Priorities for Modeling and Verifying Distributed Systems

Rance Cleaveland*
Gerald Lüttgen*
V. Natarajan*
Steve Sims*

ABSTRACT This paper illustrates the use of priorities in process algebras by a real-world example dealing with the design of a safety-critical network which is part of a railway signaling system. Priorities in process algebras support an intuitive modeling of distributed systems since undesired interleavings can be suppressed. This fact also leads to a substantial reduction of the sizes of models. We have implemented a CCS-based process algebra with priorities as a new front-end for the NCSU Concurrency Workbench, and we use model checking for verifying properties of the signaling system.

1 Introduction

Process algebras, e.g. CCS [13], provide a formal framework for modeling and verifying distributed systems. In the past decade, a number of automatic verification tools for finite state systems expressed in process algebras have been developed [10], and their utility has been demonstrated by several case studies [1, 7]. Most of these case studies are based on process algebras that provide simple mechanisms for modeling nondeterminism and concurrency. Many extensions to these *plain* languages have been proposed, including priorities [2, 6, 8, 14]. Priorities in particular are needed to model the often used concept of *interrupts*, especially in hardware and communication protocols.

This paper presents a case study of a real-world system which shows the benefits of priorities for modeling and verifying distributed systems. Our example is based on a case study by Glenn Bruns [5] dealing with

---

*Department of Computer Science, North Carolina State University, Raleigh NC 27695-8206 USA, e-mail: {rance, luettgen, nvaidhy, stsims}@eos.ncsu.edu.
Research supported 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.
Research support for the second author provided by the German Academic Exchange Service under grant D/95/09026 (Doktorandenstipendium HSP II / AUFE).

the design of a safety-critical part of a network used in British Rail's Solid State Interlocking (SSI) [11], a system which controls railway signals and points. Bruns modeled and verified a high-level design of the system that abstracted from low-level implementation details. He used plain CCS for modeling the system, a temporal logic for specifying properties of the system, and the Edinburgh Concurrency Workbench [10] for verifying that these properties hold for the model.

We investigate an elaboration of Bruns' case study using a process algebra with priorities [8]. We augment his model in two ways based on key concepts of the SSI system described in the original design document [11]. First we add an error-recovery scheme that is invoked when a communication link fails, and second we add a backup line in order to make the system fault-tolerant. In both cases the use of priorities enables the development of elegant and intuitive models. We also show that, by eliminating invalid interleavings, priorities can dramatically cut the number of states and transitions in our systems. This is particularly significant since the large complexity of practical problems often prevents their automatic verification. We verify our models by showing that several safety properties hold using the NCSU Concurrency Workbench. This verification tool is a re-implementation of the Edinburgh Concurrency Workbench that offers similar functionality, but is faster, able to handle larger systems, and gives diagnostic information when a verification routine returns false.

The remainder of the paper is structured as follows. In Section 2 we present the process algebra with priorities that we use in this paper. Section 3 gives an introduction to the railway signaling system and presents our models. Section 4 discusses our verification results. Finally, we give our conclusions and directions for future work in Section 5.

2 A Process Algebra with Priorities

The process algebra with priorities we consider in this paper is based on the language proposed in [8]. We extend this language with a multilevel priority scheme but disallow the prioritization and deprioritization operator. Therefore, our process algebra is basically CCS [13] where priorities, modeled by natural numbers, are assigned to actions. We use the convention that smaller numbers mean higher priorities; so 0 is the highest priority. 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.

Formally, let $\{ \Lambda_k \mid k \in \mathbb{N} \}$ denote a family of pairwise-disjoint, countably infinite sets of *labels*. Intuitively, Λ_k contains the "ports" with priority k that processes may synchronize over. Then the set of *actions* $\mathcal{A}_k$ with priority k may be defined by $\mathcal{A}_k =_{\mathrm{df}} \Lambda_k \cup \overline{\Lambda}_k \cup \{\tau_k\}$, where $\overline{\Lambda}_k =_{\mathrm{df}} \{ \overline{\lambda} \mid \lambda \in \Lambda_k \}$ and $\tau_k \notin \Lambda_k$. The set of all ports Λ and the set of all actions $\mathcal{A}$ are defined by $\bigcup\{ \Lambda_k \mid k \in \mathbb{N} \}$ and $\bigcup\{ \mathcal{A}_k \mid k \in \mathbb{N} \}$, respectively. For better readability we write $a{:}k$ if $a \in \Lambda_k$ and $\tau{:}k$ for τ_k. An action $\lambda{:}k \in \Lambda_k$ may be thought of as representing the receipt of an input on port λ which has priority k, while $\overline{\lambda}{:}k \in \overline{\Lambda}_k$ constitutes the deposit of an output on λ. The invisible actions $\tau{:}k$ represent internal computation steps with priority k. In what follows, we use $\alpha{:}k, \beta{:}k, \ldots$ to range over $\mathcal{A}$ and $a{:}k, b{:}k, \ldots$ to range over Λ. We also use $\lambda{:}k$ to represent elements in $\mathcal{A}_k \setminus \{\tau{:}k\}$ and extend $\bar{}$ to all visible actions $\lambda{:}k$ by $\overline{\overline{\lambda}}{:}k =_{\mathrm{df}} \lambda{:}k$. Finally, if $L \subseteq \mathcal{A} \setminus \{ \tau{:}k \mid k \in \mathbb{N} \}$ then $\overline{L} = \{ \overline{\lambda}{:}k \mid \lambda{:}k \in L \}$. For the sake of simplicity, we also write $\tau \in M$ where $M \subseteq \mathcal{A}$ if $\tau{:}k \in M$ for some $k \in \mathbb{N}$.

The *syntax* of our language is defined by the following BNF.

$$
\begin{array}{lllllllll}
P & ::= & \mathit{nil} & \mid & \alpha{:}k.P & \mid & P + P & \mid & P|P \\
 & & P[f] & \mid & P \backslash L & \mid & C \stackrel{\mathrm{def}}{=} P & \mid & P \mathrel{[\!\!>} P
\end{array}
$$

Here f is a *relabeling*, a mapping on $\mathcal{A}$ which satisfies $f(\tau{:}k) = \tau{:}k$ for all $k \in \mathbb{N}$ and $\overline{f(\overline{a}{:}k)} = f(a{:}k)$ for all $a{:}k \in \mathcal{A} \setminus \{ \tau{:}k \mid k \in \mathbb{N} \}$. Moreover, a relabeling preserves priority values, i.e. for all $a{:}k \in \mathcal{A} \setminus \{ \tau{:}k \mid k \in \mathbb{N} \}$ we have $f(a{:}k) = b{:}k$ for some $b{:}k \in \Lambda_k$. Further, $L \subseteq \mathcal{A} \setminus \{ \tau{:}k \mid k \in \mathbb{N} \}$, and C is a constant whose meaning is given by a defining equation. Additionally, we include the *disabling* operator $[\!\!>$ which is closely related to the corresponding operator in LOTOS [4].

