Developing safe exception recovery mechanisms for CHERI capability hardware using UML-B formal analysis

Colin Snook[0000−0002−0210−0983], Asieh Salehi Fathabadi[0000−0002−0508−3066], Thai Son Hoang[0000−0003−4095−0732], Robert Thorburn[0000−0001−5888−7036], Michael Butler[0000−0003−4642−5373], Leonardo Aniello[0000−0003−2886−8445], and Vladimiro Sassone[0000−0002−6432−1482]

School of Electronics and Computer Science (ECS), University of Southampton, U.K.
{cfs, a.salehi-fathabadi, t.s.hoang, robert.thorburn, m.j.butler, l.aniello, vsassone}@soton.ac.uk

Abstract. While detection of suspicious or erroneous CPU behaviour can be achieved by generic mechanisms such as memory safe processors, recovering safely from the resulting exceptions is an application specific problem. The challenge is to ensure that a complex closed system including controller and its environment remain in a safe state while undertaking abnormal state changes in the controller as part of its exception recovery process. Handling exceptional error events is a complex task requiring insight and domain expertise to ensure that a process is designed to recover from abnormal conditions and return the system to a safe state. Exception handling relies on a notion of *transactions* in order to identify how the system can be systematically returned to a consistent state. Formal methods can address this complexity, by supporting the analysis of transactions and exception handling at the abstract design stages utilising mathematical modelling and proofs. Event-B is a state-based formal method for modelling and verifying the consistency of discrete systems, however it lacks explicit support for analysing the handling of exceptions. UML-B is a diagrammatic front-end for Event-B modelling which allows models to be constructed using class diagrams and state-machines. In this paper, we use UML-B state machines to support the modelling of normal behaviour, with a notion of consistency and augment this with a technique for modelling 'transactions' which may either complete to reach a consistent state or encounter exceptional errors that have to return the system to a consistent state despite the non-completion of the transaction. We also discuss an implementation of the modelled exception handling in the 'C' programming language as a first stage towards automatic code generation of exception handlers.

Keywords: Exception handling · Formal methods · Event-B · UML-B

1 Introduction

Our work is influenced by considering implementations on *capability hardware* which provides hardware level protection against incorrect memory access [18].

Capability hardware blocks unauthorised memory access at runtime, raising hardware exceptions that should be handled by application code. Unauthorised memory access might be caused by unintentional coding errors, such as out of bounds array access, or malicious attacks, such as buffer overflow exploitation. In principle, code that is developed formally will be free from incorrect memory access. However, we assume the applications we develop will operate in software environments where vulnerabilities remain, e.g., through use of untrusted libraries.

Mechanisms for detecting exceptional erroneous behaviour are often generic since they flag unusual activity in the underlying low-level machinery. An example is the CHERI [18] memory safe capability approach which is implemented within general purpose electronic computing devices. In contrast, the design of a suitable recovery response to the detected exception is usually application, or at least domain, specific. In some cases a safe response might be to halt, but this could play into the hands of a malicious attacker by providing an easy vector to achieve denial of service attacks. In many cases it is not safe for a critical service to halt. Therefore we believe that generic memory protection mechanisms like CHERI are only useful if they are complimented by tools and techniques for application engineers to design and implement safe recovery strategies that allow the system to continue its service as much as possible.

We already use formal modelling tools to support the rigorous analysis of systems ensuring that they meet important (e.g. safety and security) properties. In HDSEC[1] we have adapted these formal analysis tools to show how they can be used to design and analyse exception recovery responses and verify that they recover the system to a condition that satisfies the important system properties. We focus on designing a safe recovery after an exception and abstract away from the mechanisms that detect the exception.

We have also implemented the modelled system in order to demonstrate the recovery responses in a real system running on a CHERI Morello PC. The implementation is a demonstrator that also contains a simulation of the environment and the user interfaces. The code is seeded to allow a capability exception to be detected so that the recovery can be demonstrated.

Programming languages provide a framework for detection, notification and handling of exceptions. Exception handling is a complex and error-prone activity, and systematic reasoning is needed to identify and characterise exceptions. Formal analysis of the exceptional control flow provides a means to validate the design of the exception handling recovery [6]. However, support for exceptions in formal methods is less mature. This paper proposes an approach to systematic reasoning about exception handling at the design level using the UML-B and Event-B formal method.

