Chasing Events to Certify a Critical System

Irfan Zakiuddin, Nick Moffat, Colin O’Halloran, Peter Ryan
{i.zakiuddin,n.moffat,c.ohalloran,p.ryan}@eris.dera.gov.uk
DERA Malvern

Abstract

This paper describes a verification, for safe behaviour, of a Command and Control System, performed by the authors for a customer within the United Kingdom Ministry of Defence. The entire system consists of eight sub-systems which interact by exchanging messages. The system progresses through a number of critical states, finally performing a critical operation. The system is notable for its size and complexity, consisting of approximately 150 messages. The messages represent assertions of states; however, states may also be retracted and faults may occur. This work was undertaken as part of a military procurement project, not a research project, and it was completed to project budgets and deadlines.

1. Introduction

The System Assurance Group (SAG) at the United Kingdom Defence Evaluation and Research Agency (DERA) develops and applies leading edge technology for the assessment of critical systems. Frequently, the goal of an assessment is to assist and advise a customer concerning acceptance of a procured system. The UK Ministry of Defence is the largest source of business for the SAG, but the SAG increasingly finds itself applying its expertise and technology for non-exchequer customers.

The process algebra CSP [1,2] and the model-checker FDR [3] form key elements in the SAG’s repertoire. The SAG has strong links with industry and academia, in particular Formal Systems (Europe) Limited (who develop and market FDR) and the Programming Research Group at Oxford.

1.1 Purpose of the Work

The project described in this paper concerns the integration of an additional sub-system (to provide further functionality) into a legacy military Command and Control system. The original and the upgraded system are both safety-critical.

Because of certification obligations, the customer approached the SAG to perform a system level analysis of the system. It consisted of eight physically distinct but communicating subsystems. There was already sufficient information relating to the correctness and reliability of individual components (obtained by detailed
fault tree analyses). However, no formal analysis of the interactions between sub-
systems had been performed.

A system level hazard analysis had been performed for the customer before the
SAG became involved in the modelling task. The hazard analysis had identified a
number of hazards, which were subsequently grouped into a few hazard catego-
ries. The customer was involved throughout this process, and was regularly con-
sulted during the modelling task itself. Specific questions of interest were derived
from the hazard categories. These questions addressed the hazards and they were
carefully designed so that they could be conveniently formulated as CSP refine-
ment assertions.

From the point of view of formal analysis and model-checking the most signifi-
cant feature of this modelling task is the size and complexity of the system. In
view of this, SAG staff believed the system would not be tractable for model-
checking. The initial strategy was to model as much of the system as possible and
then, when the state space exceeds feasible limits, proceed by making abstractions.
However, the authors were delighted to discover that the system can be model-
checked quite easily, in its entirety; this is achieved using the partial order method
previously developed for modelling security protocols [6].

2. System Description

We describe the system in terms of the types of messages which pass through it
and the way these messages propagate through the system. There are two types of
message, positive messages and retraction messages. Dependencies between mes-
sages are expressed using the notion of precursor sets.

2.1 Positive Messages, Retractions and Precursors

A positive message is the assertion of a state or the issue of a command, e.g.
‘component B is on’, or ‘prepare C’. A retraction message is, in all cases, the neg-
ative counterpart to a positive message, e.g. ‘component A is off’. However, not
every positive message has a counterpart retraction. There are about 150 positive
messages in the system; of these only 20 have negative counterparts considered
relevant to the safety case.

For any message in the system there is the important notion of its set of precur-
sors. The precursors of a message are those other system messages which must
occur to enable the message itself to be transmitted. So each message will have a,
possibly empty, precursor set. There are two types of precursor set, a conjunctive
precursor set and a disjunctive precursor set.

If the precursor set of a message is conjunctive, then all messages in the precur-
sor set must occur to enable the message itself. If the precursor set is disjunctive,
then one (or more) of the messages in the precursor set is sufficient to enable the
message itself.

The notion of precursor applies both to positive messages and to retractions.
The actual model was more complicated than that presented in this paper. Here we
assume that the positive messages have only positive precursors and, similarly,
that retractions have only retraction precursors.
The system may be represented by two sets of firing rules, one set for the positive messages, and one set for the retractions. The general form of the firing rules is:

(precursor set, output message, flag),

where ‘flag’ represents whether the precursor set is conjunctive or disjunctive.

Figure 1 shows a schematic of the diagram that was the basis of our understanding of the system. The information in the diagram was not complete, since it only showed some of the positive firing rules present in the system, and none of the retractions. It was augmented by the other system documentation. The first few horizontal arrows are marked in the fashion found in the actual diagram. Each vertical arrow represents a sub-system, which is named at the top of the arrow. Time progresses down the figure.

