LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - The University of Edinburgh

A Case Study in Safety-Critical Design

by

Glenn Bruns

LFCS Report Series
LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

ECS-LFCS-92-239
September 1992

A Case Study in Safety-Critical Design

Glenn Bruns

Department of Computer Science, University of Edinburgh
Edinburgh EH9 3JZ, UK

Abstract. We have modelled the design of a safety-critical railway system in the process calculus CCS, described important properties of the design in temporal logic, and verified with the Concurrency Workbench that some of the properties hold of the model. Verifying properties of a design, rather than an implementation, presented special problems, particularly in capturing in the formal model the kinds of abstraction found in the design, and in showing that the verified properties would also hold in all implementations of the design.

1 Introduction

Many case studies demonstrating the verification of distributed systems involve communication protocols, low-level algorithms, or hardware. A less-studied topic is the verification of system designs. A design describes structure, such as the top-level components of a system and their interconnection, as well as behaviour, such as the responses of components to inputs. Verifying properties of a design allows design decisions to be checked before spending much, possibly wasted, implementation effort. Since the design contains less detail than a full implementation, the verification task may also be more tractable.

We describe here our experience in attempting to verify properties of the design of a safety-critical system. We had three specific goals. First, to formalize the key parts of the system design as a CCS process, leaving more detailed design issues open. Second, to formalize safety-critical properties of the system as temporal logic formulas and show, using an automatic verification tool, that these properties hold of the model. Finally, to prove that any "acceptable implementation" would also possess the properties shown to hold of the design model. By "acceptable implementation" we are intentionally vague. Such an implementation could be one reached from the design systematically according to a set of refinement rules, or simply one satisfying certain ad-hoc, application-specific constraints.

2 Background

The function of British Rail's Solid State Interlocking (SSI) [7] is to adjust, at the request of the signal operator, the settings of signals and points in the railway to permit the safe passage of trains. "Safe", in this context, means that the system will protect the signal operator from inadvertently sending trains along routes

that could lead to a collision or derailment. The entire BR network is controlled by many SSI's, each responsible for one sub-network.

Figure 1 depicts an SSI and the devices it controls. Safe commands issued from the control panel are allowed to effect signals and points via messages sent over a high-speed communication link to trackside functional modules (TFM's). Each message is received by all TFM's connected to the link, but only acted on by the TFM with the address specified in the message.

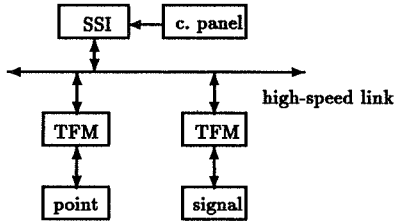

Fig. 1. The SSI and its environment

A safety-related features of the SSI is its pattern of communication with the TFM's. Instead of sending signal or point commands only as needed, the SSI sends a message of the form $\langle TFM\ address, state \rangle$ to each attached TFM about once every second, giving the intended current setting. These messages are sent in a predefined cyclic pattern, called a *major cycle*, one TFM after another. After sending a message, the SSI waits at most a few milliseconds for the addressed TFM to respond with the current state of the device. This scheme allows failures of the TFM's and communication link to be detected quickly, and forces devices that have autonomously changed state to return to their proper state.

In some sections of the BR network, many miles of track lie between TFM's, making the high-speed link very expensive. A cheaper low-speed link could not be directly adopted as it would not provide the bandwidth needed for the TFM command cycling scheme. On the other hand, dropping the scheme would compromise safety and force changes to the SSI and TFM's. A solution is to employ a *slow-scan* system, built of a low-speed link (also called a *low-grade link*, or *LGL*) with *protocol converters* at each end (see Figure 2). The SSI-side protocol converter (SPC) accepts TFM commands once every second, and responds immediately with trackside device status, but only sends the TFM command along the low-grade link occasionally. The TFM-side protocol converter (TPC) sends a command to each attached TFM once every second. Responses from the TFM's are occasionally sent by the TPC to the SPC, in order to update the SPC's device status information. Because the SPC mimics a TFM, and the TPC mimics an SSI, the slow-scan system can replace a section of high-speed link, although the safety and performance properties will be altered.

