Challenging Formal Specifications by Mutation: a CSP security example

Thitima Srivatanakul*, John A. Clark, Susan Stepney, and Fiona Polack

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
[jill,jac,susan,fiona]@cs.york.ac.uk

Abstract. When formal modelling is done we must validate both the model and the assumptions. Formal techniques tend to concentrate on the former. We examine how fault injection (specification mutation) and model checking can help address the latter, in particular, the effects of failure. We find that, in contrast with software testing, "equivalent mutants" are valuable for specification validation.

Keywords: CSP, model checking, mutation testing, security properties

1 Introduction

High integrity systems have particular properties that must hold. These typically pertain to their safe or secure operation. Formal development notations, such as Z, B and CSP, are used for high integrity systems because they introduce the possibility of: precise specification of requirements; formal (or highly rigorous) proof that a specification has particular properties; that a more detailed design refines the specification.

We must always remember that formal system descriptions are *abstractions*. They carry contextual *assumptions* about the environment, for example, that users behave as expected, that the consequences of allowed system behaviour are fully understood and are acceptable, that demands will not exceed particular thresholds, and that the physical execution environment operates perfectly.

So when formal modelling is done we must both *validate the model* – have we got the formal requirements we want? – and *validate the assumptions* – are we happy with the assumptions we know we have made? are there further implicit assumptions we need to expose?

Formal techniques tend to concentrate on validating the model. We need further ways of validating the assumptions. In this paper we examine how fault injection (specification mutation) and model checking can help address the above issues of specification validation and the effects of failure.

* Thitima Srivatanakul is funded by the Royal Thai Government.

1.1 Validating assumptions

It is easy, when specifying systems, to get unintended properties. For example, the Bell and La Padula security policy model [2] inadvertently allows "covert channels" that break the *intent* of the policy. This is "fixed" by [6]'s non-interference model. Yet this model itself (justifiably) abstracts from physical implementation details, and so allows information leakage via aspects such as power consumption [8] or electromagnetic radiation [1]. Most formal models assume operations are atomic, abstracting away from any information leakage by data dependent execution times. Also, the physical execution environment may be the cause of less subtle failures. For example, components may overheat or wear out. Such failures are refinement failures arising from the way the physical world operates.

It would be infeasible or impossible to model all such assumptions and failures formally. So, although we can specify precisely, and formally refine to an implementation, there is no guarantee that we have an appropriate system.

In safety, we often employ a defensive design philosophy: "X should not happen, but if it does, then ...". If we believe that the consequences of X occurring are worryingly high, then we may take additional steps to ensure X does not occur (and so protect against failure of the original means of prevention). In security, we additionally need to protect against maliciously induced failures. Our system may be acceptably secure whilst our assumptions hold, but we need to consider what happens if they do not, and what the likelihood of this is. In practice we need to consider the operation of the system under normal conditions and also under various failure conditions.

1.2 Mutation testing

Mutation testing [5] is the best known software fault injection technique. It is used to assess how thoroughly a test set exercises a program. Small syntactic variants of programs are derived; these *mutants* differ from the original source program in a very small way, for example, a single + operator may be replaced with a −. A test set's *mutation score* is the proportion of mutants it "kills" (detects). The closer the score is to 1, the more thoroughly the test set is deemed to have exercised the program.

In practice a small number of mutants have identical behaviour to the original on all input vectors. These are termed *equivalent mutants*. The mutation score can be redefined as the proportion of non-equivalent mutants killed. However, determining whether a mutant really is equivalent, or whether the current test set is merely insufficiently powerful to kill it, requires considerable manual effort. Indeed, determining the equivalence or otherwise of mutants is a major expense in mutation testing. Equivalent mutants are "a problem".

As well as mutating the program, data state mutation is used to determine testability. [21] describes some further non-standard applications, such as safety and security assessment (assessing the robustness of code to failures).

1.3 Specification mutation and its uses

Specification mutation was first attempted by Budd [4]. More recently [3] applies mutation to SMV process descriptions. However, the general area is little researched. It is clear what code mutation testing aims to achieve, but it is less clear to what uses specification mutation can be put. We illustrate some uses of specification mutation with reference to our case study, written in CSP.

In general, we have a property P, a specification that abstracts away from the functionality of the system, and captures just the required properties of the system. We also have a system specification, S, which defines the functional behaviour of the system. In order to demonstrate the integrity of the system, it must be shown that S *satisfies* P. (In our particular case study, it is sufficient to show that S is a refinement of P.) We now mutate S, to obtain S'.

S and S' may both satisfy P, yet exhibit different behaviours. The system specification S is more than just a refinement of the P properties; it also expresses the functional properties of the system. S is in effect a refinement of a number of abstract requirements, explicit ones concerned with security, and implicit ones concerned with functionality. If the behaviours of S and S' can be distinguished in any way then the designers can be challenged to say which behaviour they really want. They may decide that the mutant S' better captures the requirements, or they may choose to stick with the original S. Thus, mutation can serve as a *requirements validation* tool.

Specification mutants can model failures in development. We may have specified S, but the implementation is actually a refinement of the mutant S'. If S and S' are not equivalent, then distinguishing behaviours can form the basis of test scenarios (that is, ways to exercise the implemented system to determine which of S and S' it implements). Furthermore, since the number of mutants may be very high, we may choose to concentrate our efforts on cases where S' does not satisfy P, that is, we can direct out testing towards security critical failures of development.

Specification mutants can represent hypothetical failures arising due to failures of the execution environment, for example, due to equipment failure.

