, $L \subseteq \mathcal{A}\backslash\{\tau\}$ is a *restriction set*, and C is a *process constant* whose meaning is given by a defining equation $C \stackrel{\mathrm{def}}{=} P$. A relabeling f satisfies the properties $f(\tau) = \tau$ and $f(\overline{a}) = \overline{f(a)}$. We adopt the usual definitions for *closed* terms and *guarded* recursion, and refer to the closed guarded *terms* as *processes.* Let $\mathcal{P}$ represent the set of all processes, ranged over by $P, Q, R, \ldots$.

Regarding the semantics of processes we first introduce a real-time semantics, referred to as $\mathtt{CCS^{rt}}$ semantics, which explicitly represents timing behavior. We concentrate here on the operational semantics for our notion of prefixing since the semantics of the other operators is standard (Moller & Tofts 1990). Formally, the semantics of a process is defined by a labeled transition system which contains explicit time transitions – each representing a delay of one time unit – as well as action transitions. With respect to time transitions, the operational semantics is set up such that processes willing to communicate with some process running in parallel are able to wait until the communication partner is ready. However, as soon as it is available the communication has to take place, i.e. further idling is prohibited. This assumption is usually referred to as *maximal progress assumption* (Yi 1991). Accordingly, the process $\alpha\!:\!k.P$, where $k > 0$, may delay one time unit and then behave like $\alpha\!:\!(k-1).P$. The process $\alpha\!:\!0.P$ performs an α transition leading to P. Moreover, if $\alpha \neq \tau$, it may also idle by performing a time transition to itself. Unfortunately, $\texttt{CCS}^{\texttt{rt}}$ semantics unfolds every delay value into a sequence of elementary time units, thereby creating many additional states. For example, the process $\alpha\!:\!k.nil$ has $k+2$ states, namely *nil* and $\alpha\!:\!l.nil$ where $0 \leq l \leq k$. It would be much more efficient if we could represent $\alpha : k.nil$ by a single transition labeled by $\alpha : k$ leading to the state *nil*. This idea of compacting the state space of real-time systems can be realized by viewing k as a *priority value* assigned to action α. In other words, one may consider the delay value k as the time stamp of action α.

To this end, we present a new semantics for our language that uses a notion of priority taken from Cleaveland & Hennessy (1990), generalized to a *multi-level* priority scheme. We refer to our process algebra as $\texttt{CCS}^{\texttt{dp}}$ when interpreted with respect to the new semantics which, in contrast to the priority approach mentioned above, dynamically adjusts priorities along transitions. Intuitively, visible actions represent potential synchronizations that a process may be willing to engage in with its environment. Given a choice between a synchronization on a high priority and one on a low priority, a process should choose the former. Thus, high-priority τ-actions have pre-emptive power over low-priority actions. The reason that high-priority visible actions do *not* have pre-emptive power over low-priority actions is that visible actions only indicate the potential of a synchronization, i.e. the potential of progress, whereas τ-actions describe complete synchronizations, i.e. *real* progress, in our model. Note that this notion of pre-emption naturally mimics the maximal progress assumption employed in $\texttt{CCS}^{\texttt{rt}}$ semantics.

Formally, the $\texttt{CCS}^{\texttt{dp}}$ semantics of a process $P \in \mathcal{P}$ is given by a labeled transition system $\langle \mathcal{P}, \mathcal{A} \times \mathbb{N}, \longrightarrow, P \rangle$ where $\mathcal{P}$ is the set of states, $\mathcal{A} \times \mathbb{N}$ the set of labels, $\longrightarrow$ the transition relation which is defined in Table 1 via structural operational rules, and P is the initial state. For the sake of simplicity we write $P \xrightarrow{\alpha:k} P'$ for $\langle P, \alpha : k, P' \rangle \in \longrightarrow$ and say that P engages in action α with priority k and thereafter behaves like process P'. The presentation of the operational rules requires two auxiliary definitions which are formally given in Appendix 1. First, we introduce *initial action sets* which are inductively defined on the syntax of processes as usual. More precisely, $\mathcal{I}^k(P)$ denotes the set of all potential initial actions of P with priority greater than k, where $\mathcal{I}^0(P)$ is defined to be the empty set. Second, we define a *priority adjustment function*. Intuitively, our semantics is set up in a way such that if one parallel component of a process engages in an action with priority k, then the priority values of all initial actions at every other parallel component have