The slow-scan system must also mimic the high-speed link in its failure be-

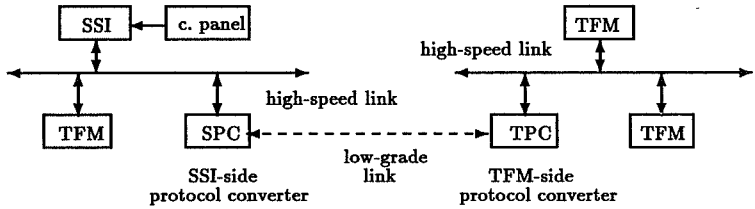

Fig. 2. The slow-scan system

haviour. For example, if the high-speed link fails so that the attached TFM's stop receiving messages, then the TFM's will detect the problem and put their outputs to a safe state: signals to red; points locked in their current setting. Therefore, if the low-grade link of the slow-scan system fails, the TPC should stop sending commands to the attached TFM's.

3 Safety Considerations

The development of any formal model is guided by the purposes to which that model will be put. Here we are interested in showing safety-critical properties of our model, so before building the model we should consider the kinds of properties that we will try to show.

Taken on its own, the slow-scan system cannot be considered safety-critical, since it does not directly control physical devices. However, by first looking at the safety-critical properties of the train routing system as a whole, and then working through the levels of the system, it is possible to obtain derived safety requirements of the slow-scan system. There is not room here to present such a hierarchical safety analysis. We will simply observe that the safe routing of trains depends on the *timely and error-free delivery of commands* and *timely detection of failures.*

The SSI can satisfy these requirements if a high-speed link is used. The bandwidth of the link and the error coding of messages ensures that the first requirement is met. The TFM-command cycling scheme ensures the second is met. Both requirements are threatened by slow-scan, however. Its low-grade link slows the delivery of messages and lacks the bandwidth needed to send many redundant messages. In this study, we focus on failures of the low-grade link and their detection by the slow-scan system.

4 A Simple Model of Slow-Scan

We first present a simple model of the slow-scan system and then show how the model can be extended to include the occurrence and handling of LGL failures.

Our goal is to capture the aspects of the design relevant to the safety-critical properties of interest, and to leave detailed design issue open. For example, we would like our model to be independent of a policy that determines when TFM commands should be passed along the low-grade link. Furthermore, we want to keep the model simple enough to enable mechanical verification with the Concurrency Workbench.

We will model the slow-scan system as CCS processes. Only a brief and informal overview of CCS will be given here.

Processes are described in CCS by terms for which the only possible behaviour is to perform *actions*, which are either names (a,b,c,...), co-names ($\overline{a}$,$\overline{b}$,$\overline{c}$,...), or the special action τ.

Process terms have the following syntax, where L ranges over sets of actions and f ranges over functions from actions to actions:

$$P ::= 0 \mid a.P \mid P_1 + P_2 \mid P_1 \mid P_2 \mid P\backslash L \mid P[f]$$

The term 0 denotes the nil process, which can perform no actions. The operator . expresses sequential action. The process $a.P$ can perform the action a and thereby become process P. The operator + expresses choice. If P_1 can perform a and become P_1', then so can $P_1 + P_2$, and similarly for P_2. The operator $\mid$ expresses parallel execution. If process P_1 can perform a and become P_1', then $P_1 \mid P_2$ can perform a and become $P_1' \mid P_2$, and similarly for P_2. Furthermore, if P_1 can perform a and become P_1', and P_2 can perform $\overline{a}$ and become P_2', then $P_1 \mid P_2$ can perform τ and become $P_1' \mid P_2'$. The operator $\backslash$ expresses the restriction of actions. If P can perform a and become P', then $P\backslash L$ can only perform a to become $P'\backslash L$ if $a, \overline{a} \notin L$. Finally, $P[f]$ expresses the relabelling of actions. if P can perform a and become P', then $P[f]$ can perform $f(a)$ and become $P'[f]$. A relabelling function f has the property that $f(\tau) = \tau$, and $f(\overline{a}) = \overline{f(a)}$.

The idea of repetition is captured by allowing recursive process definitions of the form $P \stackrel{\text{def}}{=} E$, where P is a process constant and E is a process term possibly containing P. For example, the process defined by $P \stackrel{\text{def}}{=} a.P + b.0$ has the possibility of performing either a or b, and continues to have this possibility as long as action a is performed. Once action b is performed, the process terminates.

The set of actions that can be eventually performed by a process is called its *sort*. For example, the sort of $P \stackrel{\text{def}}{=} a.P + b.0$ is $\{a, b\}$.

We are ready to present the first, simple model of the slow-scan system. The flow diagram for our first model (see Figure 3) shows that the SSI and TFM's are considered outside the boundary of the slow-scan system. Because we are using CCS to model the system, we will admittedly be able to say little about real-time and probabilistic aspects of the system's behaviour.