Specification mutants can also represent hypothetical failures engineered from malice by an attacker (in whatever way). If S' breaks P then we must direct attention to ensuring that a implementation that refines S' cannot be engineered by an attack on an implementation of S.

The above discussion provides sufficient initial motivation for investigating mutant specifications. However, analysing a system is hard. Investigation of a myriad of mutant systems requires automation. Our chosen approach uses model checking, as outlined below.

1.4 Model checking

Model checking challenges a specification by exploring the whole state space. Model checking tools are becoming common in some areas of high integrity systems development. There have been a number of security applications. [13]

uses FDR to analyse a system model for non-interference through determinism. Security protocols are a successful application domain (for example, [14, 16, 15]). [9] pointed out flaws in the Needham-Schroeder public key protocol [11] a full seventeen years after its publication.

Model checking is a highly automated decision procedure. A specification can be checked for certain properties, such as deadlock, determinism, or being a refinement of another specification. The result of the check, assuming that the algorithm runs to completion, is either *yes* (the property holds) or *no*. Model checking is highly automated, and the process cost-effective relative to other (manual, or less exhaustive) analyses. Our desire to analyse very many different mutants requires such automation.

We use FDR2 [12, 10], a CSP model checker, for our case study. We use it to check whether the mutant S's remain refinements of P, in terms of traces, failures, and failures/divergences, refinement. Trace refinement ensures that the more concrete refined specification possesses only behaviours that are possible for the abstract specification. Failures refinement additionally ensures the concrete deadlocks less than the abstract. Failures/divergences refinement additionally ensures the concrete diverges (livelocks) less than the abstract.

1.5 The mutation tool

Our mutation tool is an extension of that described in [17]. It operates on a CSP specification expressed in FDR2 syntax [10]. The process of using the tool is:

1. Parse the CSP/FDR specification S, to recognise linguistic elements, for example, events and process names.
2. Inject mutants (faults) into S, according to a user-selected subset of the predefined *mutation operators* (modification rules).
3. Output the mutated specifications S', in FDR2-format.
4. Pass the mutated specifications S' to FDR, to test them against the unmutated specification P.

The mutation operators are carefully chosen to avoid generating mutants that are syntactically and (to some extent) type incorrect. For example, the CSP operators $\|$ (parallel) and $|||$ (interleaving) are operators on processes; it would be incorrect to apply them to events or communications. The tool generates mutants on processes, events, expressions and structures, according to the CSP mutation operators (appendix A).

At present, all the mutants are point mutations, replicating the kinds of typographic errors made in writing formal specifications. We have not yet considered systematic changes of the sort introduced by hasty search-and-replace editing.

2 The case study

2.1 Overview

Two of us worked on the formal development (SS) and evaluation (JAC) of an electronic purse [18, 19]. Here we adapt the specification for our case study.

The system consists of a number of Smartcard-based electronic purses that carry financial value. The purses interact with each other, via a communication device, to exchange value. All the security measures are implemented on the cards, with no real-time external audit logging or monitoring. These cards are used by members of the public to carry out fully electronic financial transactions with other individuals, banks, retailers, etc. It is critical that the system can neither forge value, nor lose value.

A formal *security policy model* captures the required security properties of the system, in terms of the exchange of value between pairs of purses. The key operation transfers value from one purse to another, modelled as an atomic action that decrements the value in the 'from' purse and increments the value in the 'to' purse. The model also allows a failed transaction, where the value is 'lost' (not received by the 'to' purse), modelled by having the 'from' purse decrement the value from its balance, and add it to a special *lost* component. (The challenge for the implementation is then to implement both these cases correctly.) The two key system security properties are (1) no value may be created in the system (2) all value is accounted in the system (no value is lost *from the system* that is; it may be unavailable for transfer).

The system defined here is based on [19]. The original formal development uses Z. Here, we recast the (key features of) the purse security policy specification in terms of its concurrent properties, using CSP. The system security properties are specified as the *Secure* process, and the abstract atomic transfers as the *Purse* process.

2.2 The CSP specification

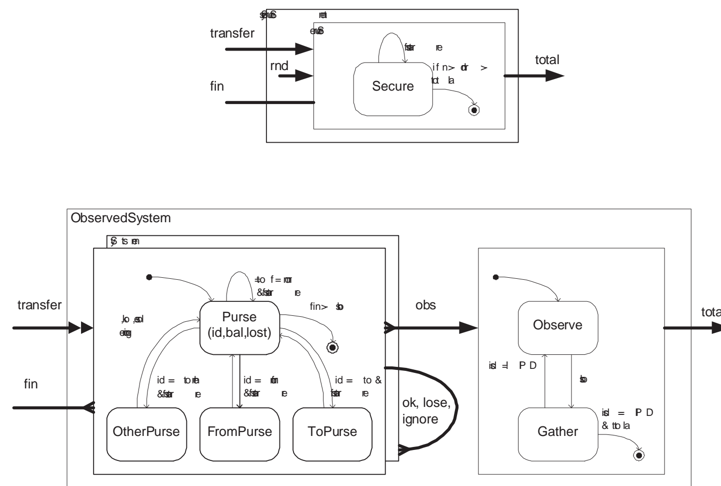

Fig. 1. Summary of the CSP specification. Sequential processes are shown as state transition diagrams; parallel communication channels are shown as bold arrows.

First we introduce some useful constants.