The system was designed to be fail-safe. Faults needed to be modelled so that this property could be checked.

3. Basic Types in the Models

Two models of the system were created, a positive model and an integrated model. The positive model considers positive messages only and ignores the possibility of retractions. The integrated model considers both positive messages and retractions, in one model. Both models are given the capability to perform faults.

The modelling language used is CSP\(_M\) [4], the machine readable form of CSP. Appendix A gives a brief summary of this language. Both models build on a common collection of datatypes, representing the messages passed around the system and the firing rules.

First we define the type of the messages passed around the model; it is called MESSAGE. It has the form:
Sender.Receiver.Value

where Sender and Receiver are the names of sub-systems, and Value is the message content. The Sender and Receiver fields both have type Module:

\[
\text{datatype Module = MOD1 | MOD2 | MOD3 | MOD4 | MOD5} \\
\hspace{1cm} | \text{MOD6 | MOD7 | MOD8}
\]

And the Value type is a very long enumeration representing pieces of information such as: ‘A is on’, ‘X is in state 3’, etc. So we can say:

\[
\text{nametype MESSAGE = Module.Module.Value}
\]

From the MESSAGE type we can construct the type of the firing rules, which is a 3-tuple:

\[
\text{FIRINGTUPLE = Set(MESSAGE) x MESSAGE x Logic}
\]

where

\[
\text{datatype Logic = And | Or}
\]

The first component of FIRINGTUPLE is the set of precursors to the output message, the second field of FIRINGTUPLE. The Logic component represents whether the set of precursors is conjunctive or disjunctive. In fact, we use a very small sub-type of FIRINGTUPLE, which is the set of actual firing rules extracted from the system documentation; we call this set RULE.

4. The Positive Model

The first stage of the modelling work was to implement a model for positive messages only.

4.1 Overview of the Model

Initially, this model was not capable of exhibiting faulty behaviour. Each of the positive firing rules is translated into either the CONJUNCTIVE or the DISJUNCTIVE process, depending on the type of the rule. There is one basic channel which is used for all communications:

\[
\text{channel signal: MESSAGE}
\]

In the fault-free implementation the CONJUNCTIVE process looks out for messages from ‘sees’ on the ‘signal’ channel; having seen any such input it removes it from its collection of ‘sees’. Of course, if there is nothing left to ‘see’ then it can branch to perform the outputs, via the ENGAGE process, described later:

\[
\text{CONJUNCTIVE(sees,do) =} \\
\text{signal?s:sees -> CONJUNCTIVE(diff(sees,{s}),do)} \\
\text{[]} \\
\text{empty(sees) & ENGAGE(do,{}))}
\]

The other basic process is DISJUNCTIVE, where the process need only see one of its inputs before branching to output.
Both of these processes use the ENGAGE process. This repeatedly outputs the the ‘do’ message, while also letting messages from the ‘lets’ set occur once each. The ‘lets’ parameter allows the messages it contains to occur. If it did not exist, then when the DISJUNCTIVE process branches to issue its output, after having seen only one of a number of possible ‘sees’ messages, then the remaining ‘sees’ messages would be blocked in the entire model. This would not accurately model the actual system.

The fault-free positive model is created by composing all the firing rule processes in parallel, with communication on common events. Using the powerful functional programming language found in CSP, the firing rules were translated into processes and these were combined to give the full model. This task would have been very difficult without the functional programming support.

4.2 Verification of the Positive Model

The basic property verified for the fault-free positive model was correct ordering of particular messages. A subset of about 40 messages was identified by the customer and expected behaviour of the system was defined via orderings of these 40 distinguished messages.

The 40 distinguished messages are partially ordered according to expected sequences of occurrence. Each of the 40 distinguished messages is expected only to occur after some (or possibly none) of the other distinguished messages. Thus the situation is very similar to the scheme used to model the system itself. Each distinguished message has expected precursors; these are those distinguished messages which are expected to occur before the message occurs.

The distinction between precursors and expected precursors is as follows: precursors are derived from the design documentation and are used when generating the model, whereas expected precursors define expected behaviour (w.r.t. distinguished messages) which should emerge from the model.

In view of this specification of expected behaviour, the positive model is required to exhibit Schneider [5] authentication properties. A distinguished message should authenticate each of its expected precursors (for the distinguished messages, they are all conjunctive), in the sense that its occurrence should prove that each precursor has already occurred.