The LGL component of the model will be formalized first. Few details about the LGL interface are given in the high-level design, so the model is based on what one might expect in a typical communication link. For example, messages can be written to the input port or read from the output port at any time. This feature ensures that the protocol converters need never wait on the LGL ports. We would like our model to say as little as possible about the content of messages

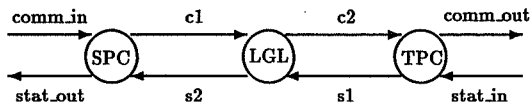

Fig. 3. Flow diagram for a simple slow-scan model

and the message buffering capacity of the LGL, although we naturally expect it to have only finite buffering capacity. Our CCS model of the LGL is as follows:

$$LGL \stackrel{\text{def}}{=} Comm[c1/in, c2/out, c2_u/out_u] \mid Comm[s1/in, s2/out, s2_u/out_u]$$
$$Comm \stackrel{\text{def}}{=} in.Comm' + \overline{out_u}.Comm$$
$$Comm' \stackrel{\text{def}}{=} in.Comm' + \overline{out}.Comm$$

The model contains two concurrent processes, one for each direction of message flow. Messages are modelled simply as content-less"pulses". Each link process can buffer a single message, because it is not possible in CCS to describe a buffer with finite but arbitrary capacity. If a message arrives at the input port of a link already buffering a message, the buffer is overwritten. If the output port of a link is read while the link is empty, the special action $\overline{out_u}$ occurs.

The slow-scan design requires of the SPC that SSI commands are responded to immediately with TFM status information. This requirement ensures that the slow-scan system properly mimics the behaviour of a high-speed link. The design also states that the SPC must pass commands along the LGL to the TFM, but the policy for determining which commands should be passed, and when they should be passed, is left open. Similarly, the policy for updating TFM status information with messages sent from the TFM is left open in the design. The CCS process that models the SPC is as follows:

$$SPC \stackrel{\text{def}}{=} comm\_in.\overline{stat\_out}.SPC +$$
$$\overline{c1}.SPC + s2.SPC + s2_u.SPC$$

We have attempted to leave a policy for passing command and status messages open by allowing actions $\overline{c1}$ and $s2$ to occur at any time (except just after the receipt of an SSI command). This SPC model says too little about a message-passing policy in one sense – since our model need never pass command messages along – and too much in another sense – since a message-passing policy need not be capable of sending a command at every instant. Unfortunately, it does not seem possible to express the model we would like with a process algebra such as CCS.

The TPC is modelled much like the SPC. Here we would like the model to say nothing about passing status information along the TFM and reading command information from the LGL. Our CCS model of the TPC is as follows:

$$TPC \stackrel{\text{def}}{=} \overline{comm\_out}.stat\_in.TPC +$$
$$\overline{s1}.TPC + c2.TPC + c2_u.TPC$$

By composing processes TPC, LGL, and SPC, we get the complete model SS, with sort $\{comm\_in, \overline{comm\_out}, stat\_in, \overline{stat\_out}\}$.

$$SS \stackrel{\text{def}}{=} (SPC \mid LGL \mid TPC) \backslash \{c1, c2, c2_u, s1, s2, s2_u\}$$

5 Modelling Low-grade Link Failures

Our simple model captures the basic operation of the slow-scan system: the SPC immediately responds to SSI commands with TFM state information; the SPC and TPC occasionally write command or status messages to the LGL input ports; and the SPC and TPC occasionally read the LGL output ports. The slow-scan design additionally discusses failure detection, but does not describe any specific failure detection mechanisms. Since we are most interested in the safety-critical aspects of the system, we will add some failure modes to our model, and a mechanism for failure detection and handling.

We will consider two LGL failure modes. The LGL can fail if its buffering capacity is exceeded, and it can fail spontaneously because of a break in the communication medium. We assume that once a failure occurs the LGL will continue to accept messages, but will never deliver messages. The revised model of the LGL incorporating these failure modes is as follows:

$$LGL \stackrel{\text{def}}{=} Comm[c1/in, c2/out, c2_u/out_u] \mid Comm[s1/in, s2/out, s2_u/out_u]$$

$$Comm \stackrel{\text{def}}{=} in.Comm' + \overline{out_u}.Comm + \overline{fail}.Comm''$$

$$Comm' \stackrel{\text{def}}{=} in.\overline{fail}.Comm'' + \overline{out}.Comm + \overline{fail}.Comm''$$

$$Comm'' \stackrel{\text{def}}{=} in.Comm'' + \overline{out_u}.Comm''$$

The new action $\overline{fail}$ occurs when a failure occurs on either of the links comprising the LGL.