$Npurse = 4$	[number of purses (variable)]
$InitBal = 1$	[initial balance in each purse (variable)]
$TotalBal = Npurse * InitBal$	[hence, total system balance]
$PID = 0 \mathinner{\ldotp\ldotp} (Npurse - 1)$	[purse identifiers]
$VALUE = 0 \mathinner{\ldotp\ldotp} TotalBal$	[value transferred between purses]

Channel *fin* signals the system to finalise. Channel *transfer* identifies the transfer action, labeled by from purse, to purse, and transfer value. Channel *total* outputs the total value stored and lost in the system, on finalisation. The internal channel *rnd* is used to generate a random value that is considered lost.

$$fin;\ transfer : PID.PID.VALUE;\ total : VALUE.VALUE$$
$$rnd : VALUE$$

A system is *Secure* if, on finalisation, the total value stored is no greater than the initial balance ('no value created'), and if the values stored and lost together equal the initial balance ('all value accounted').

$$\begin{array}{l} Secure = \\ \quad transfer?from.to.val \rightarrow Secure \\ \quad \Box\ fin \rightarrow rnd?v \rightarrow total!(TotalBal - v).v \rightarrow Skip \end{array}$$

$$SecureSystem = Secure \setminus \{| rnd |\}$$

In the purse system specification, the channel *obs* is used to observe the *bal*ance and *lost* components of each purse on finalisation. The channels *ok*, *lose*, and *ignore* synchronise the purses on the particular choice of transfer operation.

$$obs : PID.VALUE.VALUE;\ ok;\ lose;\ ignore$$

When a *Purse* engages in a *fin* event, it outputs its state, then terminates. When it engages in a *transfer* event, it behaves like a *FromPurse*, a *ToPurse*, or an *OtherPurse*, depending on its *id*.

$$\begin{array}{l} Purse(id, bal, lost) = \\ \quad fin \rightarrow obs!id.bal.lost \rightarrow Skip \\ \quad \Box\ transfer?from.to.val \rightarrow \\ \quad\quad \text{if } to = from \text{ then } Purse(id, bal, lost) \\ \quad\quad \text{else if } (id = from) \text{ then } FromPurse(id, bal, lost, val) \\ \quad\quad \text{else if } (id = to) \text{ then } ToPurse(id, bal, lost, val) \\ \quad\quad \text{else } OtherPurse(id, bal, lost) \end{array}$$

A *FromPurse*, if it has sufficient balance for the requested transfer, non-deterministically chooses to do an *ok* transfer (decrementing its balance), or to *lose*

the transfer (decrementing its balance and incrementing its *lost*), or to *ignore* the transfer (leaving its state unchanged). A *ToPurse* increments its balance for an *ok* transfer, otherwise does nothing. An *OtherPurse* does nothing.

$$
\begin{array}{l}
\mathit{FromPurse}(id, bal, lost, val) = \\
\quad \text{if } val \leq bal \text{ then } (\\
\quad\quad ok \rightarrow Purse(id, bal - val, lost) \\
\quad\quad \sqcap\ lose \rightarrow Purse(id, bal - val, lost + val) \\
\quad\quad \sqcap\ ignore \rightarrow Purse(id, bal, lost) \\
\quad) \text{ else } (ignore \rightarrow Purse(id, bal, lost))
\end{array}
$$

process

$$
\begin{array}{l}
\mathit{ToPurse}(id, bal, lost, val) = \\
\quad ok \rightarrow Purse(id, bal + val, lost) \\
\quad \Box\ lose \rightarrow Purse(id, bal, lost) \\
\quad \Box\ ignore \rightarrow Purse(id, bal, lost)
\end{array}
$$

process

$$
\begin{array}{l}
\mathit{OtherPurse}(id, bal, lost) = \\
\quad ok \rightarrow Purse(id, bal, lost) \\
\quad \Box\ lose \rightarrow Purse(id, bal, lost) \\
\quad \Box\ ignore \rightarrow Purse(id, bal, lost)
\end{array}
$$

A purse *System* runs the purses in synchronised parallel, with each purse initially having a zero *lost* component.

$$
System = \|[\{\!|ok, lose, ignore, fin, transfer|\!\}]\|\ id : PID \bullet Purse(id, InitBal, 0)
$$

The processes *Observe* and *Gather* collect and output the total value of all the purses' balances and lost components on finalisation.

$$
\begin{array}{l}
Observe(tb, tl, ids) = \\
\quad obs?id.bal.lost \rightarrow Gather(tb + bal, tl + lost, ids \cup \{id\}) \\
Gather(tb, tl, ids) = \\
\quad \text{if } ids = PID \text{ then } total!tb.tl \rightarrow Skip \text{ else } Observe(tb, tl, ids)
\end{array}
$$

The *ObservedSystem* is the purse *System* and the initial *Observe* process, with the internal channels hidden.

$$
\begin{array}{l}
ObservedSystem = \\
\quad (System \ [\![\ \{\!|obs|\!\}\ ]\!]\ Observe(0, 0, \emptyset)) \setminus \{\!|obs, ok, lost, ignore|\!\}
\end{array}
$$

The *ObservedSystem* (corresponding to the specification S) is a *SecureSystem* (has the property P): it does not create value, and accounts for all lost value.

assert

$$
SecureSystem \sqsubseteq_{FD} ObservedSystem
$$

2.3 The FDR2 specification

Using the FDR2 model checker, we have confirmed that the *ObservedSystem* is a refinement of the *SecureSystem*.

We added an extra guard, $lost + val \leq TotalBal$, to the *ok* choice of the *ToPurse*, to limit the state space FDR generates. Such states are unreachable in a secure system. (Adding this guard does not preclude the possibility of the system violating its security property, as this guard is a condition on a single purse, and the security property is a condition on the entire system.)