So, to check that event ‘b’ authenticates event ‘a’:
- prevent ‘a’ occurring, by placing the model in parallel with the STOP process and forcing communication on the ‘a’ events only;
- hide all events except ‘b’;
- check for a refinement of the STOP process in the traces model.
In CSP\textsubscript{M}, this authentication property can be expressed as the assertion:

\begin{equation}
\text{assert STOP } \left[ T= (\text{SYSTEM } \{a\} \text{ STOP}) \backslash \text{diff}(\text{Sigma}, \{b\}) \right]
\end{equation}

If FDR decides that the refinement check fails then FDR will exhibit a trace where ‘b’ will have occurred without a prior ‘a’.

A more general method is to create a specification process which captures the expected partial ordering of all the distinguished events, and then verify the model against this specification with a single refinement check.

5. Positive Model with Faults

A safety case requires a reasoned argument showing that identified hazards cannot occur unless a sufficiently incredible combination of system faults occurs. Since the results from the modelling formed an input to the safety case, faults had to be modelled.

5.1 Identifying Faults of Concern

For the modelling to be useful to a safety case, the following are required:

- the model is a valid representation of the system;
- questions asked of the model address issues of concern;
- correct conclusions are drawn from the results.

Accordingly, it is beneficial if the following are true:

1. potential faulty behaviour of the actual system is expressed so that faults are independent;
2. modelled faults correspond closely to actual faults.

The customer’s view of the system was that the firing rules correspond to separate aspects of system functionality. It is certainly reasonable to suppose that faulty behaviour of the physically separate components is independent (for the present case, at least). It is more difficult to argue that faults within the same physical component are independent. This was outside the scope of the SAG’s task in this case, but the customer was made aware that the safety case should address this issue (perhaps by hazard analyses of the system components).

5.2 Implementing Faults

In the positive model faults were implemented using the firing rules. There are two basic types of fault which were added to the positive model, they are component faults and reception faults. These are implemented by defining extra types and channels.

A reception fault is where a firing rule will act without regard for one of its triggering precursors. To implement it we first define a special type and channel. (The ‘@’ and ‘_’ notation are explained in Appendix A):

\begin{equation}
\text{RECEPTION} = \{(t,x) | t@((xs,_,_),) <- \text{RULE}, x <- xs \}
\end{equation}

channel reception\_fault : RECEPTION
The type RECEPTION is the set of ordered pairs where the first member of the ordered pair is a firing rule and the second is a member of the precursor set for that firing rule.

A component fault occurs when a conjunctive firing rule issues its output before having seen all of its inputs.

channel component_fault: RULE

To inject faults into the model, we extend the definitions of CONJUNCTIVE and DISJUNCTIVE. Each of these processes is now also parameterised by the firing rule that it implements; this is the ‘tuple’ parameter. In addition, the process CONJUNCTIVE is given the new parameter ‘imagined’, this stores those precursors which have been circumvented due to a fault and must still be permitted to occur in the ENGAGE process, to prevent artificial deadlocks. The ‘imagined’ parameter contains those precursor events which the firing rule faultily ‘believes’ it has seen.

CONJUNCTIVE(sees, do, tuple, imagined) =

empty(sees) & ENGAGE(do, imagined)
[]
signal?s:sees ->

CONJUNCTIVE(diff(sees, {s}), do, tuple, imagined)
[]
component_fault.tuple ->

ENGAGE(do, union(sees, imagined))
[]
reception_fault?(_, s): { (tuple, s) | s <- sees } ->

CONJUNCTIVE(diff(sees, {s}), do, tuple,

union(imagined, {s}))
[]
signal?s:imagined ->

CONJUNCTIVE(sees, do, tuple, diff(imagined, {s}))

In the DISJUNCTIVE process, the occurrence of either a component fault or a reception fault is sufficient to cause the firing rule to branch to issue its output.

DISJUNCTIVE(sees, do, tuple) =

signal?s:sees -> ENGAGE(do, diff(sees, {s}))
[]
component_fault.tuple -> ENGAGE(do, sees)
[]
reception_fault?(tuple, _) -> ENGAGE(do, sees)

The CONJUNCTIVE and DISJUNCTIVE processes illustrate how easily fault injection can be performed using CSP.