We adopt the usual definitions for *closed* terms and *guarded recursion*. We call the closed guarded terms *processes*. $\mathcal{P}$ represents the set of all processes and is ranged over by $P, Q, \ldots$. Note that our framework allows an infinite number of priority levels although there is a maximum priority.

The *semantics* of our language is given by a labeled transition system $\langle \mathcal{P}, \mathcal{A}, \longrightarrow \rangle$ where $\mathcal{P}$ is the set of processes, and $\longrightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ defined in Table 1 is the transition relation. We will write $P \xrightarrow{\alpha:k} P'$ instead of $\langle P, \alpha{:}k, P' \rangle \in \longrightarrow$, and we say that *$P$ may engage in action α with priority k and thereafter behaves like process P'*.

The presentation of the operational rules requires *initial action sets* which are inductively defined on the syntax of processes, as usual. Intuitively, $\mathrm{I}_k(P)$ denotes the set of all initial actions of P with priority k, $\mathrm{I}_{<k}(P)$ the set of all initial actions of P with a higher priority than k, and $\mathrm{I}(P)$ the set of all initial actions of P. Moreover, we define $\mathrm{I}_{<0}(P) =_{\mathrm{df}} \emptyset$. Note that the initial action sets are independently defined from the transition relation $\longrightarrow$; so the transition relation is well-defined.

Intuitively, $\alpha{:}k.P$ may engage in action α with priority k. The *summation operator* $+$ denotes *nondeterministic choice*. The process $P{+}Q$ may behave like process P (Q) if Q (P) does not preempt it by performing a higher prioritized τ-action. The *restriction operator* $\backslash L$ prohibits the execution of actions in $L \cup \overline{L}$ and may be seen as permitting the *scoping* of actions. $P[f]$

TABLE 1. Operational semantics

			$\alpha{:}k.P \xrightarrow{\alpha:k} P$
$P \xrightarrow{\alpha:k} P'$,	$\tau \notin \mathrm{I}_{<k}(Q)$	implies	$P + Q \xrightarrow{\alpha:k} P'$
$Q \xrightarrow{\alpha:k} Q'$,	$\tau \notin \mathrm{I}_{<k}(P)$	implies	$P + Q \xrightarrow{\alpha:k} Q'$
$P \xrightarrow{\alpha:k} P'$,	$\tau \notin \mathrm{I}_{<k}(P\vert Q)$	implies	$P\vert Q \xrightarrow{\alpha:k} P'\vert Q$
$Q \xrightarrow{\alpha:k} Q'$,	$\tau \notin \mathrm{I}_{<k}(P\vert Q)$	implies	$P\vert Q \xrightarrow{\alpha:k} P\vert Q'$
$P \xrightarrow{a:k} P'$, $Q \xrightarrow{\overline{a}:k} Q'$,	$\tau \notin \mathrm{I}_{<k}(P\vert Q)$	implies	$P\vert Q \xrightarrow{\tau:k} P'\vert Q'$
$P \xrightarrow{\alpha:k} P'$,	$f(\alpha{:}k) = \beta{:}k$	implies	$P[f] \xrightarrow{\beta:k} P'[f]$
$P \xrightarrow{\alpha:k} P'$,	$\alpha{:}k \notin (L \cup \overline{L})$	implies	$P \backslash L \xrightarrow{\alpha:k} P' \backslash L$
$P \xrightarrow{\alpha:k} P'$,	$\tau \notin \mathrm{I}_{<k}(Q)$	implies	$P \,\rangle\!\rangle Q \xrightarrow{\alpha:k} P' \,\rangle\!\rangle Q$
$Q \xrightarrow{\alpha:k} Q'$,	$\tau \notin \mathrm{I}_{<k}(P)$	implies	$P \,\rangle\!\rangle Q \xrightarrow{\alpha:k} Q'$
$P \xrightarrow{\alpha:k} P'$,	$C \stackrel{\text{def}}{=} P$	implies	$C \xrightarrow{\alpha:k} P'$

behaves exactly as process P where the actions are renamed according to 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 on some priority level k resulting in the internal action $\tau{:}k$. However, if Q (P) is capable of engaging in a higher prioritized internal action or in a synchronization, then lower prioritized actions of P (Q) are preempted. The process $P \,\rangle\!\rangle Q$ behaves like P and, additionally, it is capable of *disabling* P by engaging in Q. The side conditions ensure that its semantics is consistent with that of the summation operator. In practice, Q is often an *interrupt handler*. Finally, $C \stackrel{\text{def}}{=} P$ denotes a *constant definition*, i.e. C is a recursively defined process that is a distinguished solution to the equation $C = P$. The side conditions of the operational semantic rules guarantee that high-priority τ-actions have preemptive power over low-priority actions. The reason that high-priority visible actions do *not* have priority 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.

The usual definition of strong bisimulation – as introduced in [13] – is a congruence relation over $\mathcal{P}$ [8]. In the context of our process algebra with priorities we will refer to it as *prioritized strong bisimulation*.

In the following case study, it is sometimes useful to have visible actions that have preemptive power over lower prioritized actions. More precisely, such an action, e.g. $a{:}k$, is signaling that certain events have occurred, i.e. it plays the role of an *atomic proposition*. We give $a{:}k$ preemptive power by inserting a $\tau{:}k$-loop at the origin states of transitions which are labeled by $a{:}k$. For example the process $a{:}0.P$ is rewritten to $C \stackrel{\text{def}}{=} a{:}0.P + \tau{:}0.C$. For the sake of simplicity, we write $\#a{:}0.P$ as a shorthand for C.

3 Modeling a Railway System

In this section we model a network used in a *safety-critical railway signaling system*. The basic design is adapted from [5]. However, instead of CCS we use the process algebra of Section 2 as it allows a more intuitive modeling of the system. Further, we extend the model by an *error-recovery scheme* and a *fault-tolerant network link* in order to reflect the underlying design document [11] more precisely. Since in both cases *interrupt mechanisms* come into play, the use of a process algebra with priorities is needed for reflecting the design correctly.