Event-B [4] is a formal method to model and verify correctness of safety/security critical systems. While exception handling can be modelled within the existing features of the Event-B toolkit, there is no explicit support for it. We use UML-B [16] and Event-B to visualise and verify the normal expected behaviour

[1] https://hd-sec.github.io/

of a system and then add support for handling exceptions in safety/security systems from the design level to the implementation. The encoding of statemachine states provides a) a mechanism for detecting where the exception occurred and hence choosing the appropriate recovery, and b) for going into a suitable recovery state. We propose extensions to the UML-B statemachine notation to facilitate the automatic deduction of which transitions represent exception recovery and how system variables should be rolled back during a recovery. The Event-B (generated by UML-B) already has sufficient features to express the recovery behaviour and does not need to be extended.

We illustrate our approach using a Smart Ballot System (SBB) [2], an integral part of some modern voting systems. Earlier research work [8] presented a correct-by-construction secure SBB system using Event-B. Our proposed approach can address the robustness of SBB against exceptions in [8].

This work follows on from previous work presented at ABZ2024 [13]. We have revised the way that we model and verify exception recovery so that it is better integrated with the UML-B development of normal behaviour and is modelled using statemachine transitions. We have introduced a notion of transactions and model these with superstates. This enables us to formally verify that exception recovery correctly rolls back any partially completed transaction behaviour. We have also added a demonstration implementation which runs on a Cheri Morello memory safe PC.

The paper is structured as follows. Section 2 introduces POSIX signals, Event-B and the SBB case study. Our proposed approach is described using the case study, firstly by modelling the normal behaviour in Section 3, then by adding a concept of transactions in Section 4 and finally by adding the exception recovery in Section 5. Section 6 summarises an overview of the complete approach, Section 7 describes the implementation of the case study on a Cheri Morello machine as a demonstration. In Section 8, we review existing literature and research, highlighting key methodologies, findings, and gaps that our study aims to address. Finally Section 9 discusses our plans for the next steps and Section 10 summarises related works and concludes.

2 Background

Signals are a mechanism for asynchronous event notification used in Unix-based (POSIX-compliant) operating systems. Signals are used by the kernal to interrupt (e.g. suspend, terminate or kill) a process. When an event occurs, the operating system interrupts the target process' normal flow of execution to handle the signal. If the process has registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed. The CHERI-BSD operating system [1] running on Morello hardware adds a new signal SIGPROT which is used to notify the active process that the Morello hardware has detected a memory protection error. In our example case study, we also use the standard signal SIGALRM, which is used to notify that a timeout set by a process has expired.

Event-B [4] is a refinement-based formal method for system development. The mathematical language of Event-B is based on set theory and first order logic. An Event-B model consists of two parts: *contexts* for static data and *machines* for dynamic behaviour. Contexts contain carrier sets, constants, and axioms that constrain the carrier sets and constants. Machines contain variables, invariant predicates that constrain the variables, and events. In Event-B, a machine corresponds to a transition system where *variables* represent the states and *events* specify the transitions. An event comprises a guard denoting its enabling-condition and an action describing how the variables are modified when the event is executed. Event-B is supported by the Rodin tool set [5], an extensible open source toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques. In this paper we make extensive use of the UML-B plug-in [17] which provides a diagrammatic modelling notation for Event-B in the form of state machines and class diagrams that automatically generate Event-B models.

SBB (Smart Ballot Box) [2] is a computerised system to automate election voting. The SBB system inspects a ballot paper by detecting a barcode and decrypting it to evaluate whether the ballot is valid. If the ballot is valid, then a vote can be cast, spoiled or cancelled by the user and the ballot paper is sorted accordingly into the storage boxes. If the ballot is not valid, the SBB rejects the paper. The key function of the SBB is to ensure that only valid ballot documents are included in the ballot boxes.

3 Modelling normal-behaviour and verifying safety invariants

Utilising UML-B, we model the SBB normal behaviour (without exceptions) as a state-machine. [2]. The normal-behaviour SBB case, presented in Figure 1, starts in the Waiting state and, in the case of accepting the ballot, progresses through the following sequence of states: Waiting, BarcodeReading, BarcodeProcessing, UserSelection, PrepareAccepting, Accepting, Waiting. There are several functional variables (not shown in the state-machine diagram) which are manipulated by actions of the transitions. They are

- paper_count - a count of the papers input to the roller (incremented by the transition ROLLER_paper_in),
- accepted_count - a count of the papers categorized as accepted by the roller (incremented by the transition ROLLER_accept_paper),
- spoilt_count - a count of the papers categorized as spoilt by the roller (incremented by the transition ROLLER_spoil_paper),
- rejected_count - a count of the papers categorized as rejected by the roller (incremented by the transition ROLLER_reject_paper),

[2] The example models described in this paper are available here: https://tinyurl.com/ABZ2025-SnSaHo. The Rodin and UML-B formal modelling tools used, are available as a bundled installation package via https://www.uml-b.org/Downloads.html.