Table 1 Operational semantics for $\mathtt{CCS}^{\mathtt{dp}}$

`Act1`	$\dfrac{-}{a\!:\!k.P \xrightarrow{a:l} P}\; l \geq k$	`Act2`	$\dfrac{-}{\tau\!:\!k.P \xrightarrow{\tau:k} P}$
`Sum1`	$\dfrac{P \xrightarrow{\alpha:k} P'}{P+Q \xrightarrow{\alpha:k} P'}\; \tau \notin \mathcal{I}^k(Q)$	`Sum2`	$\dfrac{Q \xrightarrow{\alpha:k} Q'}{P+Q \xrightarrow{\alpha:k} Q'}\; \tau \notin \mathcal{I}^k(P)$
`Dis1`	$\dfrac{P \xrightarrow{\alpha:k} P'}{P \rhd Q \xrightarrow{\alpha:k} P' \rhd [Q]^k}\; \tau \notin \mathcal{I}^k(Q)$	`Dis2`	$\dfrac{Q \xrightarrow{\alpha:k} Q'}{P \rhd Q \xrightarrow{\alpha:k} Q'}\; \tau \notin \mathcal{I}^k(P)$
`Com1`	$\dfrac{P \xrightarrow{\alpha:k} P'}{P\vert Q \xrightarrow{\alpha:k} P'\vert[Q]^k}\; \tau \notin \mathcal{I}^k(P\vert Q)$	`Rel`	$\dfrac{P \xrightarrow{\alpha:k} P'}{P[f] \xrightarrow{f(\alpha):k} P'[f]}$
`Com2`	$\dfrac{Q \xrightarrow{\alpha:k} Q'}{P\vert Q \xrightarrow{\alpha:k} [P]^k\vert Q'}\; \tau \notin \mathcal{I}^k(P\vert Q)$	`Res`	$\dfrac{P \xrightarrow{\alpha:k} P'}{P\backslash L \xrightarrow{\alpha:k} P'\backslash L}\; \alpha \notin L \cup \overline{L}$
`Com3`	$\dfrac{P \xrightarrow{a:k} P' \quad Q \xrightarrow{\overline{a}:k} Q'}{P\vert Q \xrightarrow{\tau:k} P'\vert Q'}\; \tau \notin \mathcal{I}^k(P\vert Q)$	`Con`	$\dfrac{P \xrightarrow{\alpha:k} P'}{C \xrightarrow{\alpha:k} P'}\; C \stackrel{\text{def}}{=} P$

to be decreased by k, i.e. those actions become 'more important.' Thus, the semantics of parallel composition deploys a kind of *fairness assumption*, and priorities have a *dynamic* character. More precisely, the priority adjustment function applied to a process $P \in \mathcal{P}$ and a natural number $k \in \mathbb{N}$, denoted as $[P]^k$, returns a process term which is 'identical' to P except that the priorities of the initial, top-level actions are decreased by k. Note that a priority value cannot become smaller than 0.

Intuitively, $a : k.P$ may engage in action a with priority $l \geq k$ yielding process P. The side condition $l \geq k$ reflects that k does not specify an exact priority but the *maximal* priority of the initial transition of $a : k.P$. It may also be interpreted as *lower-bound* timing constraint. Due to the notion of pre-emption incorporated in $\mathtt{CCS}^{\mathtt{dp}}$, $\tau : k.P$ may not perform the τ-transition with a lower priority than k. The summation operator $+$ denotes *non-deterministic* choice, i.e. the process $P+Q$ may behave like P (Q) if Q (P) does not pre-empt it by being able to engage in a higher prioritized internal transition. Thus, our notion of pre-emption reflects implicit *upper-bound* timing constraints. Also the process $P \rhd Q$ behaves like P and, additionally, it is capable of *disabling* P by engaging in Q. The *restriction* operator $\backslash L$ prohibits the execution of actions in $L \cup \overline{L}$ and thus permits the scoping of actions. $P[f]$ behaves exactly as P where actions are renamed by the *relabeling* f. The process $P|Q$

stands for the *parallel composition* of P and Q according to an interleaving semantics with synchronized communication on complementary actions of P and Q both having some priority k which results in the internal action $\tau:k$. The side conditions of the interleaving rules implement pre-emption. Finally, $C \stackrel{\text{def}}{=} P$ denotes a *constant definition*, i.e. C is a recursively defined process that is a distinguished solution of the equation $C = P$.

For our framework we obtain the following important results, which are formally stated and proved in a technical report (Bhat et al. 1997). Given an arbitrary process in our language there exists a one-to-one semantic correspondence between the associated transition systems according to $\mathtt{CCS}^{\mathtt{rt}}$ and to $\mathtt{CCS}^{\mathtt{dp}}$ semantics. Moreover, the standard strong bisimulations (Milner 1989), which can be defined straightforwardly for $\mathtt{CCS}^{\mathtt{rt}}$ and $\mathtt{CCS}^{\mathtt{dp}}$, coincide.

We conclude this section by discussing a related approach by Jeffrey (1992) who has established a formal relationship between a quantitative real-time process algebra and a process algebra with (*static*) priorities. He also translates real-time into priorities based on the idea of time stamping. In contrast to Temporal CCS semantics, a process modeled in Jeffrey's framework may either *immediately* engage in an action or idle forever. However, this semantics does not reflect our intuition about the semantic behavior of *reactive* systems, i.e. a process should wait until a desired communication partner becomes available instead of engaging in a 'livelock.' Only because of these counter-intuitive assumptions, Jeffrey does not need to choose a *dynamic* priority framework.