3.1 Solid State Interlocking

Our example is embedded in *British Rail's Solid State Interlocking* (SSI) system [5, 11], which adjusts and controls *signals* and *points* along rail routes. Its aim is to prevent situations which may lead to a collision or derailment of trains. Therefore, a formal verification of the design of the SSI and its environment is of particular importance.

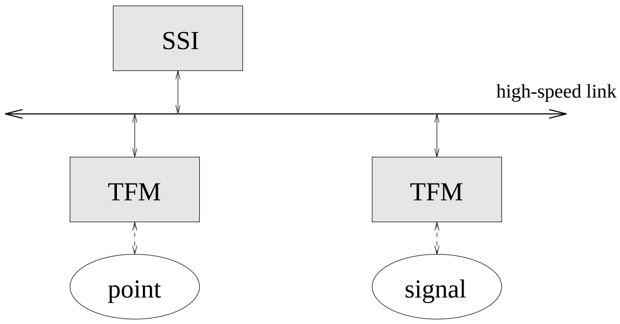

FIGURE 1. The SSI environment – overview

Figure 1 shows the basic design of the interlocking system. It consists of three different components: the SSI, several *trackside functional modules* (TFM), and a *high-speed link* which connects the TFMs with the SSI. The SSI is the main logical unit of the system. It is connected to a control panel to which a signal operator can input her/his commands. The SSI checks the validity of those commands and sends them to the TFMs along the track via the high-speed link. A TFM connects a signal or a point to the network. Its task is to listen to the network in order to receive messages for adjusting its signal or point and to send status information about the signal or point to the SSI.

The pattern of communication between the SSI and the TFMs is as follows. The SSI sends cyclically a message to each TFM. The message includes the TFM's address and the status for the corresponding signal or point (e.g. signal on/off). After sending a message the SSI waits a short time for the addressed TFM to respond with the current state of its signal or point. This *polling* scheme reflects the safety-critical design of the system since it leads to a quick detection of failures. For example, if the addressed TFM does not respond then either the TFM or the connection between the SSI and the TFM is broken. Moreover, if the corresponding signal or point has autonomously changed its state, it is forced to return to its proper state. The disadvantage of the polling scheme is its communication overhead, which necessitates an expensive high-speed network. This expense is even worse if the distance between some TFMs and the SSI is very large. Therefore, the question arises as to whether distant high-speed links can be connected via a low-grade link without violating safety requirements. Our case study will concentrate on this aspect of the SSI system since the use of a high-speed link is known to satisfy the requirements on the error-free delivery of commands and timely detection of failures [5].

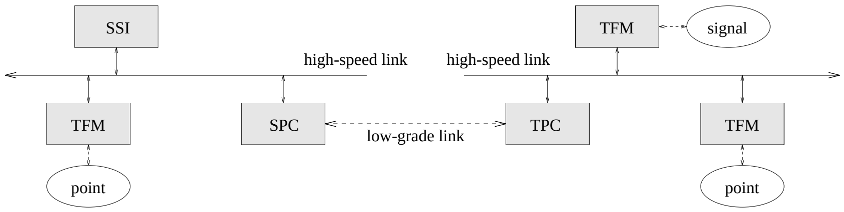

FIGURE 2. The slow-scan system – overview

The integration of a low-grade link (LGL) is illustrated in Figure 2. It is connected to the SSI-side high-speed link via a *SSI-side protocol converter* (SPC) and to the TFM-side high-speed link via a *TFM-side protocol converter* (TPC). Intuitively, the SPC is expected to behave like the TFMs on the other side of the LGL, i.e. to accept commands for those TFMs and to respond to the SSI with their current states, but the SPC occasionally sends these commands along the LGL and receives new status information about those TFMs. On the other side, the TPC should mimic a SSI. We refer to the part of the system which consists of SPC, LGL, and TPC as the *slow-scan system*. In order not to violate safety conditions of the overall system, the slow-scan system is expected to satisfy properties of the following kind. If the low-grade link fails, then the TPC (SPC) will detect the problem and stop sending messages to the TPC-side TFMs (SSI). The TFMs are also expected to change signals to red and to lock points in their current setting if they stop receiving messages.

3.2 The Slow-scan Model

In the following, we formally model the slow-scan system in three steps. First, we present the system as in [5] and discuss the advantages of priorities for modeling. In the second step, we augment our model with an *error-recovery scheme* and remodel the low-grade link in a *full-duplex* fashion. Finally, we show how the required fault-tolerance of the system [11] can be reflected in our design.

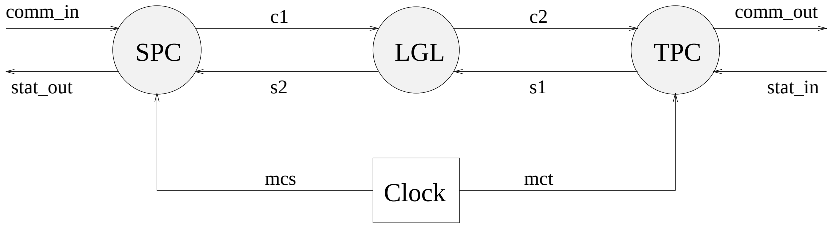

FIGURE 3. The LGL model

Tables 2 and 3 contain the model of the slow-scan system as it is accepted by the NCSU Concurrency Workbench where the symbol $*$ introduces comments. The front end of the workbench for the process algebra with priorities was generated by the process algebra compiler PAC [9] and uses the following syntactical notations for expressions: `bi C P` for the process algebra term $C \overset{\text{def}}{=} P$, and `'a:k` for the action $\overline{a}{:}k$.

Figure 3 shows the channels between the three parallel components of the slow-scan system; it also includes an additional clock. Since the correct behavior of our system depends on time constraints, which cannot be modeled in our process algebra directly (cf. [15]), we use the clock in our model to signal the progression of time to the SPC and TPC via the channels `mcs` and `mct`, respectively.

The low-grade link is modeled by two parallel unidirectional links. Since we are concerned with the design of a system we choose a poor capacity (or bandwidth) link, capacity one for each direction, and we abstract from message headers and contents. Moreover, the SPC and TPC should be able to deliver a message to and get a message from the medium at any time. If no capacity in a link is left, a new message overwrites a message which is already in the medium, and an overfull error occurs. Therefore, a link behaves for each direction as an input-enabled one-place buffer. Additionally, it offers the action `'outu` to its environment if the buffer is empty. With respect to its reliability, we assume that a link can fail because of a broken wire (action `'fail_wire`) or if its buffering capacity is exceeded (action `'fail_overfull`). If an error has occurred, the medium enters the error state CommF in which it only accepts messages but never delivers any messages.