Fig. 1. State Machine, normal-behaviour SBB

– cast_count - a count of the votes cast by the user (incremented by the transition USER_cast),

The Waiting state contains two desired *safety* properties that are expected to hold when the SBB has completed the processing of any papers and is in the Waiting state:

– The count of votes cast by the user (cast_count) should be the same as the count of papers categorized as accepted by the roller (paper_count).
– The count of papers input to the roller should be the same as the sum of papers categorized as accepted, spoilt or rejected by the roller.

In general a system, may have important properties that are expected to hold whenever the system is quiescent but that are temporarily violated while the system is engaged in active processing. We refer to properties that are expected to hold in *quiescent states* as *quiescent invariants* and states that are not quiescent as *active states*. Active states may contain *intermediate invariants* that describe the expected progress during the activity. In fact, intermediate invariant properties are needed in the active states to help the provers prove the quiescent invariants are re-established. This is because the prover considers one transition at a time and attempts to infer the invariants of the post-state (e.g. the desired quiescent invariants) from the known pre-state (e.g. intermediate invariants) as well as any guards of the transition. Hence, since the prover cannot 'see' back up the sequence of transitions, we have to provide this sight via the intermediate invariants in the active states. Once this is done, the proofs are automatically discharged by the Rodin provers. Notice how the intermediate invariants document where the counts are out of step and by how much. For example in the Accepting state, cast_count is one ahead of the accepted_count because it has been incremented by the transition User_cast, but the Roller has yet to finish categorizing the paper as accepted.

In order to encode the state machines in Event-B, the UML-B tools automatically generate sets, constants and axioms in a newly generated context component. The SBB states are an enumeration of a carrier set where each state (Waiting, BarcodeReading, ...), is specified as a *constant* and the set of states, SBB_STATES, are specified as an axiom using *carrier sets*. The enumeration is then specified as a partition via the following axiom:

```
@axm1: partition(SBB_STATES, {Waiting}, {BarcodeReading},{BarcodeProcessing},
    {UserSelection}, {Accepting}, {Spoiling}, {Rejecting}, {PrepareRejecting}, {PrepareSpoiling},
    {PrepareAccepting})
```

The dynamic behaviour of the state machine (Figure 1), is generated as part of the containing machine component. Each event that represents a transition, checks, within its guards, that the current state of the SBB is the transition source state, and changes the state to the transition target state, within its actions. For example:

```
event BR_reading_succeeds
when
    @grd1: SBB =BarcodeReading
    <other guards about the functional vars>
then
    @act1: SBB :=BarcodeProcessing
    <other actions on the functional vars>
end
```

4 Identifying and adding transactions

In Section 3 we saw how the verification of the quiescent (safety) invariants led us to introduce intermediate invariants that document where (i.e. in which states) the functional variables are out of step (i.e. do not satisfy the quiescent invariants). We could think of these states, and the transitions that are involved in passing through them, as a process or *transaction* which must be completed to bring the system back to a safe state. In this section, we show how we identify such transactions and represent them in the model.

4.1 Adding transactions to the state-machine model

In UML-B we can arrange state-machine states hierarchically by nesting a state-machine within a superstate. We use this superstate structure here to represent the transactions. We introduce a transaction superstate to contain all the states that have a similar same intermediate invariant. Some of the contained states may have other intermediate invariants that differ within them.

For example in Figure 1 all of the states, BarcodeReading, BarcodeProcessing, UserSelection, PrepareAccepting, Accepting, PrepareSpoiling, Spoiling, PrepareRejecting and Rejecting have the same invariant:
paper_count = accepted_count+rejected_count+spoilt_count+1
because a paper has been fed in to the roller but since its processing is not complete, none of the accepted, rejected or spoilt counts has been increased yet. Hence this group of states form a transaction and we wrap them in a superstate Paper_in_transaction. A useful feature of superstates is that they can contain invariants that apply throughout all of their contained sub-states. Therefore we can move the intermediate invariant that we used to identify the transaction up to the superstate and remove all the repetitions of it in the sub-states.