3 SCSI-2 – AN OVERVIEW

We demonstrate the utility of the process algebra $\mathtt{CCS}^{\mathtt{dp}}$ by a case study dealing with the bus-protocol of the widely-used *Small Computer System Interface* (ANSI 1994), or *SCSI* for short. The *SCSI bus* is designed to provide an efficient peer-to-peer I/O connection for peripheral devices such as disks, tapes, printers, processors, etc. It usually connects several of these devices with one *host adapter* which often resides on a computer's motherboard. In contrast to the host adapter, peripherals are not attached directly to the bus but via *SCSI controllers*, also called *logical units* (LUNs). Thus, LUNs provide the physical and logical interface between the bus and the peripherals. Conceptually, up to seven LUNs can be connected to one bus, and one LUN can support up to seven peripherals. However, in practice most peripherals contain their own SCSI controller (cf. Figure 1).

The *SCSI-2 bus-protocol* implements the logical mechanism regulating how peripherals and the host adapter communicate with each other on the bus. Communication on the SCSI bus is point-to-point, i.e. at any time either none or exactly two LUNs may communicate with each other. In order to allow easy addressing each LUN is assigned a fixed SCSI id in form of a number ranging from one to seven. Id 0 is reserved for the host adapter which is also,

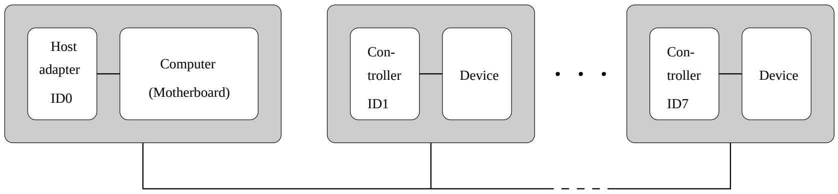

Figure 1 Typical SCSI configuration

conceptually, a LUN. Communication on the bus is organized by the use of eight signal lines whereas the actual information, like *messages*, *commands*, *data*, and *status information*, are transferred over a data bus.

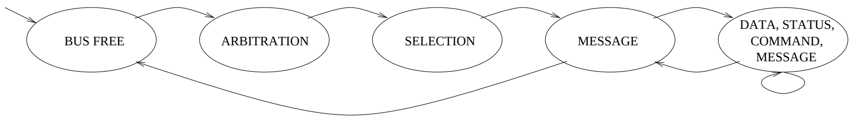

Figure 2 Usual progression of the SCSI-2 bus-phases

The SCSI-2 bus-protocol is organized in eight distinct phases, called `Bus Free`, `Arbitration`, `Selection`, `Reselection`, `Command`, `Data`, `Status`, and `Message` Phase. At any given time, the SCSI bus is exactly in one phase. The usual progression of phases is shown in Figure 2. During the `Bus Free` Phase no device is in possession of the bus, i.e. LUNs may request access. If more than one device competes for the bus in order to initiate a communication, the one with the highest SCSI id is granted access. In the `Arbitration` Phase, every LUN that has posed a request determines if it has been granted access. All LUNs which lose may compete for the bus again later, whereas the winner, also referred to as *initiator*, proceeds to the `Selection` Phase. In this phase the initiator tries to connect to the desired destination, called *target*. When the link between initiator and target has been established, the so-called *information transfer phases*, including the `Command`, the `Data`, the `Status`, and the `Message` Phase are entered. In the `Command` Phase the target may request a command from the initiator. During a `Message` Phase information is exchanged between the initiator and the target concerning the bus-protocol itself. Finally, the `Status` Phase is used to transfer status information to the initiator upon completion of a command executed by the target. The key idea for accelerating communication on the bus, which has significantly contributed to the success of SCSI, is that the target can free the bus whenever it receives a time-intensive command from the initiator. As soon as the execution of such a command is finished, the target competes for the bus in order to transmit the result to the former initiator.

4 MODELING THE SCSI-2 BUS-PROTOCOL

In this section we model the SCSI-2 bus-protocol in our language. The syntax we use here is the one implemented in the *Concurrency Workbench of North Carolina* in which $\texttt{CCS}^{\texttt{rt}}$ and $\texttt{CCS}^{\texttt{dp}}$ are integrated as front ends. It slightly departs from the syntax introduced in Section 2 in that output actions $\overline{a} \in \overline{\Lambda}$ are notated as `'a`, the internal action τ as `t`, and process definitions $C \overset{\text{def}}{=} P$ as `proc C = P`. Moreover, we use the notation $\alpha(\texttt{obs}){:}k$ which, for the purposes of this section, may be interpreted as $\alpha{:}k$. Actions `obs` come into play in the next section where they serve as 'probes' for verification purposes.