TABLE 2. The slow-scan model (Part 1)

```
* level 0: fail_overfull, det, mcs, mct
* level 1: -
* level 2: out (c2, s2), fail_wire, comm_in, comm_out, stat_in, stat_out
* level 3: in (c1, s1)
* level 4: outu (c2u, s2u), tick

* Slow-scan system

bi SS     (SPC | LGL | TPC | Clock)\{c1:3,c2:2,c2u:4,s1:3,s2:2,s2u:4,
                                     mcs:0,mct:0}

* SSI-side protocol converter (SPC)

bi SPC    SPC0
bi SPC0   comm_in:2.'stat_out:2.SPC0 + 'c1:3.SPC0 + s2:2.SPC0 + s2u:4.SPC0 +
          mcs:0.'c1:3.SPC1
bi SPC1   comm_in:2.'stat_out:2.SPC1 + 'c1:3.SPC1 + s2:2.SPC0 + s2u:4.SPC1 +
          mcs:0.'c1:3.SPC2
bi SPC2   comm_in:2.'stat_out:2.SPC2 + 'c1:3.SPC2 + s2:2.SPC0 + s2u:4.SPC2 +
          mcs:0.#'det:0.SPCF
bi SPCF   comm_in:2.SPCF + s2:2.SPCF + s2u:4.SPCF + mcs:0.SPCF
```

The states of the SPC are parameterized by a time mark. In each state the SPC is able to accept a message from the SSI (action `comm_in`) and, subsequently, of responding with the appropriate status information of the requested TFM (action `'stat_out`). At least once every clock cycle (action `'mcs`) the SPC sends a message over the LGL to the TPC (action `'c1`) and increases its internal time-counter by changing its state from SPC0 to SPC1 or from SPC1 to SPC2. If the SPC receives a message (action `s2`) from the TPC within two time units, it resets its internal time-counter to 0 by changing its state to SPC0. Otherwise, the SPC times out (action `'det`) and enters the failure state SPCF. In this state the SPC never sends messages to the SSI or TPC again, but it remains input-enabled.

Up to now, we have not discussed how priorities can be used in modeling the system. However, one has probably already noticed that various parts of the model without priorities would be counterintuitive. For example, if a link has no capacity for an additional message, it should favor outputting a message over accepting a new one, as the latter immediately leads to the failure of the link. Similarly, a link should favor accepting a new message instead of signaling that the buffer is empty. If the clock gives a new time pulse by performing the action `'tick`, it should immediately inform the SPC and TPC by performing the (interrupt) actions `'mcs` and `'mct`. In other words, no action should interfere between the actions `'tick` and `'mcs` and the actions `'mcs` and `'mct`. Moreover, the actions `'det` and `'fail_overfull`

TABLE 3. The slow-scan model (Part 2)

```
* Track-side protocol converter (TPC)

bi TPC    TPC0
bi TPC0   'comm_out:2.stat_in:2.TPC0 + 's1:3.TPC0 + c2:2.TPC0 + c2u:4.TPC0 +
          mct:0.'s1:3.TPC1
bi TPC1   'comm_out:2.stat_in:2.TPC1 + 's1:3.TPC1 + c2:2.TPC0 + c2u:4.TPC1 +
          mct:0.'s1:3.TPC2
bi TPC2   'comm_out:2.stat_in:2.TPC2 + 's1:3.TPC2 + c2:2.TPC0 + c2u:4.TPC2 +
          mct:0.#'det:0.TPCF
bi TPCF   stat_in:2.TPCF + c2:2.TPCF + c2u:4.TPCF + mct:0.TPCF

* Low grade link (LGL)

bi LGL    Comm[c1:3/in:3,c2:2/out:2,c2u:4/outu:4] |
          Comm[s1:3/in:3,s2:2/out:2,s2u:4/outu:4]

bi Comm   Comm0
bi Comm0  in:3.Comm1 + 'outu:4.Comm0 + 'fail_wire:2.CommF
bi Comm1  in:3.#'fail_overfull:0.CommF + 'out:2.Comm0 + 'fail_wire:2.CommF
bi CommF  in:3.CommF + 'outu:4.CommF

* Clock

bi Clock  'tick:4.'mcs:0.'mct:0.Clock
```

are signaling failures which have already occurred, i.e. they should not be delayed. Finally, the failing of the LGL is always possible, i.e. no action of the LGL should be able to preempt the action `'fail_wire`.

Based on these observations, we give the actions `'fail_overfull` and `'det` – which can be viewed as *atomic propositions* – overall preemptive power in our model, i.e. they are translated to `#'fail_overfull:0` and to `#'det:0`, respectively. The actions `'mcs` and `'mct` are also assigned the highest priority. Thus, they cannot be prevented by any action in the system, and the atomicity of the above mentioned action sequences is guaranteed. Moreover, in the LGL, `'out` should have a higher priority than `in`, and `in` a higher one than `'outu`. The action `'tick` is assigned to the lowest priority level, reflecting our design decision to adopt the *maximal progress assumption* of real-time process algebra [15]. This assumption states that time may only proceed if the system cannot engage in a communication. Finally, `'fail_wire` is assigned the highest priority-level with respect to the actions of the LGL. These observations lead to the priority scheme of actions for the slow-scan model which is summarized in the beginning of Table 2.

3.3 The Recovery Model

The slow-scan model represents a substantial abstraction from reality since it is not capable of recovering from a failure. Therefore, we augment the slow-scan model by an error-recovery scheme. Moreover, we change the design of the medium to a more realistic *full-duplex* version.