Each firing rule is now capable of performing two types of fault and the entire model is capable of performing more faults than there are firing rules. To verify the
system in the presence of a particular number of faults of specific types, it is necessary to constrain the model. Fault control is easily performed using CSP. The following process manages faults in our model:

\[
\text{ATMOSTnFAULTS}(n) = \\
\quad \text{if } (n>0) \text{ then } (\text{fault } \rightarrow \text{ATMOSTnFAULTS}(n-1)) \text{ else STOP}
\]

\(\text{ATMOSTnFAULTS}(n)\), allows up to \(n\) ‘fault’ events, this process is placed in parallel with the model, and ‘component\_fault’ and ‘reception\_fault’ are, in turn, renamed to ‘fault’ (again, the notation is explained in Appendix A):

\[
\text{count\_regulated}(m,n,\text{model}) = \\
\quad (\text{model} \\
\quad \quad [ f \leftrightarrow \text{fault} | f \leftarrow \{|\text{component\_fault}|\} ] \\
\quad \quad \text{ATMOSTnFAULTS}(m)) \\
\quad \quad [ f \leftrightarrow \text{fault} | f \leftarrow \{|\text{reception\_fault}|\} ] \\
\quad \quad \text{ATMOSTnFAULTS}(n)
\]

So ‘\text{count\_regulated}(m,n,\text{model})’ behaves like the ‘\text{model}’ process except that the number of component and reception faults are respectively restricted to ‘\(m\)’ and ‘\(n\)’.

### 6. The Integrated Model

The integrated model is motivated by a need to analyse interactions of positive messages and retractions. We present the integrated model with faults, rather than present a fault-free integrated model first. The implementation of faults in the integrated model is discussed briefly at the end of this section.

#### 6.1 Basic Concepts

In concept, the integrated model is quite different to the positive model. The positive model is based on an event based view of the system, whereas the integrated model is based on a state based view.

In the positive model, processes synchronise on events which model messages being sent and received by various sub-systems. The integrated model has no notion of the exchange of messages. Instead, for each message in the system, there is a \text{WIRE} process which records whether that message is high or low, and which is also capable of switching that message from high to low and vice versa.

#### 6.2 The WIRE Process

So the first thing to describe is the basic \text{WIRE} process; it is from this that the integrated model is built. For each message in the system there is a \text{WIRE}. A \text{WIRE} is always in one of two states, ON or OFF. It is always prepared to issue events on the ‘report’ channel which describe its current state but leave the \text{WIRE} otherwise unchanged. Furthermore, a ‘switch’ event may always occur, causing the \text{WIRE} to toggle its state. The initial state of all \text{WIRE}s is OFF.
The types and channels used in the WIRE process are as follows:

\[
\text{Switch} = \text{On} \mid \text{Off}
\]

\[
\begin{align*}
\text{channel report} & : \text{Switch.MESSAGE} \\
\text{channel switch} & : \text{Switch.MESSAGE} \\
\text{channel faulty} & : \text{Switch.MESSAGE}
\end{align*}
\]

\[
\text{WIRE}(x) =
\hspace{1em}
\text{let}
\hspace{1em}
\begin{align*}
\text{OFF(isfaulty)} & = \\
\text{report.Off.x} & \rightarrow \text{OFF(isfaulty)} \\
\text{not isfaulty} & \& \text{switch.On.x} \rightarrow \text{ON(isfaulty)} \\
\text{not isfaulty} & \& \text{faulty.On.x} \rightarrow \text{ON(true)}
\end{align*}
\]

\[
\begin{align*}
\text{ON(isfaulty)} & = \\
\text{report.On.x} & \rightarrow \text{ON(isfaulty)} \\
\text{not isfaulty} & \& \text{switch.Off.x} \rightarrow \text{OFF(isfaulty)} \\
\text{not isfaulty} & \& \text{faulty.Off.x} \rightarrow \text{OFF(true)}
\end{align*}
\]

within \text{OFF(false)}

The ‘faulty’ channel is used to represent the occurrence of faults, with the ‘isfaulty’ parameter recording whether or not a fault has occurred. Initially, a WIRE has ‘isfaulty’ false. The implementation of faults is discussed later.

6.3 Events Required for the State Changes

To understand how the model is built we must first understand how to cause the state changes to occur in the WIREs. This is achieved using CSP event renamings and event synchronisation. For each message in the system, there are four sets of messages associated with it; these may be grouped into pairs:

1. the first pair are those messages which affect a WIRE:
   (a) messages which are required to switch the WIRE from OFF to ON;
   (b) messages which are required to switch the WIRE from ON to OFF.

2. the second pair are those messages which the WIRE itself affects:
   (a) messages which the WIRE has a role in switching from OFF to ON;
   (b) messages which the WIRE has a role in switching from ON to OFF.