3 Applying the mutants

3.1 The Mutation Set

Mutant Types	**No. of Mutants Generated**
Process Definition Modification Operators	289
Expression Modification Operators	125
Structure Modification Operators	16
Parameter Modification Operators	167
Total	597

Table 1. Overview on Number of Mutants Generated for Each Type

The mutation operators are identified in appendix A. Our specification has 10 process descriptions: 597 mutants were created. These mutated systems were analysed for three separate cases: $Npurse = 2 \wedge InitBal = 1$, $Npurse = 3 \wedge InitBal = 1$, and $Npurse = 4 \wedge InitBal = 2$. Table 1 summarises the total number of generated mutants categorised by mutation type; tables 2–5 show samples of the generated mutants from each type.

The larger the specification, the larger the number of mutants created. The need for the mutation generation tool is essential. With the tool, it only took only a few seconds to generate all the mutants for the system. The choice of mutation operators highly affects the number of mutations generated. It is straightforward to produce *syntactically* correct mutants. However, when FDR2 compiles some specifications, it goes into an endless loop; it is important not to produce too many mutants that cause this behaviour, as it becomes very time consuming and costly to execute. (However, it is sometimes inevitable that such mutants are produced, as we can see in detail later.)

3.2 The Results

We first check whether each mutant *ObservedSystem* is still a Failures-Divergence refinement of the *SecureSystem*. If so, it is said to be *equivalent*: it possesses the same properties as the original specification. Mutants that are not FD-refinements may violate only FD-refinement, or may also violate trace refinement, or may result in FDR compilation errors.

Tables 6–9 show the result of the mutation execution. There are a total of 597 mutants, of which 23 are equivalent mutants. The non-equivalent mutants

PED1	*Purse*	drop the event *fin*
PER1	*FromPurse*	replace the event *ok* with *ignore*
PER2	*FromPurse*	replace the first event *ignore* with *lose*
PER3	*FromPurse*	replace the event *lose* with event *ignore*
PEI1	*Purse*	insert an extra event *fin*
PEI2	*FromPurse*	insert an extra event *ok*
PCO1	*FromPurse*	replaced the first internal choice $\sqcap$ by external choice $\square$
PCO2	*ToPurse*	replace the first external choice $\square$ by internal choice $\sqcap$
PPO1	*System*	replace the parallel operator $\parallel$ by interleaving $\vert\vert\vert$
PMR1	*Secure*	replace the channel *transfer* message *from.to.val* by the channel *rnd* message *v*
PMR2	*Purse*	replace the channel *obs* message *id.bal.lost* by the channel *transfer* message *from.to.val*
PCI1	*Secure*	insert an input channel event $rnd?v$
PCI2	*Purse*	insert an output channel event $obs!id.bal.lost$
PCR1	*Secure*	replace the channel *transfer* with channel *total*
PCE1	*Purse*	drop the event *obs*
PPR1	*FromPurse*	replace $Purse(id, bal - val, lost + val)$ with $Purse(id, bal, lost)$
PPR2	*Purse*	replace $Purse(id, bal, lost)$ with $OtherPurse(id, bal, lost)$

Table 2. Mutants on Process Definitions. (See appendix A for definition of the three letter codes)

ELR1	The logical operator $\wedge$ from *Observe* is replaced with $\vee$
EUR1	negate *bal* in the expression $val \leq bal$ in *FromPurse*
EUR2	negate *val* in the (additional FDR) expression $val + bal \leq TotalBal$ in *ToPurse*
EAK1	*val* in the expression $val \leq bal$ is increased by 1
EAK2	*to* in the expression $id = to$ in *Purse* is increased by 1
ESK1	*bal* in the expression $val \leq bal$ is decreased by 1
EAD1	*val* in the (additional FDR) expression $lst + val \leq TotalBal$ is replaced with 0
ERR1	The relational operator $=$ in $to = from$ in *Purse* is replaced with $\geq$
ERR2	The relational operator $=$ in $id = to$ in *Purse* is replaced with $\neq$
ERR3	The relational operator $=$ in $ids = PID$ in *Purse* is replaced with $\geq$
ERD1	The value *to* in the expression $id = to$ in *Purse* is replaced with 0
ERD2	The value *bal* in the expression $val \leq bal$ is replaced with 0

Table 3. Mutants on Expressions

SCR1	The (additional FDR) expression $val + bal \leq TotalBal$ in *ToPurse* is replaced with *false*
SSR1	The expression $to = from$ in Purse is replaced with *true*
SSR2	The expression $id = to$ in Purse is replaced with *true*

Table 4. Mutants on Structures

PRR1	The value val of $Purse(id, bal - val, lost)$ is changed to 0 in $FromPurse$
PRR2	id of $ok \rightarrow Purse(id, bal + val, lost)$ is changed to 0 in $ToPurse$
PRR3	The value val of $FromPurse(id, bal, lst, val)$ is changed to 0 in $Purse$
PAK1	The value val in $ToPurse(id, bal, lost, val)$ from the process description $Purse$ is incremented by 1
PAK2	The initial value of lost (0) of each $Purse(id, InitBal, 0)$ is changed to 1
PSK1	The parameter $InitBal$ in $Purse(id, InitBal, 0)$ in $System$ is reduced by 1

Table 5. Mutants on Process Parameters

Operator	Total	Equiv	Non-Equiv		
			FD	T	Error
PED	2	0	0	2	0
PER	60	3	28	29	0
PES	0	0	0	0	0
PEI	12	0	10	2	0
PCO	8	3	5	0	0
PPO	6	0	2	3	1
PMR	8	1	0	0	7
PCR	8	0	0	2	6
PCI	7	1	2	4	0
PCE	3	0	1	1	1
PCS	0	0	0	0	0
PPR	175	2	57	38	78
Total	**289**	**10**	**105**	**81**	**93**