Because the slow-scan system is intended to simulate a high-speed link, an LGL failure should cause the slow-scan system to simulate a high-speed link failure. The SSI detects such a failure when the TFM fails to respond to a command. Conversely, the TFM detects a high-speed link error when the SSI fails to send a command. Therefore, once an LGL failure is detected, the slow-scan system should stop responding to SSI commands and stop sending TFM commands.

The problem in detecting an LGL failure in the basic slow-scan model is that the LGL may be inactive for long periods in the normal course of events. An LGL failure detection strategy cannot report that a failure has occurred simply because no message has been received over the LGL after some period of time. Thus, to detect failures, additional messages have to be introduced along the LGL.

Consider the more general problem of having two distributed processes detect failures in a communication medium connecting them. Assume that both processes receive pulses from a clock. If one process sends a message once every

clock pulse, and the other process increments a local counter once every clock pulse (clearing the counter if a message is received), then a failure must have occurred in the medium if the counter exceeds a certain bound. The following process Mon illustrates this idea, with S as the sending process and $R(i)$ as the receiving process having local counter value i:

$$
\begin{aligned}
Clock &\stackrel{\text{def}}{=} \overline{tick}.\overline{ts}.\overline{tr}.Clock \\
S &\stackrel{\text{def}}{=} ts.\overline{out}.S \\
R(i) &\stackrel{\text{def}}{=} tr.(\,\textbf{if}\, i > n \,\textbf{then}\, \overline{det}.R(0) \\
&\qquad\qquad\qquad\qquad \textbf{else}\, R(i+1)) \\
&\quad + in.R(0) \\
Mon &\stackrel{\text{def}}{=} (Clock \mid S \mid R(0))\backslash\{tr, ts\}
\end{aligned}
$$

Some new notational features have been introduced here: parameterized actions and conditional statements. The process term $R(i)$ can be regarded as shorthand for the indexed process constant R_i. The term **if** b **then** P_1 **else** P_2 behaves as process P_1 if the boolean expression b evaluates to true, or as the process P_2 if b evaluates to false.

Two such distributed channel monitors can be combined, with a single clock, to check both directions of a bidirectional channel. Each process sends a message each clock pulse and expects to receive a message at least every n clock pulses. Note that although the synchronizing process is named 'Clock', there is no notion of real time in the model.

Incorporating this channel monitoring strategy into our simple model of the slow-scan system, we get the following SSI-side protocol converter process:

$$
\begin{aligned}
SPC(i) &\stackrel{\text{def}}{=} comm\_in.\overline{stat\_out}.SPC(i) \\
&\quad + \overline{c1}.SPC(i) + s2.SPC(0) + s2_u.SPC(i) \\
&\quad + mcs.\overline{c1}.(\,\textbf{if}\, i > n \,\textbf{then}\, \overline{det}.SP \,\textbf{else}\, SPC(i+1)) \\
SPCF &\stackrel{\text{def}}{=} comm\_in.SPCF + s2.SPCF + s2_u.SPCF + mcs.SPCF
\end{aligned}
$$

As before, the SPC accepts commands from the SSI, responds to the SSI with TFM status information, and sometimes sends a command over the LGL. In this new model, the SPC is guaranteed to send at least one message over the LGL each clock tick. Also, if n clock ticks pass without the receipt of a message from the TPC, then the SPC enters failure mode, in which it never again sends messages to the SSI or LGL. Of course, a more detailed model would contain a mechanism by which the SPC could exit failure mode.

Similarly, the TFM-side protocol converter process is as follows:

$$
\begin{aligned}
TPC(i) &\stackrel{\text{def}}{=} \overline{comm\_out}.stat\_in.TPC(i) \\
&\quad + \overline{s1}.TPC(i) + c2.TPC(0) + c2_u.TPC(i) \\
&\quad + mct.\overline{s1}.(\,\textbf{if}\, i > n \,\textbf{then}\, \overline{det}.TPCF \,\textbf{else}\, TPC(i+1)) \\
TPCF &\stackrel{\text{def}}{=} stat\_in.TPCF + c2.TPCF + c2_u.TPCF + mct.TPCF
\end{aligned}
$$

The clock process is as follows:

$$Clock \stackrel{\text{def}}{=} \overline{tick}.\overline{mcs}.\overline{mct}.Clock$$

The complete model of the slow-scan with LGL failure detection:

$$SS \stackrel{\text{def}}{=} (SPC(0) \mid LGL \mid TPC(0) \mid Clock) \backslash \{c1, c2, c2_u, s1, s2, s2_u, mcs, mct\}$$

6 Analysis of the Model

6.1 Formalizing the Safety Properties

We will try to show that two safety-critical properties hold of our slow-scan model:

- After a low-grade link fails, the slow-scan system will eventually detect the failure and stop responding to the SSI and TFM.
- A failure is detected only if a failure has actually occurred.