Messages of group 1 are ‘report’ events and messages of group 2 are ‘switch’ events. The changes in state for a given WIRE are achieved by renaming the events offered by the WIRE to the sets of events just mentioned. This needs to be done such that the state of the WIRE will change precisely when the system firing rules allow the status of the corresponding message to change.
6.4 State Changes, the Conjunctive Case

To understand how the state changes for the conjunctive case are implemented, consider a WIRE called x. The report.On.x events offered by that WIRE are renamed to the switch.On events of all the WIREs which x has a role in turning from OFF to ON (2(a) above). Thus, if z needs x and y to be ON, in order for it to go ON, then, because x has a role in turning z ON, report.On.x is renamed to switch.On.z. Similarly, report.On.y is also renamed to switch.On.z. WIRE(z) is initially in the OFF state, so it is offering switch.On.z; when x and y are both in their ON states the model is capable of performing the switch.On.z event, as a consequence of WIRE(x), WIRE(y) and WIRE(z) synchronising on the switch.On channel.

Turning WIREs which are in the ON state to OFF is done in a similar fashion, by synchronising on the switch.Off channel.

6.5 State Changes, the Disjunctive Case

For the more complicated case of the disjunctive RULEs we make use of an extra channel, the or_switch channel.

channel or_switch : Switch.MESSAGE.MESSAGE

For a RULE in the system of the form:

\{(x,y),z,Or\}

either x or y is sufficient to turn on z. In this case report.On.x of the WIRE process is renamed to or_switch.On.z.x and report.On.y is renamed to or_switch.On.z.y. The event switch.On.z is renamed to both or_switch.On.z.y and or_switch.On.z.x (as well as to itself). As a consequence, the renamed WIRE(z), in the OFF state, is willing to perform either the or_switch.On.z.y event or the or_switch.On.z.x event.

Now consider the case when WIRE(x) alone enters the ON state; it is capable of synchronising with WIRE(z), in the OFF state, on the event or_switch.On.z.x, causing WIRE(z) to go ON. Similarly, when WIRE(y) enters the ON state, it is able to synchronise with WIRE(z), in the OFF state, on the event or_switch.On.z.y, also causing WIRE(z) to go ON.

The final touch to implementing the disjunctive rules is to rename the or_switch events back to ordinary switch events, after putting all the process together to build the model. This is necessary because verification of the model is through the switch events. The renaming is that each set of events:

\{or_switch.On.a.x | x ← MESSAGE\}

is mapped to the single event switch.On.a. A similar renaming is applied to turn or_switch.Off events to switch.Off events.

6.6 Implementing Faults in the Integrated Model

Until a WIRE performs a fault event, it is always able to do so. Two types of fault are modelled: faulty.On events and faulty.Off events. These correspond to the
WIRE changing from OFF to ON (or vice versa) respectively, and then sticking there (only being able to report its state).

As was the case with the positive model, a fault manager was used to constrain the allowable faults in the integrated model.

7. Discussion of the Fault Injection

Since we have modelled the system in terms of firing rules, the notion of fault clearly depends upon:

1. the interpretation of firing rules;
2. the choice of the particular firing rules used to describe the system.

7.1 Interpretation of Firing Rules

The positive model and the integrated model interpret firing rules differently. In fact, the interpretation by the positive model can only be an approximation to the interpretation by the integrated model. In the positive model, once a message has been sent the assertion contained in that message is assumed to hold indefinitely. However, the integrated model has the capability for a message to be asserted, then to be retracted and for the consequences of that retraction to propagate.

Since the two models interpret firing rules differently, for the same firing rules the notion of fault and the faulty behaviours enabled may be different in the two models. In both models it is beneficial for the modelled faults to correspond to actual system faults, and clearly we prefer that they correspond to the same actual system faults.

The positive model is intended to behave like the integrated model in the absence of retractions. For this to be true in the presence of faults, the faults modelled in the positive model must be at least as significant as those in the integrated model in the absence of retractions.

It turns out that for some sets of firing rules, the faults modelled in the positive model are not at least as serious as those in the integrated model in the absence of retractions. Fortunately, for the faults that are allowed by the questions of interest to the safety case, the component faults in the positive model are, in fact, equivalent to wire faults in the integrated model.

7.2 Choice of Firing Rules

The same set of positive firing rules was used by the positive model and by the integrated model. For both models, even when the meaning of firing rules had been defined there remained a variety of ways of representing the system as firing rules. In the absence of faults the different ways were equivalent for our purposes. However, the equivalence was not preserved when faults were modelled. This is because modelled faults are defined in terms of firing rules. Representations of the same system by different sets of firing rules can mean that modelled faults represent subtly different actual faults.

As an example, consider the following conjunctive firing rule:

\[\{\{A, B\}, C, \text{And}\}\]
Recall that firing rules constrain the system independently. In the absence of faults the following two firing rules are together equivalent to the one above:

\[ (\{A\}, C, \text{And}) \]
\[ (\{B\}, C, \text{And}) \]

However, these different representations of a fault-free system exhibit different faulty behaviour, since modelled faults are defined in terms of firing rules. With the former representation, a single component fault can allow C to occur before both A and B occur, but this would require two faults to occur with the latter representation.