Table 6. Results of mutation execution for Process Definition Modification Operator

Operator	Total	Equiv	Non-Equiv		
			FD	T	Error
ENI	0	0	0	0	0
ELR	1	0	0	0	1
ELD	4	0	2	0	2
EUR	14	1	1	6	6
EAR	12	0	0	0	12
EAK	14	1	5	6	2
ESK	14	1	1	6	6
EAD	8	0	0	0	8
ERR	45	6	14	15	10
ERD	13	1	6	6	0
Total	**125**	**10**	**29**	**39**	**47**

Table 7. Results of mutation execution for Expression Modification Operator

Operator	Total	Equiv	Non-Equiv		
			FD	T	Error
SSR	10	2	3	5	0
SCR	6	1	3	0	2
Total	**16**	**3**	**6**	**5**	**2**

Table 8. Results of mutation execution for Structure Modification Operator

Operator	Total	Equiv	Non-Equiv		
			FD	T	Error
PRR	63	0	16	44	3
PAK	52	0	0	7	45
PSK	52	0	0	1	51
Total	**167**	**0**	**16**	**52**	**99**

Table 9. Results of mutation execution for Parameter Modification Operator

comprise: 156 Trace Refinement but Failure-Divergence (FD) Refinement violations; 177 Trace Refinement (T) Violations; and 241 FDR Compilation Errors. It should be noted that some particular mutation operators (e.g. EAR, PAK and PSK), when applied to our specification, produce an S' that the FDR2 tool fails to compile. Table 10 shows in detail the results of execution for each mutant in the samples specification.

Types	Equivalent	Non-equivalent Mutant		
		T	FD	Error
Process Definition Modification Operator	PER2, PER3, PCO1, PMR1, PCI1, PPR1, PPR2	PED1, PER1, PEI1, PPO1, PCI2, PCR1	PEI2, PCO2, PCR1, PCE1	PMR2
Expression Modification Operator	EUR1, EAK1, ESK1, ERR1, ERR2*, ERR3, ERD2	EAK2, ERD1, ERR2*		ELR1, EUR2, EAD1
Structures Modification Operator	SSR1	SSR2	SCR1	
Parameter Modification Operator		PRR1, PAK1, PAK2, PSK1	PRR2, PRR3	

Table 10. Results of the sample mutants
* ERR2 is an equivalent mutant only for a system with 3 purses

3.3 Equivalent mutants

Out of the 23 equivalent mutants generated, 20 mutants are distinct from each other, that is, have different behaviours. The other three produce the essentially same specification as each other; for example EUR1 (negating the balance) and ERD2 (replacing *bal* with 0) both result in $val \leq bal$ always being *false* (when $bal > 0$ and $val > 0$).

The followings summarise behaviours of some important equivalent mutants from the samples.

1. System engages only in *ignore*. EUR1 changes the conditional expression, $val \leq bal$, of process *FromPurse* to $val \leq -bal$. If all purses have an initial balance of more than 0, then the mutant causes the expression to evaluate to *false* always, because *val* cannot be a negative value. As a result, the *FromPurse* can engage only in *ignore*, causing all the purses in the system to *ignore* all the transactions. There are no changes to the *bal*ance and *lost* of each purse, therefore the security properties of the system still hold. (The system can, however, engage in *ok* and transfer nothing, if at least one purse initialises its balance to 0.)

2. System idles. SSR1 shows that system cannot engage in any of the events.
3. The transfer value is limited. ESK1 (reduce *bal* by 1) and EAK1 (increase *val* by 1) strengthen the conditional expression, if $val \leq bal$ in process *FromPurse*. The mutant requires the transfer value to be strictly less than the balance, so prohibits the transfer of an amount equal to the balance. This does not violate any of the security properties; it simply places an additional constraint in the system.
4. Purse cannot make transfer to a 'higher' *id* purse. ERR1 replaces the operator $=$ in $to = from$ of *Purse* with $\geq$, resulting in a purse not able to transfer to a purse with a numerically higher *id*. This has no effect on the security properties; it simply places an additional constraint to the system.
5. Making the *FromPurse* deterministic. PCO1 changes internal choice $\sqcap$ to external choice $\Box$; the choice between events is externally determined.
6. Modification of *bal*ance and *lost* while engaging in *ignore*. In *FromPurse*, PER3 changes the event *lose* to *ignore*, whilst still modifying the *bal*ance and *lost* components. This has no effect on the security properties; the system is simply 'lying' about which event has occurred.
7. Ignoring *bal*ance and *lost* while engaging in *lose*. In *FromPurse*, PER2 changes the event *ignore* to *lose*, and PPR1 changes $P(id, bal{-}val, lost{+}val)$ to $P(id, bal, lost)$. This has also no effect on the security properties; the system is simply 'lying' about which event has occurred.
8. Transfer money to the 'other' purse. This is an interesting result. ERR2 replaces the relational operator in $id = to$ to become $id \neq to$. In the case with three purses running in the system, there is only one 'other' purse to act as the *ToPurse*, increasing its balance, while the intended 'to' purse does nothing. So the security property still holds in this case. (This does not remain the case for four purses, when there are two 'other' purses, both increasing their balance.)

Other equivalent mutants, such as PCI1, PMR1, ERR3, have no practical or meaningful interpretations. They are equivalent to the original specification, but they make the system more difficult to understand.