We use the modal mu-calculus [10] in a slightly extended form [15] as a temporal logic to formalize behavioural properties. The syntax of the mu-calculus is as follows, where L ranges over sets of actions and Z ranges over variables:

$$\phi ::= \neg\phi \mid \phi_1 \wedge \phi_2 \mid [L]\phi \mid Z \mid \nu Z.\phi$$

Informally, the formula $\neg\phi$ holds of a process P if ϕ does not hold of P. The formula $\phi_1 \wedge \phi_2$ holds of P if both ϕ_1 and ϕ_2 hold of P. The formula $[L]\phi$ holds of P if ϕ holds for all processes P' that can be reached from P through the performance of action $\alpha \in L$. The formula $\nu Z.\phi$ is the greatest fixed point of the recursive modal equation $Z = \phi$, where Z appears in ϕ. Some intuition about fixed point formulas can be gained by keeping in mind that $\nu Z.\phi$ can be replaced by its "unfolding": the formula ϕ with Z replaced by $\nu Z.\phi$ itself. Thus, $\nu Z.\psi \wedge [\{a\}]Z = \psi \wedge [\{a\}](\nu Z.\psi \wedge [\{a\}]Z) = \psi \wedge [\{a\}](\psi \wedge [\{a\}](\nu Z.\psi \wedge [\{a\}]Z)) = \ldots$ holds of any process for which ψ holds along any execution path of a actions.

The operators $\vee$, $\langle\alpha\rangle$, and μ can be defined as duals to existing operators (where $\phi[\psi/Z]$ is the property obtained by substituting ψ for free occurrences of Z in ϕ):

$$\phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$$
$$\langle L\rangle\phi \stackrel{\text{def}}{=} \neg[L]\neg\phi$$
$$\mu Z.\phi \stackrel{\text{def}}{=} \neg\nu Z.\neg\phi[\neg Z/Z]$$

Informally $\langle L\rangle\phi$ holds of a process that can perform an action in L and thereby evolve to a process satisfying ϕ. As with $\nu Z.\phi$, the formula $\mu Z.\phi$ can be understood through unfolding, except here only finitely many unfoldings can be made. Thus, $\mu Z.\psi \vee \langle\{a\}\rangle Z$ holds of a process that can evolve to a process satisfying ψ after finitely many occurrences of action a.

These additional abbreviations are also convenient (where L ranges over sets of actions, and Act is the set of CCS actions):

$$[\alpha_1, \ldots, \alpha_n]\phi \stackrel{\text{def}}{=} [\{\alpha_1, \ldots, \alpha_n\}]\phi$$
$$[-]\phi \stackrel{\text{def}}{=} [Act]\phi$$
$$[-L]\phi \stackrel{\text{def}}{=} [Act - L]\phi$$

The booleans are defined as abbreviations: $\mathtt{tt} \stackrel{\text{def}}{=} \nu Z.Z$, $\mathtt{ff} \stackrel{\text{def}}{=} \neg\mathtt{tt}$. An example using these abbreviations is $\langle a, b\rangle\mathtt{tt}$, which holds of processes that can perform either an a or a b action. As another example, the formula $\nu Z.\langle -\rangle\mathtt{tt} \wedge [-]Z$ holds of deadlock-free processes.

The mu-calculus along with the abbreviations presented so far constitutes an expressive but still low-level temporal logic for describing properties of processes. In practice it is usually convenient to define additional abbreviations that capture important concepts of the application. Before considering specific properties of the slow-scan system we present two more abbreviations that often make it possible to avoid the fixed-point operators:

$$[L]^{\infty}\phi \stackrel{\text{def}}{=} \nu Z.\phi \wedge [L]Z$$
$$\langle L\rangle^{*}\phi \stackrel{\text{def}}{=} \mu Z.\phi \vee \langle L\rangle Z$$

Informally, $[L]^{\infty}\phi$ holds if ϕ always holds along all paths composed of actions in the set L. For example, the absence of deadlock can be written $[-]^{\infty}\langle -\rangle\mathtt{tt}$ — in every state some action is possible. The dual operator $\langle L\rangle^{*}\phi$ holds of processes having a finite execution path composed of actions from L leading to a state in which ϕ holds.

We are ready now to formalize the two important slow-scan properties. Recall the first property:

> After either of the low-grade links fail, the slow-scan system will eventually detect the failure and stop responding to the SSI and TFM.