TABLE 4. The recovery model (Part 1)

```
* Low grade link (LGL)

bi LGL     Comm00[c1:3/in:3,s1:3/in':3,c2:2/out:2,s2:2/out':2,
                  c2u:4/outu:4,s2u:4/outu':4,ok:1/online:1]

bi Comm00  in:3.Comm10 + 'outu:4.Comm00 + in':3.Comm01 + 'outu':4.Comm00 +
           'fail_wire:2.CommF + 'online:1.Comm00
bi Comm10  in:3.#'fail_overfull:0.CommF + 'out:2.Comm00 + in':3.Comm11 +
           'outu':4.Comm10 + 'fail_wire:2.CommF + 'online:1.Comm10
bi Comm01  in:3.Comm11 + 'outu:4.Comm01 + in':3.#'fail_overfull:0.CommF +
           'out':2.Comm00 + 'fail_wire:2.CommF + 'online:1.Comm01
bi Comm11  in:3.#'fail_overfull:0.CommF + 'out:2.Comm01 +
           in':3.#'fail_overfull:0.CommF + 'out':2.Comm10 +
           'fail_wire:2.CommF + 'online:1.Comm11
bi CommF   in:3.CommF + 'outu:4.CommF + in':3.CommF + 'outu':4.CommF +
           'repaired:2.Comm00
```

Full-duplex media have the property that if one direction fails then the other should also be considered as unreliable. In the remainder of this section, the action names of both directions of the link will only differ by a trailing prime. As long as the full-duplex medium which is modeled in Table 4 provides service, i.e. it is in one of the states Comm00, Comm10, Comm01, or Comm11, an `'online` (`'ok`) is signaled to the environment. In contrast to the slow-scan model, a broken medium can be repaired in the recovery model. This is modeled by the action `'repaired` which is enabled in the failure state CommF and allows the LGL to reset to its initial state Comm00. The recovery of the system as modeled in Table 5 works as follows. If the SPC (TPC) is in its failure state SPCF (TPCF) and detects that the medium has been repaired by receiving the action `ok` (`online`), it sends one interrupt (action `'reset`) to the clock and another (action `'init`) to the TPC (SPC). The invoked interrupt handler of the clock resets the clock. The handler of the TPC (SPC) agrees to that request by sending an acknowledgment (action `'ack_init`) back to the SPC (TPC), signaling its reinitialization (action `'recovered`), and resetting itself to its initial state. Since we are dealing with abstract models, we leave it open to an implementation how to send interrupt signals between SPC, TPC, and the clock; e.g. one could use the already repaired line.

TABLE 5. The recovery model (Part 2)

```
* level 0: fail_overfull, det, recovered, mcs, mct
* level 1: online (ok), init, ack_init, reset
* level 2: out (c2, s2), fail_wire, repaired,
*          comm_in, comm_out, stat_in, stat_out
* level 3: in (c1, s1)
* level 4: outu (c2u, s2u), tick

* Slow-scan system

bi SS      (SPC | LGL | TPC | Clock)\{c1:3,c2:2,c2u:4,s1:3,s2:2,s2u:4,mcs:0,
                                  mct:0,ok:1,init:1,ack_init:1,reset:1}

* SSI-side protocol converter (SPC)

bi SPC     SPC0 [> (init:1.'ack_init:1.#'recovered:0.SPC + ack_init:1.SPC)
...
bi SPCF    comm_in:2.SPCF + s2:2.SPCF + s2u:4.SPCF + mcs:0.SPCF +
           ok:1.'reset:1.'init:1.nil

* Track-side protocol converter (TPC)

bi TPC     TPC0 [> (init:1.'ack_init:1.#'recovered:0.TPC + ack_init:1.TPC)
...
bi TPCF    stat_in:2.TPCF + c2:2.TPCF + c2u:4.TPCF + mct:0.TPCF +
           ok:1.'reset:1.'init:1.nil

* Clock

bi Clock   Clock0 [> reset:1.Clock
bi Clock0  'tick:4.'mcs:0.'mct:0.Clock0
```

If a link has been repaired, the system should reset itself immediately. Therefore, all actions involving the recovery scheme are interrupt actions. However, they should not be able to interfere with the atomicity of the clock signals (actions `'mcs` and `'mct`). Therefore, the actions `ok`, `online`, `init` and `ack_init` are assigned to priority level one. The action `'repaired` should never be preempted by any communication in which the buffer is involved, so it gets the priority value two. Finally, the action `'recovered` is handled as the other ‘atomic propositions’ `'det` and `'fail_overfull`.

3.4 The Fault-tolerant Model

We now turn our attention to modeling *fault-tolerance* which is an essential requirement of the SSI [11]. We have already modeled an error-recovery scheme for the medium, which ensures fault-tolerance on a *software-level*. In practice, the *hardware* of the system is also replicated in order to guarantee

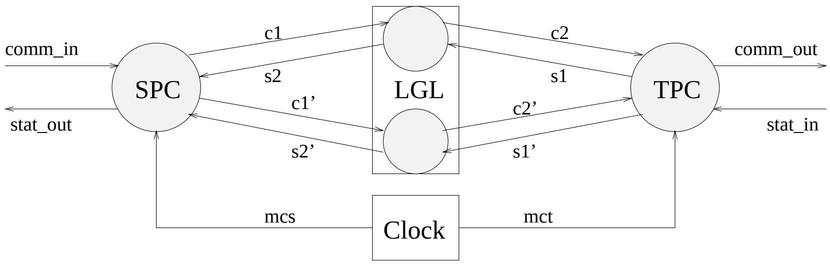

FIGURE 4. The fault-tolerant model

TABLE 6. The fault-tolerant model (Part 1)

```
* level 0: fail_overfull, det, recovered, mcs, mct
* level 1: online (ok, ok'), init, ack_init, switch, ack_switch, reset
* level 2: out (c2, s2), out' (c2', s2'), fail_wire, repaired,
*          comm_in, comm_out, stat_in, stat_out
* level 3: in (c1, s1), in' (c1', s1')
* level 4: outu (c2u, s2u), outu' (c2u', s2u'), tick

* Slow-scan system

bi SS     (SPC | LGL | TPC | Clock)\{c1:3,c2:2,c2u:4,s1:3,s2:2,s2u:4,
                                    c1':3,c2':2,c2u':4,s1':3,s2':2,
                                    s2u':4,mcs:0,mct:0,ok:1,ok':1,
                                    init:1,ack_init:1,reset:1,
                                    switch:1,ack_switch:1}
```

better safety-critical behavior [11]. Therefore, we explicitly duplicate the data path in our design. The new situation is depicted in Figure 4, where the LGL consists of the usual link and a spare link whose corresponding actions are annotated by a prime. Our fault-tolerant model, where the 'prime' states of the SPC and TPC indicate that the system works on the second line, behaves as follows. If a failure of the currently used line is detected by the SPC (TPC), i.e. it is in its failure state, and the other line is 'up', then the SPC (TPC) signals its wish to switch the line to the TPC (SPC) by performing the action `'switch`. The interrupt handlers react on that request (action `'ack_switch`) in the same fashion as in the recovery model. Tables 6 and 7 summarize the necessary changes to our model.