The sometimes surprising repercussions of a particular choice of representation are disquieting, particularly when analysing safety-related software! This serves to emphasise the need to consider very carefully the consequences of modelling decisions.

8. Partial Order Methods

Throughout this work, the models constructed have been extremely large, they have unconstrained state-spaces of over \(10^{45}\) states. Clearly, exhaustive exploration of such state spaces is entirely infeasible. The key to analysing such huge models lies in the unique partial order reduction operator provided by FDR, called \textit{chase}. Thanks to \textit{chase} and also to FDR’s powerful state exploration engine, FDR can verify non-trivial properties for the entire system in a few minutes on a low-end work station.

The aim of partial order reductions is to ignore uninteresting interleaving of behaviours from models of concurrent systems. \textit{chase} was invented to reduce the state space of the spy process in model-checking security protocols [5]. There, the spy is essentially an inference system, over a fixed finite set of facts. Initially, the spy knows a few of the facts and is ignorant of the rest. On hearing a communication between agents in the system, the spy learns that fact, moreover, it can combine this newly learned fact with others it already knows to deduce further facts. The key point concerns the deduction of a set of new facts.

A set of newly deduced facts may be derived through a variety of interleavings of deduction, to conclude the complete set. However, these multiple deduction paths are not relevant, what is relevant is the final set of new facts. \textit{chase} was invented to address this issue.

8.1 What Does \textit{chase} Do?

FDR explores the operational semantics of CSP processes. A CSP process is given operational semantics by being represented as a labelled transition system, of the sort illustrated in Figure 2. In this operational semantics, nodes are connected by labelled transitions, which represent the occurrence of CSP events.

CSP comes equipped with an operator to hide events in processes (see Appendix A for the CSP hiding operator). The consequence of hiding a set of events A, in a process P, is to render them invisible and unavailable to the environment; they occur solely at the discretion of P. In terms of the labelled transition system for P, the events from set A, before hiding, were ordinary visible labelled transitions.
After hiding, all events from set A become what are termed $\tau$-events; they are now indistinguishable from each other and they represent internal transitions in $P \setminus A$.

In a labelled transition system a stable node is one which cannot perform $\tau$-transitions. Conversely, an unstable node is one which may perform one or more $\tau$-events; an unstable node may also be able to perform visible events.

During state exploration, FDR traverses the transition system of the implementation. It explores all reachable nodes comparing them with the specification. Chase is applied at this stage and all it does is chase $\tau$-events!

When an unstable node is encountered then a single, arbitrary, $\tau$-event is chosen to reach the next node and all remaining events are ignored. Even if the unstable node may perform a number of $\tau$-events, just one of them is chosen. There are no guarantees as to how the choice over several $\tau$-events is resolved - one is chosen and the choice should be regarded as arbitrarily non-deterministic. This process is iterated until a stable node is reached, at which point FDR’s normal exhaustive mode of operation is resumed.

In the spy process for security protocols, the inference events are hidden and chase is applied. The result is that multiple inference paths to a new set of conclusions are eliminated, leaving a single (arbitrary) sequence of inferences, from a set of known premises to the set of conclusions.

### 8.2 Using chase

Whilst it is evident that applying chase to a CSP process can drastically reduce the state space explored by FDR, it is also evident that chase is not a ‘safe’ operator. That is to say applying chase does not, in general, preserve semantics (unlike the various other FDR compression operators). The question is: when is it safe to use chase?

As a simple rule of thumb, if the model is deterministic, then that is sufficient to imply it is safe to apply chase to it. The justification for the determinism condition (as found in [1]) is that $\tau$-events result in the internal resolution of non-determinism; if there is no non-determinism present then the choice of $\tau$-events is not important.

The determinism condition is sufficient rather than necessary. In view of the very significant benefits of applying chase, future DERA research will investigate broadening the applicability of the chase operator.