There are two distinct parts to this property: a) failures are eventually detected, and b) after detection eventually no responses are made to the TFM's or SSI. On the way to formalizing the first part, we have the idea "after a $\overline{fail}$ action occurs then eventually a $\overline{det}$ action occurs". Care needs to be taken here with the notion of eventuality, however, because we want to consider only executions in which the clock continues to tick. Our CCS model contains execution paths in which the clock does not continue to tick, and we do not expect $\overline{det}$ to eventually occur in all these paths. The next step is therefore to define an abbreviation for the property "if the clock continues to tick then eventually action α will occur":

$$even(\alpha) \stackrel{\text{def}}{=} \mu Z.[-\overline{tick}, \alpha]^{\infty}[\overline{tick}]Z$$

A reasonable translation of this formula to English is "no execution path exists containing infinitely many $\overline{tick}$ actions and no α actions". A useful and

closely-related abbreviation captures the property "if the clock continues to tick then eventually property ϕ holds":

$$even(\phi) \stackrel{\text{def}}{=} \mu Z.(\nu Y.\phi \vee ([\overline{tick}]Z \wedge [-\overline{tick}]Y))$$

Now we can formalize "after a $\overline{fail}$ action occurs then eventually a $\overline{det}$ action occurs":

$$failures\text{-}detected \stackrel{\text{def}}{=} [-\overline{fail}]^{\infty}[\overline{fail}]even(\overline{det})$$

The formula begins with $[-\overline{fail}]^{\infty}$ rather than $[-]^{\infty}$ because we are concerned only with the first failure that occurs.

A potential pitfall is that *failures-detected* is vacuously true if a failure never occurs. So, for example, the nil process 0 satisfies the formula. The formula is also vacuously true of processes in which the clock cannot continue to tick, such as $\overline{fail}.\overline{tick}.0$. To ensure that the slow-scan model does not satisfy failures-detected in one of these ways, we can write two more formulas:

$$failures\text{-}possible \stackrel{\text{def}}{=} \langle - \rangle^{*}\langle \overline{fail} \rangle \mathtt{tt}$$

$$can\text{-}tick \stackrel{\text{def}}{=} [-]^{\infty}\langle - \rangle^{*}\langle \overline{tick} \rangle \mathtt{tt}$$

The formula *failures-possible* says that there is some execution path containing the action $\overline{fail}$. The formula *can-tick* says that, from any state, there is some execution path containing the action $\overline{tick}$. Note that *can-tick* is stronger than the property we need: "$\overline{tick}$ can occur infinitely often after a $\overline{fail}$ action".

To complete the formalization of the first property, we need to also express the second part: "after detection eventually no responses are made to the TFM's or SSI". Using the auxiliary formula *silent*, the property can be expressed as follows:

$$silent \stackrel{\text{def}}{=} [-]^{\infty}[\overline{comm\_out}, \overline{stat\_out}]\mathtt{ff}$$

$$eventually\text{-}silent \stackrel{\text{def}}{=} [-]^{\infty}[\overline{det}]even(silent)$$

The property *silent* expresses that no occurrence of actions $\overline{comm\_out}$ or $\overline{stat\_out}$ is ever possible. The first property of interest is thus fully captured by the conjunction of *failures-detected*, *eventually-silent*, *failures-possible*, and *can-tick*.

The second property, "a failure is detected only if a failure has actually occurred", is much simpler to express:

$$no\text{-}false\text{-}alarms \stackrel{\text{def}}{=} [-\overline{fail}]^{\infty}[\overline{det}]\mathtt{ff}$$

Other properties could be formalized besides the two important safety-critical ones. For example, we have already seen that the absence of deadlock can be expressed as $[-]^{\infty}\langle - \rangle \mathtt{tt}$. However, this property is weaker than the property *can-tick*, which states that not only is some action possible in every state, but that actions leading to a $\overline{tick}$ action are possible in every state.

6.2 Checking the Safety Properties

By formalizing the behaviour of the slow-scan design, we enable precise and even automated reasoning about it. Since the slow-scan model has a finite and reasonably small state space (of 3842 states), the Concurrency Workbench [6] can check whether the properties formulated in the last section hold of the model.

The complexity of some of the properties means that they cannot be checked by the Workbench in 24 hours on a powerful workstation. For each of these properties, checking was made of a smaller model derived from the slow-scan model by hiding actions not relevant to the particular property. Then, by using a technique [3] that cannot be described here because of space, the properties were shown by hand to hold of the full model if and only if they held in the smaller model. In what follows, we will write that a property was checked *indirectly* if this technique was used, and checked *directly* if the Workbench alone was sufficient.