3.4 Non-equivalent mutants

Non-equivalent mutants can be categorised into (1) those that violate the security properties, where *ObservedSystem* does not refine on *SecureSystem* and (2) those that encounter a compilation error.

The followings are the invalid behaviours of the system, violating P.

1. Dropping and inserting observable events. PED1, PEI1, PCI2 drop or insert one of the observable events, making the process *SecureSystem* and *ObservedSystem* behave differently.
2. Modification of *bal*ance while engaging in other events. PER1 replaces *ok* with *ignore*, this allows *FromPurse* to decrement the *bal*ance while engaging in *ignore*, which means that no other purse increments the balance, so this violates the 'all value accounted for' property.

3. Value is modified arbitrarily. This is strictly prohibited in order to maintain security properties. PAK1 and PRR1 show that this cannot be invalidated.
4. The intended *ToPurse* cannot act as a *ToPurse*. ERD1 changes the *to* variable to a fixed number 0; only *Purse*(0) can have money transferred to it.
5. Another purse acts as a *ToPurse*. EAK2 changes the condition for becoming *ToPurse*, causing another purse to act as the *ToPurse*. However, if that other purse is the *FromPurse*, it behaves as a normal *FromPurse*, so its balance is decremented as usual, and there is no purse that acts as the *ToPurse* having its balance incremented.
6. Value transferred to (or from) more than one purse in a single transaction. ERR2 (when there are more than 3 purses in the system) and SSR2 allow more than one purse to increase the transferred balance.
7. Purses with the same *ID* exist in the system. PRR2 illustrates this behaviour. However, this too does not exhibit the security properties.
8. Initial values are inconsistent with the global value *TotalBal* assignments. PSK1 and PAK2 change the value of the initial balance and initial lost respectively. Obviously, added values cannot be summed up to *TotalBal*.

Other mutants that fail to refine on the FD model (for example, PEI2 and PCO2) have different constraints on the events that the process can engage in.

Mutants that fail to compile might result from exceeding the boundary values (for example, EUR2, ELR1) or type error (for example, PMR2, where the message *id.bal.lost* in channel *obs* is replaced by *from.to.val*). Further work on the tool will reduce the number of such mutants produced.

4 Analysis of results

4.1 Initial insights and revisions

Whilst the analysis of most mutants was straightforward, some required more thought. In particular, the mutant ERR2, considered above, which changes $id = to$ to $id \neq to$, was initially a refinement of the secure system, which surprised us. More thought revealed that this was due to the way the specification had been constrained to facilitate FDR analysis – to reduce the state space, there were only three purses in the system, so the mutant merely swapped the roles of the sole 'other' purse and the 'to' purse. We reran the tools on a system of four purses and the mutant ERR2 was no longer equivalent; both the 'other' purses are incremented, causing value to be created.

It is satisfying that an analysis can detect flaws in the way the system is set up for tool checking. (However, it is not clear that this self-analysis could be made a systematic feature of the approach.)

A side-effect of this finding is to alert the developer to the fact that realistic tests of the implemented system will also require at least one 'from' purse, one 'to' purse, and many 'other' purses.

Another satisfying insight from the initial analysis was the discovery of redundancy of some of the boolean guards introduced when the CSP specification

was rewritten in FDR2 notation (noted above). The need for all of the added guards (which are on unreachable states) had been questioned at the time of translation. Equivalent mutants demonstrated that some were indeed redundant, and these were removed before the analysis reported here. This finding generalises – equivalent mutants may indicate redundancy in the specification, or even attractive refactorings that could make the specification more readable, or perhaps more amenable to formal analysis (proof).

4.2 Failure: Trace violations

A significant number of the mutants that do not maintain the secure properties of the system were detected through trace violations. FDR2 provides counterexamples for these trace violations. There were essentially two forms of trace violation.

1. Incorrect Trace. Many mutants produce trace violations detected by discrepancies in the value of *TotalBal*. For example, mutations EAK2 and ERD1 tamper with the identity of the *to* purse in the conditional predicate of the *transfer*. This results in loss of value, breaking the security properties.

These violations emphasise the sensitivity of the system to variations in purse identity (in addition to the modification of *bal*ance and *lost*), and so further development and testing needs to focus on these issues.

2. Incorrect events on trace. Mutants that drop or insert an event inevitably cause the traces of the two specifications to differ. For example, PED1 dropped the Purse finalisation event, *fin*.

These violations are as expected, and merely improve confidence that the specification is a good refinement of the secure properties.

4.3 Failure: Failure-divergence violations

There are a number of mutants that cause failure-divergence violations without trace violations. They mainly happen when processes cannot agree on events.

In the purse processes, the *FromPurse* non-deterministically decides what events will subsequently occur. The other purses must act deterministically to synchronise the transfer. Changing one of the deterministic choices in the *ToPurse* (PCO2), for instance, to a non-deterministic choice, causes the system to deadlock when "wrong" processes attempt to synchronise.

Although this mutation results in deadlock, the analysis reveals that no security property has been breached, since no transfer takes place. In some circumstances, it might be acceptable to implement a purse system that allowed deadlock if security was enforced; here, however, the specification does not allow the system to deadlock. The analysis has prompted explicit recognition and discussion of this design decision.