The modified model is shown in Figure 2

Notice that another intermediate invariant: accepted_count+1 = cast_count identifies a transaction consisting of the states Prepare_Accepting and Accepting (but not the other sub-states of the previous transaction). Hence we have a nested transaction and can introduce a further transaction superstate Cast_count_transaction to contain those two sub-states and move the transaction intermediate invariant into it.

As soon as we re-generate the Event-B for the UML-B model, the automatic provers re-prove the model and verify that the quiescent invariants are still satisfied. The changes are superficial notational ones which do not change the semantic.

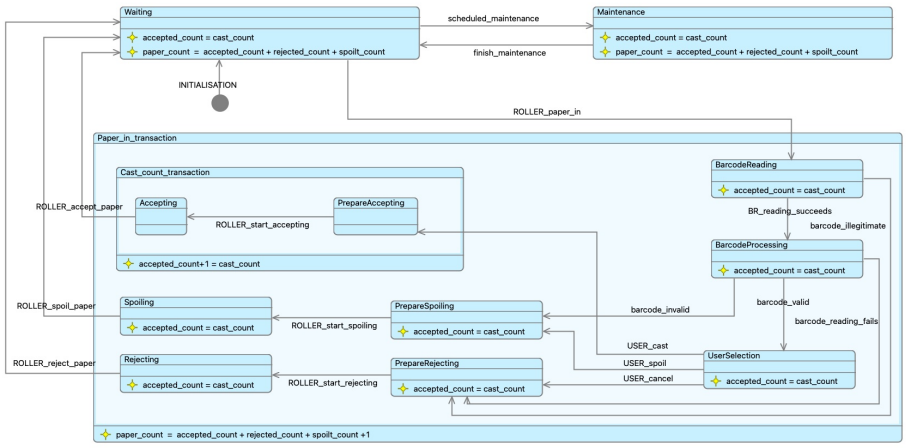

Fig. 2. State Machine, SBB with transactions

4.2 Adding rollback caching to transactions

Our model so far only deals with successful outcomes of transactions (even if it is a successful rejection response by the controller). However, the aim of identifying transactions is to consider failure cases where the transaction does not complete, which, by definition, leaves the system in an invalid condition requiring some recovery process. There are several possible approaches to recovering a safe and valid condition:

1. design specific compensation actions for each recovery (rollback is $V = G(V')$, where V' is the state of the variables V that may be altered in the transaction and G is the transformation that had completed before the exception occurred),
2. modify temporary copies of the variables and only commit their values to the real system variables when the transaction completes (no rollback is needed, but there is a pre-transaction action $V' = V$ to make temporary copies of V and there is a commit action $V = V^T$ where V^T is the value of the temporary copies of V),
3. save the values of system variables before the transaction and revert them if the transaction does not complete. (Rollback is $V = V'$ and there is a pre-transaction action $V' = V$ to make temporary copies of V),

We discount the first approach since it is difficult to know how much of the transformation G had completed. There is not much to choose between the second and third approaches. We have chosen to adopt the last approach so that the normal behaviour uses the actual variables.

We first add a duplicate set of *rollback* variables to the Event-B machine for all the variables that are altered during transactions. We then add entry actions to all the transaction superstates to save the entry values of the variables that will be modified by the transaction, in the rollback variables. We then add intermediate invariants to the transaction superstate to confirm that the values in the rollback variables, satisfy the quiescent invariant. That is, we make a copy of the quiescent invariants and replace the variables with the rollback variables used by that transaction.

For example, in SBB, we add the invariant:
paper_in_rollback = accepted_count + rejected_count + spoilt_count
to Paper_in_transaction and cast_count_rollback = accepted_count to the Cast_count_transaction.

These will be needed in the next step to prove that exceptions establish the quiescent invariants when they use the rollback variables to restore the values of variables that have been changed in the transaction.

This process of adding rollback variables is done for each of the transactions, including nested ones. Note that the rollback variable should be used by the lowest level transaction possible. For example, in the SBB model, cast_count is saved as rollback_cast_count by Cast_count_transaction, not by Paper_in_transaction.