Recall that the first property of interest involved detection of LGL failures. The property *failures-detected* was checked indirectly and shown to hold. The properties *failures-possible* and *can-tick* were also shown to hold, the first directly and the second indirectly.

The property *eventually-silent*, which holds if the slow-scan system eventually stops performing output actions after a failure is detected, could not practically be checked even indirectly. The property is expensive to check because after any $\overline{det}$ action a complicated eventuality property must be shown to hold.

The second property, *no-false-alarms*, expresses that failures cannot be detected before they occur. This property could be checked directly, but was found not to hold. It fails to hold because the action $\overline{fail}$ occurs after a failure, not simultaneously with it. A failure can be detected, and the corresponding action $\overline{det}$ can occur, between the moment of failure and the moment action $\overline{fail}$ occurs. Using the simulation facility of the Workbench, we were able to guide the slow-scan model through such a course of events. The state reached immediately after the $\overline{det}$ action occurred was as follows:

$$(SPCF \mid \overline{fail}.Comm''[f] \mid \overline{fail}.Comm''[f] \mid \overline{s1}.TPC(1) \mid \overline{mct}.Clock)\backslash L$$

Knowing that *no-false-alarms* fails to hold, other questions become interesting, such as "can both protocol converters signal detection of failure before $\overline{fail}$ occurs?", and "if a failure is detected before $\overline{fail}$ occurs, must $\overline{fail}$ eventually occur?". The formula *detects-before-failure* expresses the property that two $\overline{det}$ actions can occur before a $\overline{fail}$ action:

$$\textit{detects-before-failure} \stackrel{\text{def}}{=} \langle -\overline{fail}\rangle^*\langle\overline{det}\rangle\langle -\overline{fail}\rangle^*\langle\overline{det}\rangle \mathtt{tt}$$

This property was checking indirectly (by hiding all actions except $\overline{fail}$ and $\overline{det}$) and shown not to hold. However, it was shown in a similar way that two $\overline{fail}$ actions *can* occur before any $\overline{det}$ actions occur.

The two following formulas express the idea that $\overline{det}$ and $\overline{fail}$ are related by property "if a $\overline{fail}$ occurs before a $\overline{det}$, then eventually a $\overline{det}$ will occur, and

conversely":

$$\textit{even-detect} \stackrel{\text{def}}{=} [-\overline{\textit{fail}}, \overline{\textit{det}}]^{\infty}[\overline{\textit{fail}}]\textit{even}(\overline{\textit{det}})$$
$$\textit{even-fail} \stackrel{\text{def}}{=} [-\overline{\textit{fail}}, \overline{\textit{det}}]^{\infty}[\overline{\textit{det}}]\textit{even}(\overline{\textit{fail}})$$

Note that the property *failures-detected* is slightly stronger than *even-detect*; the former property requires that $\overline{det}$ occurs after $\overline{fail}$ even if $\overline{det}$ occurred before $\overline{fail}$. These properties were checked indirectly and shown to hold. To perform the indirect checking, the actions *comm_in*,*stat_in*,$\overline{comm\_out}$, and $\overline{stat\_out}$ were hidden, yielding a model of 641 states.

In summary, we have verified that LGL failures are detected in our model. To ensure that this property is meaningful we have also shown that in all states the clock can continue to tick, and that it is possible for such failures can occur. However, we have not verified that output actions will eventually cease after an LGL failure is detected.

7 Showing Properties of Implementations

An important goal of this study was to avoid putting features in the model that are not present in the design, so that properties shown of the model would hold of the actual system, regardless of specific choices made during detailed design and implementation. For example, the model of the SPC does not specify when messages are passed along the LGL, only that they can be passed. Less success was achieved in leaving the LGL buffering capacity unspecified; we settled for an LGL model with a capacity of one message.

Unfortunately, we cannot claim that properties shown of our CCS model will hold for all slow-scan implementations, in part because we have given no precise rules governing how a CCS model can be refined.

The observation equivalence relation of CCS is sometimes used to show that a specification and implementation (both described as CCS processes) have the same behaviour. Generally, however, we do not want a refinement relation to be symmetrical. Instead, we expect refinement to be modelled as a pre-order relation in which detail can be added in a refinement step according to some rules.