Other mutants result in non-determinism-induced deadlock that may (or may not) break the security properties, depending on the ways in which the

non-determinsm resolves in each case. Mutant PRR2 sabotages the identity of the *ToPurse* in the clause, $ok \rightarrow Purse(id, bal + val, lst)$. This results in two purses acting as if they had the same identity; if both subsequently act as the *FromPurse*, they could then non-deterministically select different choices of action and deadlock the system during the transfer. The accumulated balance would depend on the order of finalisation of the two purses; violation of the secure properties would occur in some cases but not all. This form of mutant again emphasises the susceptibility of the specified system to mistakes in purse identification. However, it also shows the value of using an exhaustive approach such as FDR; this explores the whole state space, identifying violations that could be missed by selective trace analysis, or, later in development, by selective testing.

4.4 Refinement: changed functionality

There are a number of equivalent mutants (those which are refinements of P) that change the functional behaviour of the specified system S. The result is a system that does not deadlock, but that does not behave as anticipated (based on S). Examples include modification of the balance whilst engaged in *ignore* (PER3), and mutations that prevent *lose* from changing the *bal*ance and *lost* values (PER2, PPR1).

Analysis of these mutants helps to clarify what the functional properties of the system would be; in particular, it focuses attention on the informal business requirements that have not yet been expressed in a top-level specification.

Unless the specifications S is inappropriate to the system intent, these mutations are all undesirable; they all violate properties of S even though they are refinements of P. If S is thought of as the refinement of a number of higher level specifications, only one of which has been made explicit, then these mutations of the system specification represent refinements of the wrong implicit higher-level specifications.

4.5 Refinement: reduced functionality

Some mutants are refinements of P and also produce the same functionality as S in all areas where anything happens in the system. These include mutations that cause the system to make no transfers, but to do so without deadlock and without breaching the security properties, and systems that do some transfers but maintain the security properties.

There are two kinds of mutations in this category.

1. No action possible. There are a number of mutations that prevent the system from doing any transfers. This is achieved by particular mutations of the conditional expression in either the *Purse* process (for example, SSR1) or the the *FromPurse* process (for example, EUR1). In the first case, the system has been forced into an idle loop, by replacing the first conditional predicate of *Purse* with *true*; in the second, the system is always forced to *ignore*, because the the first conditional predicate of *FromPurse* with a statement that is always false.

2. Limited action. A number of mutants strengthen preconditions or reduce non-determinism. Where these mutants are valid refinements of the security properties, they are acceptable in the sense that they do not cause anything unexpected to happen in the system.

4.6 Lessons learnt

Equivalent mutants yield valuable information about the system, and expose implicit assumptions about required (and not-required) functionality.

In many cases, the combination of harmful and harmless mutants is useful. PRR3 (which leads to the first predicate of the conditional in *FromPurse* being permanently *false*) and EUR1 (which sets this predicate to *true*), together indicate that the predicate is needed, and, with similar mutants, increase confidence that it appropriate (that is, better than an equality or a strict inequality). Again, where one mutant strengthens a precondition and is a valid refinement (for example, ESK applied to *bal*ance) and another loosens the precondition and causes deadlock (for example, ESK applied to *val*ue), the specifier can be confident that the precondition in the initial specification is exactly strong enough.

5 Conclusions

This work has led to the following conclusions.

Equivalent Mutants are Useful. With code mutation, equivalent mutants are problematic. With specification mutation they are a useful source of information. Equivalent mutants provide alternative refinements of P. Challenging the developers to choose between the original and each mutant can increase confidence in appropriateness of the specification. Equivalent mutants can indicate robustness to specific types of failure in development. Note that, from a security point of view, it may not matter if the implementation refines the mutant specification. Equivalent mutants may also indicate areas where the current specification is over-constrained.

Non-equivalent Mutants are Useful Too. Non-equivalent mutants highlight areas of system fragility. The original specification may be only one failure away from insecurity. Non-equivalent mutants may represent behaviour actually implemented (erroneously), or else model avenues of attack for the maliciously minded.

Small may be too Beautiful. A common practice in verifying systems with unbounded or large numbers of participants is to model-check an exemplary (small) system. This requires care. The exemplar must be small enough to be tractable and yet large enough to reveal interesting features under analysis. Specification mutation can expose the fragility of verification evidence to specific choices of parameters. In a sense, mutation shakes the verification evidence to see how robust it is. The results may show that the evidence does not generalise from specific successful instances or else increase confidence that it does so.

Specification Mutation is a Challenge. It is generally accepted that formal methods provide a highly rigorous approach to 'getting the system right'. In this paper we have demonstrated that specification mutation provides an interesting and informative avenue for 'getting the right system', and for exploring what can happen when you 'get the wrong system' (for whatever reason). Specification mutation can be used to challenge various assumptions and in doing so complement the areas where formal methods currently succeed. We have attempted to address in a formal way, some of the current weaknesses of formal approaches to system development. We believe that specification mutation has much to offer.

6 Future work

We believe that our work can be extended in various ways:

Mutation of critical properties. In the work presented above, we have applied mutation to the process definitions and have kept the definition of P constant. It would seem plausible to mutate the property constraints such as P and keep the system process definitions constant. By inverting the current approach in this way we could challenge the developers with respect to their encapsulation of critical properties. This could provide a useful means of reducing the risk of 'proving the wrong thing'.

Application to Safety Critical Systems. We have chosen to illustrate our mutation approach with a security property. The prototype tool set [17] was originally developed with safety applications in mind. Our approach could be adapted to support an automated Failure Modes Effects Analysis [20] or automated HAZOPs [7] variant.

Extension to Other Notations. The approach described in this paper is not limited to the specific formal notation used (CSP). Similar approaches could be adopted for challenging system descriptions expressed in B, Z or other formal notations. The technique could even be applied informally to particularly critical sections of the system, but best results require automation to support the process. Automating the generation of mutants is relatively easy. More sophisticated support is needed to provide examples of distinguishing behaviours. The approach can also be applied to less formal notations. The authors have, in fact, made significant progress in applying fault injection techniques on use cases.