### 8.3 Why is chase safe here?

The argument for the applicability of chase to our models is based on the sufficient condition of determinism. Consider the following deterministic process:

```plaintext
POSITIVE_MODEL(HideSet) =
  let
  -- InitEnabled and InitDone represent the initial state
  -- of the system
  Init = Union({outs | r <- RULE, r == ((),outs,_)})
  InitEnabled = diff(Init,HideSet)
```
InitDone = inter(HideSet, Init)
-- closenext iterates next till a fixed point is
-- reached

closenext (news, occurred) =
  let
    Forced = union (inter (news, HideSet), occurred)
  within
    if news == next (news, occurred) then news
    else closenext (next (news, occurred), Forced)
-- next calculates the next set of enabled events,
-- from those that events that have just occurred and
-- those which had formerly occurred.

next (news, occurred) =
  diff (Union (\{ outs | (ins, outs, _) <- RULE, 
             ins <= union (occurred, news) \}, occurred)
-- MODEL performs the events, it does one enabled event
-- then goes on to calculate its next state

MODEL (dones, enabled) = signal?e: enabled ->
  MODEL (union (dones, \{e\}), closenext (\{e\}, dones))
within

MODEL (InitDone, InitEnabled)

This process is another way to implement the positive model, with events from the ‘HideSet’ parameter hidden.

Its operation is quite simple, the MODEL sub-process makes a deterministic choice from the ‘enabled’ parameter. This message is added to the ‘done’ parameter and it is used to compute the next value for the ‘enabled’ parameter. This computation uses the firing rules to find the ‘next’ set of enabled messages. However, the next set of enabled messages may include messages from HideSet. So the calculation of next enabled messages needs to be closed under the consequences of messages from HideSet occurring. Closure is achieved using the ‘closenext’ function and this function is guaranteed to terminate because we have a monotonic increase in the size of ‘news’, which is a subset of a fixed finite set.

This is a sequential style of coding which causes a number of difficulties. The impracticalities of this model and the reasons for the actual implementation are almost entirely analogous to the situation with the spy in security protocol modelling [1,6].

The key point about this process is that it is clearly deterministic. In particular, it is deterministic even when the HideSet parameter is non-empty. The determinism of POSITIVE_MODEL(HideSet) follows from the fact that it makes no use of the hiding operator or of non-deterministic choice. So the determinism of the positive model after hiding a set HideSet may be concluded from its equivalence to POSITIVE_MODEL(HideSet).
Further, the addition of faults to the positive model does not compromise determinism. To see this, consider the case where just one component fault may occur. This fault event will be chosen deterministically and its consequences, for the ‘enabled’ parameter, will also be computed functionally; so preserving determinism.

Finally, we argue that \textit{chase} is safe to apply in the integrated model. In the integrated model \textit{chase} is only applied to those messages which do not have negative counter parts. This means that the sub-model to which \textit{chase} is applied is equivalent to a sub-model of the positive model. The safety of \textit{chase} in the integrated model then follows from its safety in the positive model.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2}
\caption{Applying the \textit{chase} operator}
\end{figure}

9. Verification Results

Most of the questions of interest could be addressed by the positive model. The answers obtained were confirmed by the integrated model (that is, when only the positive firing rules were used). However, some questions explicitly referred to retraction events; these could only be addressed by the integrated model.
9.1 Unexpected Behaviour

The models were used initially to determine whether the fault-free system (as specified in the design documentation) exhibited only behaviour expected by the system experts. This task had a two-fold benefit: it tested the expectations of the experts against the system design; and it served to validate the models themselves.

In fact, analysis of the fault-free models did reveal some unexpected behaviour. For two particular pairs of events, model-checking revealed that each event can occur before the other even though they were expected to occur in a fixed order. The customer's system experts were informed, and they claimed that timing properties of the system would enforce the expected orderings. (Time had not been modelled, so the model pessimistically allowed either event of a pair to occur before the other.)

Further interrogation of the models revealed that, if the above unexpected behaviour is barred, the firing rules are sufficient to prevent a fault-free implementation of the system behaving unexpectedly.

9.2 Consequences of Faults

The models were also interrogated to discover which possible single faults in the new sub-system could cause particular events to occur unexpectedly. System faults of most concern to the customer were those possible in the new sub-system. It is, in principle, possible that a fault in the existing system was tolerated by that system or that the system was fail-safe for this fault, but that in the context of the new system design this same fault would manifest itself as a system failure. However, the customer judged this scenario unlikely, considering the fact that earlier changes to the system had not revealed similar problems.

Various hazard-derived questions addressed the possible consequences of particular faults occurring - could particular dangerous events occur subsequently? Analysis of each model revealed those modelled faults in the new component that could, individually, cause the dangerous event referred to in the question to occur.

9.3 Adding Timing Constraints

When interrogating the integrated model, an important class of properties to be verified was that the system’s interlocks were adequate. That is to say, if certain messages were retracted, then the interlocks would stop the system and the system would reach an inert state.

Requiring that the system could be made inert, consequent to a retraction, was equivalent to requiring that retractions could propagate through the system sufficiently quickly to arrest positive messages. The operation of a model-checker is inherently pessimistic; so FDR was always able to find a pathological sequence of positive messages where the retractions fail to ‘catch-up’. Therefore, our models were not able to demonstrate the absence of race conditions.