5 Adding exception handling to transactions

Having prepared by identifying transactions and their associated rollback requirements, in the next step we identify where exceptions could occur and how the system should recover from them. The model should be analysed state by state to identify negative outcomes that could prevent the activities within the state from completing successfully. Since the model abstracts away from the details of these state activities, identification of exceptions is a subjective assessment of the concepts represented by the state. As we are interested in the memory safety provided by Cheri hardware, we might consider certain states to be particularly untrusted (whether malicious or accidental). We may also wish to consider failures due to external system components such as user mis-actions and machinery failures. For example, Table 1 outlines the potential exceptions and their recovery strategy in the SBB system: The first exception is a memory

Exception	Recovery
memory capability violation in barcode software library	if occasional, reject the ballot, if persistent, external maintenance
user does not enter selection within timeout	reject the ballot
roller does not complete within timeout	external maintenance

Table 1. Exceptions handled by the SBB system

capability violation in the barcode processing software. This could be due to a simple software error or it could be due to a security attack via virus software which is trying to use memory accesses to create an attack vector. We could react by disabling the service immediately to ensure that the SBB does not record invalid results. However, this could play into the hands of an attacker trying to create denial of service attack. Therefore, we decided to adopt a two-phase recovery strategy. For occasional exceptions, the paper is rejected and the user can try again. It is the best we can do since the paper cannot be processed without a barcode. If several exceptions are detected consecutively, then the service is aborted and the system awaits external intervention.

The second exception is a timeout on the user choosing either to cast, spoil or cancel their vote. In this case the recovery strategy is to default to rejecting the paper. The third exception is a breakdown in the roller machinery that sorts the physical papers into their respective categories. If the roller does not complete within a timeout, it is assumed that manual maintenance will be required to fix the roller machinery. (Note that we are not interested in quantifying intervals of time; only in an ordinal arrangement of events and therefore, we do not need to model the tick of a clock).

Figure 3 shows the UML-B model with exception handling transitions added. The

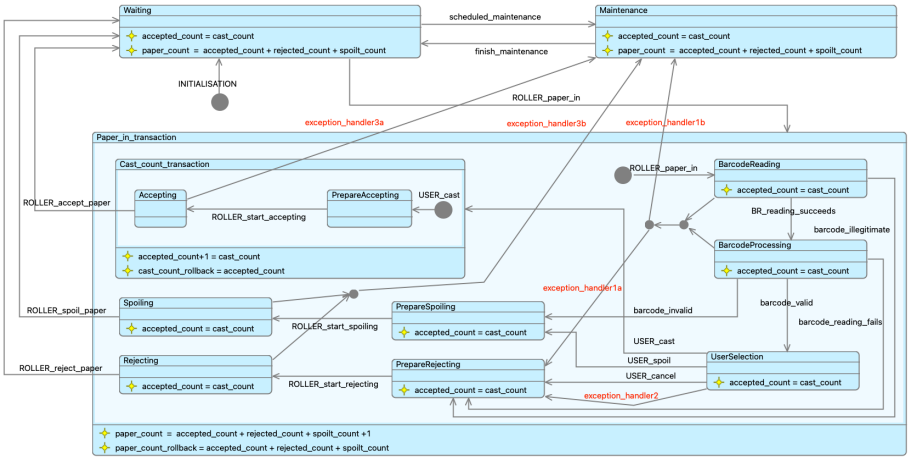

Fig. 3. State Machine, SBB with exception handling

first exception can occur either in the BarcodeReading or BarcodeProcessing states and can result in two different exception handlers. If an exception counter (which is not shown in the diagram) is below a threshold, exception_handler1a recovers to the PrepareRejecting state. This does not leave the transaction Paper_in_transaction so does not need to use the rollback mechanism. However, the exception count is incremented as part of the exception handling. If the exception count reaches the threshold the exception is handled by exception_handler1b which exits the Paper_in_transaction and recovers to the Maintenance state. In this case the paper_count is rolled back by the action paper_count := paper_count_rollback which is attached to the transition exception_handler1b. The second exception can occur in the state UserSelection and is handled by the transition exception_handler2 which recovers to PrepareRejecting without any rollback actions. The third exception can occur in Accepting, Spoiling or Rejecting and always recovers to Maintenance with the same paper_in rollback action as the first exception. However, in the case where it occurs in Accepting, the exception also exits the nested Cast_count_transaction and therefore must also roll back the cast_count via an action cast_count := cast_count_rollback which is attached to the transition exception_handler3a. (Note that, in Event-B, conditional actions are only possible using different guarded events for each condition hence the need for separate transitions for exception_handler3a and exception_handler3b).