Before we present the actual modeling of the SCSI-2 bus-protocol, we comment on some assumptions we imposed. First, we restrict ourselves to modeling two LUNs, called `LUN0` and `LUN1`, having id 0 and id 1, respectively. This is sufficient for dealing with the aspects of the SCSI-2 bus-protocol we are interested in. Note that even in the situation of two LUNs there exists competition for the bus. Moreover, we abstract away from time-out procedures and from the contents of most messages, commands, and data. These abstractions are justified since they do not affect the conceptual parts of the bus-protocol's behavior. For example, the sole purpose of a timeout is to determine if a target is alive or not. The contents of information sent over the bus, except from messages presenting the completion of some transmission, are only relevant for the device-specific part of LUNs but not for the bus-protocol itself. Additionally, the bus signals `BSY` (*busy*) and `SEL` (*select*) are *wired-or* signals in reality. However, we need not model this 'or'-behavior, since our model only deals with two LUNs, and just one LUN at a time can assert the `BSY` or `SEL` signal. Finally, all quantitative timing information occurring in the model is measured relative to a time unit of 5 ns, including *arbitration delays* (480 time units), *bus clear delays* (160 time units), *bus settle delays* (80 time units), *deskew delays* (9 time units), and *cable skew delays* (9 time units).

The underlying structure of the bus-protocol is explicitly reflected in our model. Each LUN connected to the bus is modeled as a separate parallel component containing models of the different bus phases as discussed in the previous section. The logical behavior of the bus control is implemented by bus signals. Each signal physically consists of a wire which we model as a separate process similar to a global Boolean variable. Note that signal delays are not modeled in the wires but in the operations used for transmitting information over the SCSI bus. Since we abstract away from the content of most information, we do not need to model each bit of the data bus. Hence, arbitration is modeled via a global variable which stores the highest id of all LUNs requesting access to the bus. Accordingly, the structure of our model, called `SCSIBus`, consists of the parallel composition of both LUNs, and the `BusSignals`, including the regular signals and the data path. Formally,

```
proc SCSIBus = (LUN0 | LUN1 | BusSignals) \ Restriction
```

Table 2 Modeling the bus signals and the data bus

```
proc BusSignals =   DataBus
                  | Arbitrator
                  | Off[setBSY/sset,relBSY/rel,isBSY/on,noBSY/off]
                  | Off[setSEL/sset,relSEL/rel,isSEL/on,noSEL/off]
                  | ...

proc Off  =   'off:0.Off + sset:0.On + rel:0.Off
proc On   =   'on:0.On   + sset:0.On + rel:0.Off

proc DataBus  = DataBus' [> release(obsrelease):0.DataBus
proc DataBus' =   placemsgIn(obsplace):0.'readmsgIn(obsread):0.DataBus'
                + placemsgOut(obsplace):0.'readmsgOut(obsread):0.DataBus'
                + placefinished(obsplace):0.'readfinished(obsread):0.DataBus'
                + placedata(obsplace):0.'readdata(obsread):0.DataBus'
                + placecmd(obsplace):0.'readcmd(obsread):0.DataBus'
                + placestatus(obsplace):0.'readstatus(obsread):0.DataBus'
                + sentdisconnect(obssentdiscon):0.
                  'readdisconnect(obsreaddiscon):0.DataBus'
                + sentcomplete(obssentcompl):0.
                  'readcomplete(obsreadcompl):0.DataBus'
                + writetarget0(obswritet0):0.'readtarget0(obsreadt0):0.DataBus'
                + writetarget1(obswritet1):0.'readtarget1(obsreadt1):0.DataBus'
```

where `Restriction` contains all actions that are internal to the protocol, i.e. those concerned with setting/releasing signals, requesting signal status, and placing/reading messages, commands, and data on/from the data bus.

4.1 Modeling the bus signals and the data bus

Conceptually, each bus signal is modeled as a Boolean variable which is either true (signal on) or false (signal off). Thus, the processes representing the signals `BSY` (*busy*), `SEL` (*select*), `C/D` (*command/data*), `I/O` (*input/output*), `MSG` (*message*), `ATN` (*attention*), `REQ` (*request*), and `ACK` (*acknowledgment*) are generically created by relabeling the actions of the process `Off` (see Table 2). Using the ports `sset` and `rel` one can set or release the signal and, hereby, switching the state to `On` or `Off`, respectively. Actions `'off` (`'on`) indicate that the signal is currently in state `Off` (`On`). Note that the atomicity of actions in process algebras guarantees that conflicts, arising by setting several signals simultaneously, are avoided.

In the following, we abstract away the contents of most messages. Only the distinguished messages `disconnect` and `complete` are explicitly considered since they require to exit the information transfer phases and to switch to the initial state of the LUN. Accordingly, we may model the data bus, as seen in Table 2, as a variable which can store and read out information (actions `placeXXX` and `readXXX`, respectively). The labels `obsXXX` are used to record the events of placing and reading messages on the bus.