To prevent FDR finding race conditions that don’t exist in the real system, time constraints could be added to the integrated model. A discrete model of time can be implemented in CSP. It is likely to be feasible to model-check such a model.
Recall that in the integrated model chase is only applied to those events which are not retractable. It turns out that, the addition of time constraints across the model would enlarge the domain of applicability of chase to the entire integrated model. However, this option was not pursued, due to project deadlines and insufficient data.

When race conditions were eliminated in the integrated model, by giving retractions priority over positive messages, the model found that the system interlocks are effective. We were thus able to provide qualified answers to some hazard-derived questions concerning the operation of the system following abort commands.

10. Conclusions

The significance of this approach is that it is applicable to information systems built from COTS and legacy components of a realistic size. Each component can be treated as a black box and a model may be built from system documentation and elicitation from the users. This model will contain the behaviours of the actual system. This pessimistic approach is safe but it increases the state space significantly and reports anomalies which might not exist.

The solution to the state space has already been discussed. The problem of “false-positives” turned out not to be significant and allowed the team constructing a safety case to focus on particular potential weaknesses. Another significant aspect of the work was the ability to “inject faults” into the design of the system and to determine how robust it was. This allowed the SAG to confirm that the new introduced COTS component was not safety critical and therefore reduced the potential cost of certification. This new capability is very important in an era where the cost of certifying systems is becoming prohibitively expensive. This approach is most cost effective when the safety of the system is built in at the system level (for example through interlocks) rather than at the component level.

11. Acknowledgements

The authors would like to thank Michael Goldsmith, Jason Hulance and Bryan Scattergood of Formal Systems (Europe) Ltd. for many illuminating discussions.

From the Oxford University PRG thanks are due to Bill Roscoe, in particular for first suggesting the use of chase, and also to Sadie Creese.

12. Bibliography


Appendix A. The Language CSP

CSP\textsubscript{M} is the machine readable dialect of CSP, used to define both the specification and implementation of models verified by FDR. It is very close to pure CSP and it is embedded in a Gofer/Haskell like functional programming language, which augments its utility substantially. Below is a summary of syntax of CSP\textsubscript{M}.

channel channel\_id : ChannelType is a CSP\textsubscript{M} declaration of a channel, the name of the channel is channel\_id and it communicates values of type ChannelType.

STOP is the simplest CSP process; it never engages in any action, and never terminates.

a \rightarrow P is the most basic program constructor. It waits to perform the event a and after this has occurred subsequently behaves as process P. The same notation is used for outputs (c!v \rightarrow P) and inputs (c?v \rightarrow P(v)) of values along named channels.

P |~| Q represents the non-deterministic or internal choice between P and Q.

P [] Q represents the external or deterministic choice between P and Q. It will, at first, offer the initial actions of both P and Q to its environment; its subsequent behaviour is like P if the initial action chosen was possible only for P and like Q if the action selected Q. If both P and Q have common initial actions, its subsequent behaviour is non-deterministic (like |~|).

STOP [] P is the same as P.

P | [A] | Q represents parallel (concurrent) composition. P and Q evolve separately, except that events from A occur only when P and Q agree (i.e. synchronise) to perform them.

P ||| Q represents the interleaved parallel composition. P and Q evolve separately and do not synchronise on any of their events.

P \ A is the CSP hiding operator. This process behaves as P except that events in set A are hidden from the environment and are solely (and internally) determined by P; the environment can neither observe nor influence them.

P [[a <- b]] represents the process P with a renamed to b.

P [ a <-> b | b <- bs ] Q Linked parallel. Processes P and Q are put in parallel. a and all the members of bs are renamed to a unique identifier.
‘mid’ in both P and Q, the renamed processes are placed in parallel and all
‘mid’ events are hidden.

\texttt{term@pattern} The \texttt{@@} symbol is used for pattern matching. \texttt{term} is of the
form of \texttt{pattern}. If \texttt{pattern} contains a \texttt{’_’} in place of an identifier,
then this signifies a ‘hole’, where binding of value to variable is not
required.

\texttt{A \leq B} The subset predicate, true if and only if A is a subset of B.

\texttt{inter(A,B)} Set intersection. This returns the intersection of sets A and B

\texttt{diff(A,B)} Set difference. This returns the intersection of sets A and B

\texttt{Union\{a | a \in A\}} Replicated set union. This return the set union of all
the sets in set A.

\texttt{assert SPEC \[T= IMP} The assertion of a refinement relation in the traces
model. The traces of IMP are a subset of those of SPEC.