Ideally, we assume the switch to be atomic in the *design* of the slow-scan system, i.e. SPC and TPC switch to the new link at the same time. Using priorities, this can be modeled by giving the actions `switch` and `ack_switch` the same priority as the interrupt actions `init`, `ack_init`, and `ok`.

TABLE 7. The fault-tolerant model (Part 2)

```
* SSI-side protocol converter (SPC) ... (changes to TPC analogue)

bi SPC     SPC0 [> (init:1.'ack_init:1.#'recovered:0.SPC +
                    ack_init:1.SPC +
                    switch:1.'ack_switch:1.#'recovered:0.SPC' +
                    ack_switch:1.SPC')
...
bi SPCF    comm_in:2.SPCF + s2:2.SPCF + s2u:4.SPCF + mcs:0.SPCF +
           ok:1.'reset:0.'init:1.nil + ok':1.'reset:0.'switch:1.nil

bi SPC'    SPC0' [> (init:1.'ack_init:1.#'recovered:0.SPC' +
                     ack_init:1.SPC' +
                     switch:1.'ack_switch:1.#'recovered:0.SPC +
                     ack_switch:1.SPC)

bi SPC0'   comm_in:2.'stat_out:2.SPC0' + 'c1':3.SPC0' + s2':2.SPC0' +
           s2u':4.SPC0' + mcs:0.'c1':3.SPC1'
...
bi SPCF'   comm_in:2.SPCF' + s2':2.SPCF' + s2u':4.SPCF' + mcs:0.SPCF' +
           ok':1.'reset:0.'init:1.nil + ok:1.'reset:0.'switch:1.nil

* Low grade link (LGL)

bi LGL     Comm00[c1:3/in:3,s1:3/in':3,c2:2/out:2,s2:2/out':2,
                  c2u:4/outu:4,s2u:4/outu':4,ok:1/online:1] |
           Comm00[c1':3/in:3,s1':3/in':3,c2':2/out:2,s2':2/out':2,
                  c2u':4/outu:4,s2u':4/outu':4,ok':1/online:1]
```

3.5 State Space of the Models

We have run the NCSU Concurrency Workbench on a SUN SPARC 20 workstation to construct and minimize the state spaces of our models. We refer to the slow-scan model as `bruns.pccs`, to the recovery model as `recovery.pccs`, and to the fault-tolerant model as `ftolerant.pccs`. Moreover, we refer to the slow-scan model where the buffer has been replaced by the full-duplex version as `basic.pccs`. The CCS models corresponding to `bruns.pccs` and `basic.pccs`, which are obtained by leaving out all priority values, are called `bruns.ccs` and `basic.ccs`, respectively. Table 8 provides for each model the number of states and transitions of the corresponding transition system and the time (in seconds) needed for constructing it. The table also contains this information for the minimized (with respect to prioritized strong bisimulation) models (cf. "reduced state space").

Table 8 shows that the number of states decreases by over 70% when using the calculus with priorities. Even more impressive is the reduction of transitions by approximately 85%. This results from the fact that we are not able to model the *atomicity of action sequences* and *interrupts* in plain

TABLE 8. Transition system sizes

Model Name	Global State Space			Reduced State Space		
	states	trans.	time	states	trans.	time
`bruns.ccs`	3527	17122	8	3154	14894	38
`bruns.pccs`	899	2567	6	766	2094	11
`basic.ccs`	1114	4721	2	1021	4217	9
`basic.pccs`	312	801	2	287	713	3
`recovery.pccs`	1100	2801	20	789	2233	25
`ftolerant.pccs`	11905	33760	452	7485	26164	552

CCS. That observation demonstrates the utility of priorities for the verification of distributed systems. The large reduction of the fault-tolerant model with respect to prioritized strong bisimulation is due to the symmetry of its design. The minimization of `bruns.ccs` and `basic.ccs` with respect to observational equivalence reduces the model to 2057 states / 8280 transitions and 698 states / 2293 transitions, respectively. We are currently implementing an algorithm to compute *prioritized observational* equivalence. The adaption of the corresponding observational equivalence [14] is not suitable for automated verification tools since the weak transition relation is parameterized with (arbitrary) sets of actions. However, we have developed a characterization of the prioritized observational equivalence which uses an alternative weak transition relation that is efficiently computable.

4 Verifying the Railway System

In this section, we specify and verify requirements of the slow-scan, the recovery, and the fault-tolerant model. We use the well-known modal μ-calculus [12] as our specification language and determine the validity of our properties by model checking [3]. For the verification, we use the NCSU Concurrency Workbench on a SUN SPARC 20 workstation with 512 MByte of main memory.

4.1 Requirements of the Slow-Scan System

Since the slow-scan system is embedded in a safety-critical system, we want to verify that our designs satisfy the following required properties.

After a low-grade link fails, either the slow-scan system will eventually detect the error or the link is repaired. Moreover, this property holds after every reinitialization of the system.

The slow-scan system is always capable of continuing to tick. If this property holds, then the system is *deadlock-free*, too.

A failure of the slow-scan system is possible. This property should also be valid after every reinitialization of the system.

A failure is detected only if a failure has actually occurred. Also this property should hold after every reinitialization of the system.

After a low-grade link fails, the slow-scan system will eventually stop responding to the SSI and TFMs if the low-grade link does not recover from the error.

If a failure is detected and the broken line is repaired, then the system will be reinitialized.

All properties – except for the last one – are adapted from [5]. However, since the recovery and the fault-tolerant model are able to recover from an error, the properties of [5] should also hold after every reinitialization of these models.