Bisimulation preorders [16] use the idea that a refinement must be "at least as defined" as a specification. For example, a CCS process that evolves to an error-handling state after an *error* action occurs is a refinement of a process that evolves to an undefined state after such an action. This preorder relation can be characterized logically [14]: process P is a refinement of process Q exactly when P possesses more properties than Q. However, the properties here are those expressible in an intuitionistically-interpreted sub-language of the mu-calculus. Liveness properties, such as *eventually-silent*, cannot be expressed in this logic.

Another refinement preorder comes from the modal process logic of Larsen and Thomsen [11]. Specifications in this logic resemble CCS processes except that both *necessary* and *admissible* actions are possible. A specification R is a

refinement of another specification S if R must perform every action S must perform, and if S may perform every action R may perform. This preorder also has a logical characterization [12], but here the logical language is an intuitionistically-interpreted form of Hennesy-Milner logic, which cannot express the safety and liveness properties of interest to us.

Holmstrom's refinement calculus [9] allows CCS processes to be refined from specifications given in Hennessy-Milner logic with recursion. An implementation is guaranteed to have the property expressed by its specification, but the meaning of a recursive specification is taken to be its greatest fixed point, and so again this approach does not allow liveness properties to be expressed.

8 Conclusions

Our aim was to model the slow-scan design as a CCS process, to prove that the model possessed safety-critical properties expressed in the modal mu-calculus, and to show that these properties would be possessed by implementations based on the design model.

We were able to model most of the slow-scan design directly in CCS. Timing and probabilistic aspects of the design could obviously not be captured. As has been described elsewhere [11], CCS agents are not ideal for specifying systems, since only a single process can be described (up to equivalence), while one often would like to describe a broad class of processes. A notion of priority [4] or of interrupt [2] would have been useful in modelling the *tick* action of the clock process. It would have also been convenient to ascribe simple process fairness [8] to the model, allowing simpler expression of the eventuality properties. In adding a failure detection mechanism to the basic slow-scan model, a notion of superposition [5] would have been helpful. However, a notation with all of these features would lack the appealing simplicity of CCS.

The slow-scan model lacked some features described in the design, such as system initialization. The modelling of failures and failure detection was also overly simple. In particular, many more failure modes of the system could be modelled, and the failure modes could be made more realistic. For example, it may not be valid to assume that the LGL is quiet after failure.

In specifying the safety-critical properties of the system, the modal mu-calculus was expressive enough to capture all the properties of interest. Abbreviations were necessary to keep the formulas small and understandable. As just mentioned, the eventuality properties were complicated by the lack of priority and fairness in the design model. The inability of the Concurrency Workbench to check certain properties in a reasonable period of time reflects both the complexity of the properties and the relative lack of concern with efficiency issues in the development of the Workbench.

Probably the biggest shortcoming of the study was our failure to show that properties of our design model also hold for for slow-scan implementations. This problem is the subject of current study. Also to be studied is the applicability of timed process calculi [1, 13] to this system.

Acknowledgements

We thank Ian Mitchell, Chris Gurney, and others at British Rail Research, Derby, for their help, and Stuart Anderson, Terry Stroup, Colin Stirling and Matthew Morley of Edinburgh, for their comments. The comments of the anonymous referees were also helpful. This work was supported by SERC grant "Mathematically-Proven Safety Systems", IED SE/1224.

References

1. J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3, 1991.
2. J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. Technical Report CS-R8503, CWI, Amsterdam, 1985.
3. Glenn Bruns. Verifying properties of large systems by abstraction. To be submitted for publication, 1991.
4. Juanito Camilleri. A conditional operator for CCS. In *Proceedings of CONCUR '91*. Springer Verlag, 1991.
5. K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison Wesley, 1988.
6. Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. Technical Report ECS-LFCS-89-83, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.
7. A.H. Cribbens. Solid-state interlocking (SSI): an integrated electronic signalling system for mainline railways. *IEE Proceedings*, 134(3), May 1987.
8. Nissim Francez. *Fairness*. Springer-Verlag, 1986.
9. Sören Hölmstrom. A refinement calculus for specifications in hennessy-milner logic with recursion. *Formal Aspects of Computing*, 1:242–272, 1989.
10. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
11. Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, 1988.
12. Kim Guldstrand Larsen. Modal specifications. Technical Report 89-9, Institute for Electronic Systems, Department of Mathematics and Computer Science, Denmark, 1989.
13. F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of CONCUR '90*. Springer-Verlag, 1990.
14. Bernhard Steffen. Characteristic formulae for CCS with divergence. Technical Report ECS-LFCS-89-76, Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.
15. Colin Stirling. An introduction to modal and temporal logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*. Springer Verlag, 1989. Lecture Notes in Computer Science, volume 391.
16. D. J. Walker. Bisimulations and divergence. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, 1988.