Mutations as Future Systems. Mutant specifications could be considered as small future changes to the current system. The ratio of equivalent mutants to non-equivalent mutants might give an indication of the robustness of the specification to change.

References

1. R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* Wiley, 2001.

2. D. E. Bell and L. J. Lapadula. Secure computer systems: Mathematical foundations and model. Technical report, M74-244, The MITRE Corporation, 1974.
3. P. E. Black, V. Okun, and Y. Yesha. Mutation of model checker specifications for test generation and evaluation. In *Mutation Testing for the New Century, Proceedings of Mutation 2000*, pages 14–20. Kluwer, October 2000.
4. T. A. Budd and A. S. Gopal. Program Testing by Specification Mutation. *Computer Languages*, 10(1):63–73, 1985.
5. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.
6. J. A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.
7. T. Kletz. *Hazop and Hazan: Identifying and Assessing Process Industry Hazards, 3rd edition.* Institution of Chemical Engineers., 1992.
8. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings Advances in Cryptology (CRYPTO '99)*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
9. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer, 1996.
10. Formal Systems (Europe) Ltd. *Failure-Divergence Refinement - FDR2 User Manual*, 2000.
11. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *CACM*, 21(12):993–999, 1978.
12. A. W. Roscoe. Model-checking CSP. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378. Prentice Hall, 1994.
13. A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings 1995 IEEE Symposium on Security and Privacy*, pages 114–127. IEEE Computer Society Press, 1995.
14. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and A. W. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach.* Addison Wesley, 2000.
15. S. Schneider. Security Properties and CSP. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 174–187. IEEE Computer Society Press, 1996.
16. D. X. Song. Athena: A new efficient automatic checker for security protocol analysis. In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop.* IEEE Computer Society Press, 1999.
17. T. Srivatanakul. Mutation testing for concurrency. Master's thesis, Department of Computer Science, University of York, UK, 2001.
18. S. Stepney, D. Cooper, and J. Woodcock. More powerful Z data refinement: pushing the state of the art in industrial refinement. In *ZUM'98: 11th International Conference of Z Users, Berlin 1998*, volume 1493 of *LNCS*, pages 284–307. Springer, 1998.
19. S. Stepney, D. Cooper, and J. Woodcock. An Electronic Purse: Specification, Refinement and Proof. Technical Monograph PRG-126, Oxford University Computing Laboratory, July 2000.
20. A. Villemeur. *Reliability Availability Maintainability and Safety Assessment.* Wiley, 1992.
21. J. M. Voas and G. McGraw. *Software fault injection: inoculating programs against errors.* Wiley, 1997.

A Mutation Operators

Events		
PED	Event Drop	drop one event from the process definition (unless only one event)
PER	Event Replacement	replace event by other events: (i) Local change on event – replace each event by events within the same process definition (ii) Global change on event – replace each event by events globally available within the specification
PES	Event Swap	swap two neighbouring events in the process definition.
PEI	Event Insertion	insert (by duplication) one event after each event in the process definition
Operators		
PCO	Choice Operator Replacement	replace the choice operator (external choice ($\Box$) and internal choice ($\sqcap$)) with each other.
PPO	Process Operator Replacement	replace the operator on processes (concurrency operator ($\|$), interleaving operator ($\|\|\|$) and sequential composition operator (;)) with each other.
Communications		
PMR	Message Replacement	replace the message of each communication channel with other messages
PCR	Channel Replacement	replace the channel with other channels within the process definition
PCI	Communication Insertion	insert one communication event (by duplication) after each event in the process definition
PCE	Communication Elimination	eliminate one communication event from the process definition (unless only one event)
PCS	Communication Swap	swap two neighbouring communication events in the process definition
Processes		
PPR	Process Replacement	replace each process name in the right hand side of the definition to other process: (i) replace with other process name available within the specification (ii) replace with *Stop* (iii) replace with *Skip*

Table 11: CSP Process Definition Modification Operators

Logical		
ENI	Negation Operator (not) Insertion	insert *not* before Boolean variable

ELR	Logical Operator Replacement	replace logical operator ($\wedge$, $\vee$) with other logical operator.
ELD	Logical Operand Replacement	replace logical operand with *true* or *false*
Arithmetic		
EAR	Arithmetic Operator Replacement	replace arithmetic operator $(+, -, *, /)$ with each other arithmetic operator.
EUR	Unary Operator Insertion	insert the unary operator $-$ in front of each arithmetic expression and sub expression.
EAK	Add k to Operand	add k to arithmetic expression and sub expression
ESK	Subtract k from Operand	subtract k from arithmetic expression and sub expression
EAD	Arithmetic Operand Replacement	replace arithmetic operand by other arithmetic operand
Relational		
ERR	Relational Operator Replacement	replace relational operator $(<, \leq, >, \geq, =, \neq)$ with other relational operator.
ERD	Relational Operand Replacement	replace relational operand with other relational operand

Table 12: CSP Expression Modification Operators

SSR	IF Statement Condition Replacement	remove part of the IF statement by forcing it to perform only one action always.
SCR	Condition Statement Replacement	force the condition to either be always *true* or *false*

Table 13: CSP Structure Modification Operators

PAK	Add k to Parameter Element	increase value of a parameter element by k
PSK	Subtract k from Parameter Element	decrease value of a parameter element by k
PRR	Parameter Replacement	replace one of the elements in the process parameter with other value

Table 14: CSP Parameter Modification Operators