4.2 Specifying our Requirements in the μ-Calculus

For specifying our requirements we use the modal μ-calculus. Its syntax is defined by the following BNF, which uses a set of variables $\mathcal{V}$ with $X \in \mathcal{V}$.

$$\Phi \quad ::= \quad \mathit{ff} \quad | \quad X \quad | \quad \neg\Phi \quad | \quad \Phi \wedge \Phi \quad | \quad \langle \alpha{:}k \rangle \Phi \quad | \quad \mu X.\Phi$$

Further, we define the following dual operators: $\mathit{tt} =_{\mathrm{df}} \neg \mathit{ff}$, $\Phi_1 \vee \Phi_2 =_{\mathrm{df}} \neg(\neg\Phi_1 \wedge \neg\Phi_2)$, $[\alpha{:}k]\Phi =_{\mathrm{df}} \neg\langle \alpha{:}k\rangle(\neg\Phi)$, and $\nu X.\Phi =_{\mathrm{df}} \neg\mu X.(\neg\Phi[\neg X/X])$, where $[\neg X/X]$ denotes the substitution of all free occurrences of X by $\neg X$. Finally, we introduce the following abbreviations where $L \subseteq \mathcal{A}$: $\langle L\rangle\Phi =_{\mathrm{df}} \bigvee\{\, \langle\alpha{:}k\rangle\Phi \mid \alpha{:}k \in L \,\}$, $\langle -\rangle\Phi =_{\mathrm{df}} \langle\mathcal{A}\rangle\Phi$, $\langle -L\rangle\Phi =_{\mathrm{df}} \langle\mathcal{A}\setminus L\rangle\Phi$, $[L]^\infty\Phi =_{\mathrm{df}} \nu X.\Phi \wedge [L]X$, and $\langle L\rangle^*\Phi =_{\mathrm{df}} \mu X.\Phi \vee \langle L\rangle X$.

We use the well-known standard semantics of the modal μ-calculus as e.g. presented in [12]. The model checker integrated in the NCSU Concurrency Workbench is based on [3]. More precisely, it is a *local* model checker for a fragment of the modal μ-calculus. The formulae we intend to verify can be automatically rewritten into semantical equivalent ones which satisfy the syntactic restriction required in [3]. The time and space complexity of the model checker is linear in the size of the formula, in its alternation depth, and in the size of the transition system.

We now formally specify the requirements of the slow-scan system as presented above in the modal μ-calculus. We take particular care in implementing *eventuality* since we want to consider only execution paths in which the clock continues to tick. This *fairness property* is not satisfied by the models themselves since they contain livelocks. As discussed in [5], we are interested in the following notion of *fair eventuality*:

$$\mathit{even}(\Phi) =_{\mathrm{df}} \mu X.\,(\nu Y.\,\Phi \vee ([\overline{\texttt{tick}}]X \wedge [-\overline{\texttt{tick}}]Y)) \ .$$

Moreover, we need a meta-formula which expresses that the argument formula holds again if the low-grade system has recovered:

$$\mathit{again}(\Phi) =_{\mathrm{df}} [-]^\infty[\overline{\texttt{recovered}}]\Phi \ .$$

Now, we can formalize the desired properties of the slow-scan model and the fault-tolerant model, where $\texttt{Fail} =_{\rm df} \{\overline{\texttt{fail\_wire}}, \overline{\texttt{fail\_overfull}}\}$.

After the low-grade link fails (for the first time), the slow-scan system will eventually detect the error or the link is repaired:

$$\begin{array}{l} \textit{failures-responded} =_{\rm df} \\ [-\texttt{Fail}]^{\infty}[\texttt{Fail}]\,\textit{even}(\langle\overline{\texttt{det}}\rangle tt \vee \langle\overline{\texttt{repaired}}\rangle tt) \;\; . \end{array} \tag{1}$$

The formula *failures-responded* holds after every reinitialization of the system again:

$$\textit{failures-responded-again} =_{\rm df} \textit{again}(\textit{failures-responded}) \;\; . \tag{2}$$

Since the formula *failures-responded* is trivially true if the underlying model cannot perform the action **'tick** or if it cannot fail, we are also interested in the following two properties. The slow-scan model is always capable of continuing to tick:

$$\textit{can-tick} =_{\rm df} [-]^{\infty}\langle-\rangle^{*}\langle\overline{\rm tick}\rangle tt \;\; . \tag{3}$$

A failure of the slow-scan system is possible:

$$\textit{failures-possible} =_{\rm df} \langle-\overline{\texttt{recovered}}\rangle^{*}\langle\texttt{Fail}\rangle tt \;\; . \tag{4}$$

The formula *failures-possible* holds after every reinitialization of the system again:

$$\textit{failures-possible-again} =_{\rm df} \textit{again}(\textit{failures-possible}) \;\; . \tag{5}$$

A failure is detected only if a failure has actually occurred since the last reinitialization of the system:

$$\begin{array}{l} \textit{no-false-alarms} =_{\rm df} \\ [-\texttt{Fail}, \overline{\texttt{recovered}}]^{\infty}([\overline{\texttt{det}}]\textit{ff} \vee \langle\overline{\texttt{fail\_overfull}}\rangle tt) \;\; . \end{array} \tag{6}$$

The body of the formula reflects that $\overline{\texttt{fail\_overfull}}$ signals that a failure has already occurred, i.e. it may be enabled at the same time as $\overline{\texttt{det}}$. Moreover, the formula *no-false-alarms* should hold after every reinitialization of the system:

$$\textit{no-false-alarms-again} =_{\rm df} \textit{again}(\textit{no-false-alarms}) \;\; . \tag{7}$$

The auxiliary property "the system never responds", which is used below, can be encoded as follows:

$$\textit{silent} =_{\rm df} [-]^{\infty}[\overline{\texttt{comm\_out}}, \overline{\texttt{stat\_out}}]\textit{ff} \;\; .$$

After a low-grade link fails, the slow-scan system will eventually be silent if the low-grade link does not recover from the error:

$$\textit{eventually-silent} =_{\rm df} [-]^{\infty}[\overline{\texttt{det}}]\,\textit{even}(\texttt{silent} \vee \langle\overline{\texttt{recovered}}\rangle tt) \, . \tag{8}$$

If a failure is detected and the broken line is repaired, then the system will be reinitialized:

$$\begin{aligned} \textit{react-on-repair} =_{\mathrm{df}} [-]^{\infty}[\underline{\mathtt{det}}]([-\overline{\mathtt{recovered}}]^{\infty}[\underline{\mathtt{repaired}}] \\ \textit{even}(\langle\overline{\mathtt{recovered}}\rangle tt)) \ . \end{aligned} \tag{9}$$

4.3 Verification Results

We applied the model checker for all models and formulae twice. The first time, we let the model checker construct the state space on the fly as is usual for *local* model checking. The second time, before invoking the model checker, we minimized the models with respect to prioritized strong bisimulation. The verification results are given in Tables 9/10 and 11/12, respectively. The tables show which properties hold for which formulae (columns "ok") and give the CPU time (in seconds) used by the NCSU Concurrency Workbench for checking each formula. The symbol "?" indicates that a computation ran out of memory. The speed-up of the model checker for minimized models is partly due to the fact that the transition systems for our models were constructed in a preprocessing step (minimization) and, thus, are *not* constructed on the fly. Additionally, the times for the verification results with respect to our minimized models do not include the times needed for the minimizations.