Table 3 Bus Free and Command Phase

```
proc LUN0  =   t(start0):9.'relIO:0.(BusFree0 + GetSelected0)
             + t(start0):9.'setIO(obs_setIO):0.(BusFree0 + GetSelected0)
             + t:9.LUN0
             + GetSelected0

proc BusFree0 =   t(busfree):80.'setBSY(obs_setBSY):80.'setid0:0.Arbitrate0
                + isSEL(obs_isSEL):0.LUN0
                + isBSY(obs_isBSY):0.LUN0

proc CommandI0  = isREQ:0.( 'placecmd:0.'setACK:9.noREQ:0.'release:0.
                            'relACK:0.CommandI0
                          + 'placefinished:0.'setACK:9.noREQ:0.'release:0.
                            'relACK:0.Initiator0'
                          )
proc CommandT0  = 'relMSG:0.'setCD:0.'relIO(begin_Command):0.t(begin_Phase):0.
                  CommandT0'
proc CommandT0' = 'setREQ:0.isACK(obs_isACK):0.
                  ( readcmd:0.'relREQ(obs_relREQ):0.noACK:0.CommandT0'
                  + readfinished:0.'relREQ(obs_relREQ):0.noACK:0.
                    t(end_Phase):0.
                    (MsgOutT0 + MsgInT0 + DataOutT0 + DataInT0 + StatusT0)
                  )
```

4.2 Modeling the bus-phases

Now we focus on modeling the logical characteristics of the SCSI-2 bus-protocol (see Section 6 of ANSI 1994). Due to space constraints we only provide models of the Bus Free Phase and the Command Phase for LUN0 here. For the complete model we refer the reader to a technical report (Bhat et al. 1997).

In the Bus Free Phase, no device is in possession of the bus, hence it is available for arbitration. The SCSI bus is defined to be in the Bus Free Phase as soon as the signals SEL and BSY have been false for at least a bus settle delay. Accordingly, the process BusFree0 detects the Bus Free Phase when the actions isBSY and isSEL are absent for 80 time units (cf. Table 3). If one of the actions isBSY or isSEL is observed, the bus is occupied and LUN0 returns to the start state. If the bus is free, the logical unit asserts the BSY signal (action 'setBSY) and sets the arbitration variable accordingly (action 'setid0) before it performs an arbitration delay and switches to the Arbitration Phase.

The processes Target0 and Initiator0 initiate the Information Transfer Phases (ITP) which include the Command, Data, Status, and Message Phases. In those phases, information is exchanged between the initiator and the target. The Data and the Message Phases are further divided in DataIn, DataOut, MessageIn, and MessageOut Phases according to the direction of information flow. The 'In' phases are concerned with transferring information from the target to the initiator whereas the 'Out' phases are concerned with transferring information in the other direction. The information transfer takes place by byte-wise *handshakes*. The phase of the SCSI-2 bus-protocol, in which it is currently in, is encoded via the `MSG`, `C/D`, and `I/O` signals. In the following, we explain the `Command` Phase and its modeling in detail, especially the underlying handshake mechanism (cf. Table 3).

The `Command` Phase is entered if the target, the master of the bus-protocol, intends to request a command from the initiator. The target indicates the `Command` Phase by deasserting the `MSG` and `I/O` signals and asserting the `C/D` signal. After waiting for a deskew delay the target requests a command from the initiator by setting the `REQ` signal (action `'setREQ`). In the meantime, the initiator detects that the target has switched to the `Command` Phase by observing the status of the `MSG`, `C/D`, and `I/O` signals. Upon detection of the asserted `REQ` signal (action `isREQ`) the initiator places the first byte of the command on the data bus (action `'placecmd`), waits for a deskew delay, and asserts the `ACK` signal (action `'setACK`). After the target detects the asserted `ACK` signal (action `isACK`) it reads the command from the data bus (action `readcmd`) and releases the `REQ` signal (action `'relREQ`). At this point the handshake procedure for receiving (the first byte of) the command is completed. Now, the initiator may release the data bus (action `'release`) and the `ACK` signal (action `'relACK`). Alternatively, since a command may consist of more than one byte, the bus may remain in the `Command` Phase, and the handshake mechanism may be repeated, until the message *finished* (action `readfinished`) has been transferred. Note that in practice the length of a command can always be determined from its first byte.

4.3 State spaces of our model

We have created front-ends for both process algebras, $\mathtt{CCS}^{\mathtt{rt}}$ and $\mathtt{CCS}^{\mathtt{dp}}$, for the *Concurrency Workbench of North Carolina* (Cleaveland & Sims 1996), `CWB-NC`, by using the *Process Algebra Compiler* (Cleaveland et al. 1995) which is a generic tool for integrating new interfaces in the `CWB-NC`. Whereas the integration of $\mathtt{CCS}^{\mathtt{rt}}$ has been straightforward, we have needed some more effort regarding $\mathtt{CCS}^{\mathtt{dp}}$. The reason is that Rule `Act1` gives rise to an infinite branching transition system. However, for practical purposes infinite branching can be eliminated by providing an upper bound `upper` which reflects the maximal priority value of any initial action of the considered process. The validity of this approach stems from the fact that a delay of more than `upper` time units does not change the system state but results in global idling.