6 Overview of Method of modelling transactions and exceptions in UML-B

The generic technique for modelling transactions and exceptions and analysing their recovery using UML-B state-machines and Event-B verification is summarised in this section.

– Model the normal behaviour as a UML-B state-machine.
  • Construct a UML-B state-machine to model the control modes (states) and mode changes (transitions) of the system .

  - In the containing machine, add additional variables involved in the functionality. The variables may be used to control (guard) the firing of transitions and be altered when transitions fire (actions).
  - Add quiescent invariants to the states to express desired safety properties about the expected values of the variables in particular quiescent states[3].
  - Verify the model using the Rodin provers, adding intermediate invariants to states in order to achieve the proofs.
- Identify and represent any transactions in the model.
  - Where intermediate invariants indicate that variables are out of step in a sequence of states (i.e. are different from the quiescent invariants) a superstate should be introduced to represent the transaction.
  - The sequence of states containing the intermediate invariants is then contained in a nested state-machine within the transaction superstate.
  - The transition that enters the parent transaction superstate will contain an action that alters the variable that is out of step (i.e. introduces the difference from the quiescent invariant).
  - The intermediate invariants expressing the difference from the quiescent invariant are replaced by a single intermediate invariant in the parent transaction superstate.
  - Transactions may be nested within other transactions where a variable is changed in a sub-transaction.
  - Check that the model still verifies using the Rodin provers. The changes are superficial/structural so should not affect the validity of the proofs.
- Add rollback caching of variables to support the transactions.
  - In the containing machine, add rollback variables to store the entry state of all of the ancillary variables that are altered during the transaction.
  - Add entry actions to the transaction superstate to cache the value of the variables that will be changed by the transaction, in their corresponding rollback variables.
  - Add intermediate invariants to the transaction superstate to confirm that the quiescent invariants, with variables replaced by rollback variables, obey the quiescent properties. These will be needed in the next step to prove that exceptions re-establish the quiescent invariants when they use the rollback variables to restore the values of variables that have been changed in the transaction.
  - Check that the model still verifies using the Rodin provers. The proofs should be straightforward.
- Add exception handling to the model.
  - Consider each state in turn and identify any potential exceptions that could occur in that states actions.
  - Add transitions to represent exceptions that can occur from states within the transaction.
  - Their target (recovery) states can be within the transaction or external to the transaction.
  - Junctions can be used when the same exception handler can handle an exception occurring in several source states.

[3] We refer to these invariants as *safety* properties, however, we use *safety* in a very broad way to represent any properties the modeller would like to remain true in this model

- Junctions can also be used to model alternative exception handlers (with different recovery target states) of the same exception. In this case, guards on the final segments of the transition, can be used to distinguish the cases and they can have different rollback actions.
- For exceptions that exit a transaction, add actions to the transition to roll back the variables that have been changed (i.e. v := rv where rv is the rollback variable for variable v).
- Exceptions must add rollback actions for each of the nested transaction superstates that are exited.
- Verify the model using the Rodin provers.

7 Demonstration Implementation

We have implemented the modelled system in order to demonstrate recovery responses from an exception signal in a real system running on a CHERI Morello PC. The implementation is a demonstrator that also contains a simulation of the SBB environment (the roller machine) and the user interfaces. An invalid memory access is seeded in the barcode processing simulation so that a SIGPROT signal can be induced as part of the demonstration. The user simulation asks the tester to supply the expected user responses and if this is delayed sufficiently, a SIGALRM is induced for demonstration purposes. Although this is just a demonstrator program, we can envision the controller code, including the exception handling, being generated automatically from the UML-B model.