TABLE 9. Verification results wrt. the non-minimized models

Non-minimized models	Formula 1		Formula 3		Formula 4		Formula 6		Formula 8	
	ok	time	ok	time	ok	time	ok	time	ok	time
`bruns.ccs`	*ff*	1936	*tt*	1811	*tt*	1	*ff*	6	?	?
`bruns.pccs`	*tt*	1483	*tt*	261	*tt*	1	*tt*	52	*tt*	1164
`basic.ccs`	*ff*	1460	*tt*	524	*tt*	1	*ff*	19	*tt*	7541
`basic.pccs`	*tt*	396	*tt*	86	*tt*	1	*tt*	43	*tt*	331
`recovery.pccs`	*tt*	808	*tt*	495	*tt*	1	*tt*	68	*tt*	1685
`ftolerant.pccs`	*tt*	1306	?	?	*tt*	1	*tt*	135	?	?

TABLE 10. Verification results wrt. the non-minimized models (continued)

Non-minimized models	Formula 2		Formula 5		Formula 7		Formula 9	
	ok	time	ok	time	ok	time	ok	time
`recovery.pccs`	*tt*	1942	*tt*	429	*tt*	492	*tt*	2581
`ftolerant.pccs`	?	?	?	?	?	?	?	?

In contrast to [5], we could verify most properties automatically and without using any abstractions by hand. However, the formulae *eventually-silent*, *react-on-repair*, and *failures-responded-again* are complicated and

TABLE 11. Verification results wrt. the minimized models

Minimized models	Formula 1		Formula 3		Formula 4		Formula 6		Formula 8	
	ok	time	ok	time	ok	time	ok	time	ok	time
`bruns.ccs`	*ff*	1570	*tt*	366	*tt*	1	*ff*	18	?	?
`bruns.pccs`	*tt*	604	*tt*	90	*tt*	1	*tt*	17	*tt*	512
`basic.ccs`	*ff*	225	*tt*	106	*tt*	1	*ff*	7	*tt*	763
`basic.pccs`	*tt*	213	*tt*	32	*tt*	1	*tt*	16	*tt*	142
`recovery.pccs`	*tt*	90	*tt*	86	*tt*	1	*tt*	16	*tt*	423
`ftolerant.pccs`	*tt*	122	*tt*	864	*tt*	1	*tt*	16	?	?

TABLE 12. Verification results wrt. the minimized models (continued)

Minimized models	Formula 2		Formula 5		Formula 7		Formula 9	
	ok	time	ok	time	ok	time	ok	time
`recovery.pccs`	*tt*	481	*tt*	96	*tt*	109	*tt*	768
`ftolerant.pccs`	?	?	*tt*	911	*tt*	1159	?	?

large in size. Therefore, we could not check them automatically for the model `ftolerant.pccs`, which has 7485 states and 26164 transitions after minimization. Although the size of `bruns.ccs` is relatively small, the model checker ran out of memory for the formula *eventually-silent*, which has alternation depth two.

Moreover, our timing results show that using a *local* model checker often gains no advantage. This is because most of the formulae are valid safety properties, and the local model checker has to investigate all states of the models anyway. However, our model checker has quickly detected invalid formulae.

In the prioritized models all properties that could be verified automatically hold as expected. The formula *no-false-alarms* does not hold for the models in plain CCS. This is due to the fact that the atomicity of actions cannot be expressed without priorities. Indeed, there exists an interleaving in the CCS models where one observes a `'det` before a failure has occurred.

Surprisingly, we found the formula *failures-responded* invalid in the model `bruns.ccs` whereas in [5] it is reported to hold. The reason for this is that we left out the actions `c1'` (`s1'`) which occur directly before a `'det` in Bruns' model. Although that reflects our intuition that a `'det` should be signaled as soon as an error is detected, Bruns' modeling does not allow both SPC and TPC to detect the overfull-failure of the medium before the action `'fail_overfull` has occurred.

5 Conclusions and Future Work

We have demonstrated the importance of priorities for modeling and verifying distributed systems by means of a practically relevant case study of the slow-scan part of a railway signaling system. Priorities allow us to favor one communication over another and to make action sequences atomic. While the former helps to model systems more realistically, the latter drastically cuts the number of states and transitions. Our models explicitly reflect safety-critical parts of the slow-scan system, namely an error-recovery scheme and a fault-tolerant medium, which are required in the design document [11]. We have used the NCSU Concurrency Workbench for checking properties of our design. We are currently implementing an algorithm for computing prioritized observational equivalence that will enable us to further reduce the size of our models.

Acknowledgments: We want to thank Girish Bhat for the implementation of his model checker in the NCSU Concurrency Workbench.

6 References

[1] J. Baeten, editor. *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, England, 1990.

[2] J. Baeten, J. Bergstra, and J. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. *Fundamenta Informaticae IX*, pages 127–168, 1986.

[3] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal μ-calculus. To appear in Proceedings of *Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS '96), 1996.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[5] G. Bruns. A case study in safety-critical design. In G. Bochmann and D. Probst, editors, *Computer Aided Verification (CAV '92)*, volume 663 of *Lecture Notes in Computer Science*, pages 220–233, Montréal, June/July 1992. Springer-Verlag.

[6] J. Camilleri and G. Winskel. CCS with priority choice. In *Sixth Annual Symposium on Logic in Computer Science (LICS '91)*, pages 246–255, Amsterdam, July 1991. IEEE Computer Society Press.

[7] R. Cleaveland. Analyzing concurrent systems using the Concurrency Workbench. In P. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 129–144. Springer-Verlag, 1993.

[8] R. Cleaveland and M. Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2):58–77, July/August 1990.

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

[10] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[11] A. Cribbens. Solid-state interlocking (SSI): an integrated electronic signalling system for mainline railways. *IEE Proceedings*, 134, Pt. B(3), May 1987.

[12] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[13] R. Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.

[14] V. Natarajan, L. Christoff, I. Christoff, and R. Cleaveland. Priorities and abstraction in process algebra. In P. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 217–230, Madras, India, Dec. 1994. Springer-Verlag.

[15] W. Yi. CCS + time = an interleaving model for real time systems. In J. L. Albert, B. Monien, and M. R. Artalejo, editors, *Automata, Languages and Programming (ICALP '91)*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228, Madrid, July 1991. Springer-Verlag.