We have run the `CWB-NC` on a SUN SPARC 20 workstation to construct and minimize the state spaces of our models. Whereas the $\mathtt{CCS}^{\mathtt{rt}}$ version of our model has 62 400 states and 65 624 transitions, the $\mathtt{CCS}^{\mathtt{dp}}$ possesses only 8 391 states and 14 356 transitions. This drastic saving in state space emphasizes the utility of using dynamic priorities in order to encode discrete quantitative real-time.

5 VERIFYING THE SCSI-2 BUS-PROTOCOL

In this section we specify and verify several safety and liveness properties which our $\mathtt{CCS}^{\mathtt{dp}}$ model of the SCSI-2 bus-protocol is expected to satisfy. The one-to-one correspondence between $\mathtt{CCS}^{\mathtt{rt}}$ and $\mathtt{CCS}^{\mathtt{dp}}$ semantics ensures that the properties hold with respect to $\mathtt{CCS}^{\mathtt{rt}}$ semantics, too. As specification language for the properties we use the *modal μ-calculus* (Kozen 1983), and for verification we employ the *model-checker* (Bhat & Cleaveland 1996) integrated in the `CWB-NC`. The following desired requirements of the SCSI-2 bus-protocol have been extracted from the official standard (ANSI 1994).

- *Property 1:* All bus phases are always reachable. This implies that the model is free of deadlocks.
- *Property 2:* Whenever a bus phase is entered, it is eventually exited.
- *Property 3:* The signals `REQ` and `ACK` do not change between two information transfer phases.
- *Property 4:* The signal `BSY` is on and the signal `SEL` off during information transfer phases.
- *Property 5:* Whenever a device sends a message on the bus, the message is eventually received by the intended LUN.
- *Property 6:* Whenever the initiator sets the `ATN` signal, eventually the bus enters the `MessageOut` Phase.

We formalize these properties within the modal μ-calculus which is a simple but expressive language for specifying temporal properties. Its syntax and semantics has been given e.g. by Kozen (1983). For our purposes it is sufficient to introduce the intuitive meaning of the following meta-formulas, where $\alpha, \beta \in \mathcal{A}$, $L \subseteq \mathcal{A}$, and Φ is a temporal formula.

$$\mathtt{between}(\alpha,\beta,\Phi) =_{\mathrm{df}} \nu X.[\alpha](\nu Y.(\Phi \wedge [\beta]X \wedge [-\beta]Y)) \wedge [-\alpha]X$$

$$\begin{aligned}&\mathtt{fair\text{-}follows}(\alpha,\beta,L,\Phi) =_{\mathrm{df}}\\ &\quad \nu X.[\alpha](\nu Y.\mu Z.(\Phi \wedge [\beta]X \wedge [L]Y \wedge [-\beta, L]Z)) \wedge [-\alpha]X\end{aligned}$$

The meta-formula $\mathtt{between}(\alpha,\beta,\Phi)$ can be interpreted as follows: On every path it is always the case that after α, the formula Φ is true at every state until the action β is seen. Note that action β need not occur after α since β only *releases* the requirement that Φ be true at every state. The meta-formula $\mathtt{fair\text{-}follows}(\alpha,\beta,L,\Phi)$ encodes that on every path it is always the case that after action α is seen, either Φ is always true until β is seen or Φ is always true, and an action from L occurs infinitely often on the path. Note that on fair-paths, i.e. paths on which actions from L do not occur infinitely often, action β has to occur eventually. Without this notion of fairness, which we use to encode e.g. that messages transferred over the SCSI bus have finite length, some properties cannot be validated.

Unfortunately, our process algebra $\mathtt{CCS}^{\mathtt{dp}}$ turns any visible action a and $\overline{a}$ into the internal action τ when communicating on channel a. However, in order to prove any interesting property except deadlock, we have to observe certain actions of the system, e.g. asserting and deasserting bus signals. Therefore, we attach to each output action $\overline{a}$ a visible action or *probe* o, thus leading to a complex action $\overline{a}(o)$. Whenever a transition labeled by $\overline{a}(o)$ synchronizes with a transition labeled by a, the resulting τ is annotated by o, i.e. $\tau(o)$ is produced. Hence, a communication on port a is immediately observed by probe o, as intended. Our model includes the probes `begin_Phase` and `end_Phase` marking the beginning and end of each information transfer phase, respectively, and the probes `obs_setSIG` and `obs_relSIG` indicating the assertion and deassertion of some signal `SIG`, respectively.