The processing of a state-machine state and firing of transitions, is wrapped in a conditional sigsetjmp which acts as a kind of 'try' (see Figure 4 for SBB example).

```
do forever {
    try {
        set alarm timeout for the new state
        repeat until the SBB state changes
            progress the environment
            progress the statemachine }
    //any exception handler will return to here }
```

The function that progresses the statemachine selects the case based on the current statemachine state and tests to see whether it has the necessary conditions to fire any of its outgoing transitions. The conditions may involve trigger events from the environment, user inputs, internal system variables or may be always true (i.e. the next transition fires immediately).

```
switch state
case STATE1:
    if can fire TRANSITION1
        fire TRANSITION1
    else if can fire TRANSITION2
        fire TRANSITION2
    etc.
case STATE2:
    fire TRANSITION3
    etc.
```

When fired, transition functions take any transition actions such as changing system variables and then update the statemachine state to the new (target) state. If the source state has any exit actions or the target state has any entry actions, these are also added as transition actions.

```
void SBB_statemachine(){
    bool  changedState=false;
    while (true){
        if (sigsetjmp(SBB_abort_step,true) ==0) {   // TRY
            if (sbb_control.state==SBB_Null){
                printf("Something went wrong! SBB_state is Null\n");
            }else{
                alarm(getAlarm());      //set the timeout for the current state
                changedState = false;
                printCurrentState();
                do {
                    ROLLER_step();                  //progress the roller simulation
                    changedState = SBB_checkState(); //see if we can change the state
                } while (!changedState);            //repeat until can change state
            }
        }   //SBB_abort_step - exit point for handled exceptions
        sbb_control.trigger = NULL_TRIGGER;
    }
}
```

Fig. 4. Code for the main SBB state-machine execution showing 'sigsetjmp'

```
void SBB_setup_exception_handling(){
    //set up mask to only allow further interrupts
    //from exceptions other than the ones we handle
    sigemptyset(&SBB_sa.sa_mask);
    sigaddset(&SBB_sa.sa_mask, SIGPROT);
    sigaddset(&SBB_sa.sa_mask, SIGALRM);
    //assign the exception handler routine
    SBB_sa.sa_handler = (void *)SBB_Handler;
    //assign the sigaction to SIGALRM and SIGPROT
    sigaction(SIGALRM, &SBB_sa, &SBB_SIGALRM_oldsa);
    sigaction(SIGPROT, &SBB_sa, &SBB_SIGPROT_oldsa);
}
```

Fig. 5. Code for setting up the exception handler using 'sigaction'

If there is an exception (which could be any POSIX signal but we use SIGPROT and SIGALRM as examples) the exception handler will be called to intervene with any roll back actions and change the state to the appropriate recovery action. The exxception handler then exits, using a siglongjmp, to the end of the main sigsetjmp (try) conditional block. This is also where main loop ends up after a normal transition in order to enter a new state. Therefore the exception handler sets up the next state variables to enter the designated recovery state depending on the exception that was raised and the state that was executing when the exception occurred.

The exception handling is set up at intialisation using sigaction which is a facility built in to POSIX signals library for this purpose. The sigaction assigns our exception handler to the handled signals. (See Figure 5). Note that we use a single exception handler routine for both signals. The different exception transitions of the UML-B model map to different condition branches within the handler. (Event-B does not support conditional execution within an event).