Now, we can formalize the desired properties in the modal μ-calculus as shown for Properties 2 and 3. For Property 2 we have to check for every path that probe `begin_Phase` is eventually followed by probe `end_Phase` before another `begin_Phase` is observed.

$$\mathtt{fair\text{-}follows}(\mathtt{begin\_Phase}, \mathtt{end\_Phase}, \{\mathtt{obs\_setATN}\}, \langle - \rangle tt) \ .$$

The implicit fairness constraint ensures that the initiator does not forever ignore the target's wish to enter a new phase by continuously asserting the `ATN` signal. Regarding Property 3 we encode that on all paths the probes `obs_setREQ`, `obs_relREQ`, `obs_setACK`, and `obs_relACK` do not occur in between `end_Phase` and `begin_Phase` by the formula

$$\begin{array}{ll} \mathtt{between}(& \mathtt{end\_Phase}, \mathtt{begin\_Phase}, \\ & [\mathtt{obs\_setREQ}, \mathtt{obs\_relREQ}, \mathtt{obs\_setACK}, \mathtt{obs\_relACK}]\mathit{ff} \quad) \ . \end{array}$$

We were able to validate each property in our model within at most two minutes when running the `CWB-NC` on a SUN SPARC 20 workstation. The model checker we used is a local model checker (Bhat & Cleaveland 1996). Applying a *local* model checker in contrast to a *global* one remarkably speeds-up the task of verification. In fact, the modeling of the SCSI-2 bus-protocol has been done in several stages, after each of which the above mentioned properties have been checked. At early modeling stages the model checker has invalidated most properties immediately. The encountered errors have ranged from missed fairness constraints to wrong timing information. However, the diagnostic information in form of failure traces provided by the model checker simplifies the task of finding bugs in models.

During the process of verification, we also realized that the timing constraints of the bus-protocol are not only imposed for avoiding wire glitches but also in order to implement necessary synchronization constraints during the initial bus-phases. Without these synchronization constraints, two LUNs may gain access to the bus for arbitration which leads to a deadlock. This emphasizes the necessity of dealing with real-time constraints in reactive systems.

6 CONCLUSIONS AND FUTURE WORK

We introduced the process algebra $\mathtt{CCS}^{\mathtt{dp}}$ with dynamic priorities whose semantics corresponds one-to-one with the discrete quantitative real-time semantics of $\mathtt{CCS}^{\mathtt{rt}}$. However, the $\mathtt{CCS}^{\mathtt{dp}}$ semantics yields significantly more compact models and, thus, provides a means for efficiently implementing traditional real-time process algebras. Moreover, our approach does not abstract away any aspects of real-time, i.e. all quantitative timing constraints can still be verified within $\mathtt{CCS}^{\mathtt{dp}}$ semantics.

We implemented the process algebras $\mathtt{CCS}^{\mathtt{dp}}$ and $\mathtt{CCS}^{\mathtt{rt}}$ in the Concurrency Workbench of North Carolina, an automated verification tool, which we used to formally model and reason about the SCSI-2 bus-protocol. The size of our model is about an order of magnitude smaller when constructed with $\mathtt{CCS}^{\mathtt{dp}}$ instead of $\mathtt{CCS}^{\mathtt{rt}}$ semantics and could be handled easily within the Workbench. In addition, we specified several desired properties of the bus-protocol in the modal μ-calculus and validated them by using model checking. Regarding future work, the SCSI-2 bus-protocol should be modeled in more detail, thereby enabling the verification of additional interesting properties.

Acknowledgments. We would like to thank the anonymous referees, Michael Mendler, and Pranav K. Tiwari for their comments and suggestions. Research support for the first two authors has been provided by NSF/DARPA grant CCR-9014775, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, NSF Young Investigator Award CCR-9257963, NSF grant CCR-9402807, and AFOSR grant F49620-95-1-0508. The research of the third author was partly supported by the German Academic Exchange Service under grant D/95/09026 (Doktorandenstipendium HSP II / AUFE).

REFERENCES

Albert, J. L., Monien, B. & Artalejo, M. R., eds (1991), *Automata, Languages and Programming (ICALP '91)*, Vol. 510 of *Lecture Notes in Computer Science*, Springer-Verlag, Madrid.

ANSI (1994), *ANSI X3.131–1994, Information Systems — Small Computer Systems Interface–2*, ANSI. See also http://abekas.com:8080/SCSI2/.

Baeten, J., ed. (1990), *Applications of Process Algebra*, Vol. 17 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, Cambridge, England.

Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P. & Yi, W. (1995), UPAAL — a tool suite for automatic verification of real-time systems, *in* 'Proccedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems', Lecture Notes in Computer Science, Springer-Verlag.

Bhat, G. & Cleaveland, R. (1996), Efficient local model-checking for fragments of the modal μ-calculus, *in* T. Margaria & B. Steffen, eds, 'Second

International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)', Vol. 1055 of *Lecture Notes in Computer Science*, Springer-Verlag, Passau, Germany, pp. 107–126.

Bhat, G., Cleaveland, R. & Lüttgen, G. (1997), Dynamic priorities for modeling real-time, Technical report, North Carolina State University, Raleigh, NC, USA. To appear.

Bolognesi, T. & Brinksma, E. (1987), 'Introduction to the ISO specification language LOTOS', *Computer Networks and ISDN Systems* **14**, 25–59.

Clarke, E., Emerson, E. & Sistla, A. (1986), 'Automatic verification of finite-state concurrent systems using temporal logic specifications', *ACM Transactions on Programming Languages and Systems* **8**(2), 244–263.

Cleaveland, R. & Hennessy, M. (1990), 'Priorities in process algebra', *Information and Computation* **87**(1/2), 58–77.

Cleaveland, R., Madelaine, E. & Sims, S. (1995), Generating front-ends for verification tools, *in* E. Brinksma, W. R. Cleaveland, K. G. Larsen, T. Margaria & B. Steffen, eds, 'First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95)', Vol. 1019 of *Lecture Notes in Computer Science*, Springer-Verlag, Aarhus, Denmark, pp. 153–173.

Cleaveland, R., Natarajan, V., Sims, S. & Lüttgen, G. (1996), 'Modeling and verifying distributed systems using priorities: A case study', *Software–Concepts and Tools* **17**(2), 50–62.

Cleaveland, R. & Sims, S. (1996), The NCSU Concurrency Workbench, *in* R. Alur & T. Henzinger, eds, 'Computer Aided Verification (CAV '96)', Vol. 1102 of *Lecture Notes in Computer Science*, Springer-Verlag, New Brunswick, New Jersey, pp. 394–397.

Elseaidy, W., Baugh, J. & Cleaveland, R. (1996), 'Verification of an active control system using temporal process algebra', *Engineering with Computers* **12**, 46–61.

Holzmann, G. (1991), *Design and Validation of Computer Protocols*, Prentice-Hall.

Jeffrey, A. (1992), Translating timed process algebra into prioritized process algebra, *in* J. Vytopil, ed., 'Proceedings of Symposium on Real-Time and Fault-Tolerant Systems (FTRTFT'92)', Vol. 571 of *Lecture Notes in Computer Science*, Springer-Verlag, Nijmegen, The Netherlands, pp. 493–506.

Kozen, D. (1983), 'Results on the propositional μ-calculus', *Theoretical Computer Science* **27**, 333–354.

Milner, R. (1989), *Communication and Concurrency*, Prentice-Hall, London.

Moller, F. & Tofts, C. (1990), A temporal calculus of communicating systems, *in* J. Baeten & J. Klop, eds, 'CONCUR '90', Vol. 458 of *Lecture Notes in Computer Science*, Springer-Verlag, Amsterdam, pp. 401–415.

Yi, W. (1991), CCS + time = an interleaving model for real time systems, *in* Albert et al. (1991), pp. 217–228.

APPENDIX 1 AUXILIARY DEFINITIONS

Tables 4 and 5 formally present auxiliary relations used for defining the operational semantics of $\texttt{CCS}^{\texttt{dp}}$.

Table 4 Initial action sets

$\mathcal{I}^k(nil) =_{\mathrm{df}} \emptyset$	$\mathcal{I}^k(\alpha\!:\!l.P) =_{\mathrm{df}} \{\alpha \mid l < k\}$
$\mathcal{I}^k(P+Q) =_{\mathrm{df}} \mathcal{I}^k(P) \cup \mathcal{I}^k(Q)$	$\mathcal{I}^k(P \rhd Q) =_{\mathrm{df}} \mathcal{I}^k(P) \cup \mathcal{I}^k(Q)$
$\mathcal{I}^k(P\|Q) =_{\mathrm{df}} \mathcal{I}^k(P) \cup \mathcal{I}^k(Q) \cup \{\tau \mid \mathcal{I}^k(P) \cap \overline{\mathcal{I}^k(Q)} \neq \emptyset\}$	
$\mathcal{I}^k(P[f]) =_{\mathrm{df}} \{f(\alpha) \mid \alpha \in \mathcal{I}^k(P)\}$	$\mathcal{I}^k(P \backslash L) =_{\mathrm{df}} \{\alpha \notin L \cup \overline{L} \mid \alpha \in \mathcal{I}^k(P)\}$
$\mathcal{I}^k(C) =_{\mathrm{df}} \mathcal{I}^k(P)$ where $C \stackrel{\mathrm{def}}{=} P$	

Table 5 Priority adjustment function

$[nil]^k =_{\mathrm{df}} nil$	$[\alpha\!:\!l.P]^k =_{\mathrm{df}} \begin{cases} \alpha\!:\!(l-k).P & \text{if } l > k \\ \alpha\!:\!0.P & \text{otherwise} \end{cases}$
$[P+Q]^k =_{\mathrm{df}} [P]^k + [Q]^k$	$[P \rhd Q]^k =_{\mathrm{df}} [P]^k \rhd [Q]^k$
$[P\|Q]^k =_{\mathrm{df}} [P]^k \mid [Q]^k$	$[C]^k =_{\mathrm{df}} [P]^k$ where $C \stackrel{\mathrm{def}}{=} P$
$[P[f]]^k =_{\mathrm{df}} [P]^k[f]$	$[P \backslash L]^k =_{\mathrm{df}} [P]^k \backslash L$