The exception handler, shown in Figure 6, contains a switch case for each type of signal that is handled and each case contains conditional branches for the state(s) that the signal has a defined recovery. The choice of recovery state can also be conditional for a particular signal source state combination and recovery may or may not require rolling back system variables.

```
void SBB_Handler(int sigtype, siginfo_t *info, void *context) {
    printf("\n>>SBB Handler - sigtype: %d while in state ", sigtype);
    printCurrentState();
    record_exception(info);
    switch (sigtype) {

        case SIGPROT:   //capability  violation
            if (    sbb_control.paper_in_state == SBB_BarcodeReading ||
                    sbb_control.paper_in_state == SBB_BarcodeProcessing) {
                //Exception 1  has conditional recovery targets
                if (sbb_control.attack_count<ATTACK_LIM){  //Exception1a
                      printf(">>> attack counter < attack limit\n");
                      sbb_control.state = SBB_Paper_in;
                      sbb_control.paper_in_state = SBB_PrepareRejecting;
                      sbb_control.attack_count++;
                      printf(">>> inc attack counter to %d\n", sbb_control.attack_count);
                }else if (sbb_control.attack_count>=ATTACK_LIM){  //Exception1b
                      printf(">>> attack limit EXCEEDED\n");
                      sbb_control.state = SBB_Maintenance;
                      sbb_control.paper_in_state = SBB_Paper_in_Null;
                      sbb_control.attack_count = 0;
                      printf(">>> reset attack counter to %d\n", sbb_control.attack_count);
                      //roll back paper count as exiting transaction Paper_in
                      sbb_data.papers = sbb_rollback.papers;
                      printf(">>> rolled  back papers to %d\n", sbb_data.papers);
                }else{
                    printf(">>> no handling defined for this signal-state-condition... ignoring\n");
                }
            }else{
                printf(">>> no handling defined for this signal-state... ignoring\n");
            }
            break;

        case SIGALRM:   //timeout
            if (sbb_control.paper_in_state == SBB_UserSelection){   //Exception2
                sbb_control.state = SBB_Paper_in;
                sbb_control.paper_in_state = SBB_PrepareRejecting;
            //Exception 3 has 2 different cases due to Accepting being within a nested transaction
            }else if (sbb_control.cast_count_state == SBB_Accepting){   //Exception3a
                sbb_control.state = SBB_Maintenance;
                sbb_control.paper_in_state = SBB_Paper_in_Null;
                sbb_control.cast_count_state = SBB_Cast_count_Null;
                //roll back paper count as exiting transaction Paper_in
                sbb_data.papers = sbb_rollback.papers;
                //roll back cast count as exiting transaction Cast_count
                printf(">>> rolled  back papersto %d\n", sbb_data.papers);
                sbb_data.cast = sbb_rollback.cast;
                printf(">>> rolled  back cast to %d\n", sbb_data.cast);
            }else if (sbb_control.paper_in_state == SBB_Spoiling ||
                        sbb_control.paper_in_state == SBB_Rejecting){   //Exception3b
                sbb_control.state = SBB_Maintenance;
                sbb_control.paper_in_state = SBB_Paper_in_Null;
                //roll back paper count as exiting transaction Paper_in
                sbb_data.papers = sbb_rollback.papers;
                printf(">>> rolled  back papers to %d\n", sbb_data.papers);
            }else{
                printf(">>> no handling defined for this signal-state... ignoring\n");
            }
            break;

        default:
            printf(">>> no handling defined for this signal... ignoring\n");
            return;
      }
      //Console message about state change
      printf(">>> recover to: ");
      printCurrentState();
      printf(">>> aborting the step that caused the exception\n");
      //jump back to the 'catch' clause  (end of sigsetjmp condition)
      siglongjmp(SBB_abort_step, 1);
}
```

Fig. 6. Code for the common exception handler

```
switch signal type
case SIGNAL 1:
    if current state = STATE1
        if condition for recovery 1
            change current state to recovery state 1
            //possibly no rollback is needed for some recovery states
        if condition for recovery 2
            change current state to recovery state 2
            rollback system variables to saved pre−transaction values
case SIGNAL 2: ... etc. ...
exit to end of main try block (using siglongjmp)
```

Of course the signal could occur in a state for which we did not model a recovery. In this case the signal is ignored. The recovery for a particular signal and state may also depend on further conditions. For example exception 1 depends on a count and takes a different recovery of the exception occurs several times (which may be a persistent attack). Each branch sets the appropriate recovery state in the state machine control data structure and also rolls back any variables that were part of a transaction where the recovery leaves that transaction. The handler exits via a siglongjmp which will return to the end of the sigsetjmp used in the amihn process (Figure4).

To demonstrate the code and signal handling we have executed it on a CHERI morello PC. The code for the barcode reading and processing states is 'seeded' with an invalid memory access (using a data value as a pointer) so that a SIGPROT exception can be generated. SIGALRM timeouts are easily simulated by not responding to the user interface simulation code. The console output provided by the demonstration program provides a record of the occurrence of exceptions.

8 Related Work

This section reviews relevant research on memory safety in capability-based hardware, exception handling and fault recovery mechanisms, the application of formal methods in safety-critical systems, and recent advances in code generation for safe exception handling.

Memory Safety and Capability-Based Hardware: Capability-based architectures have shown promise in enhancing memory safety and preventing unauthorized access at the hardware level. The CHERI (Capability Hardware Enhanced RISC Instructions) architecture is a notable approach that provides fine-grained memory safety guarantees by embedding capabilities into processor instructions. Watson et al. [19] present CHERI as a hybrid capability-system architecture that allows scalable compartmentalization, enhancing the security of software through pointer integrity and memory access control. Further refinements to CHERI's instruction set, as detailed in the technical report by Sewell et al. [14], enable efficient enforcement of memory safety in complex, high-assurance systems. These foundational works provide a hardware-level basis for secure exception handling, which our approach extends by focusing on application-specific recovery mechanisms and maintaining system consistency after exceptions.

Exception Handling and Fault Recovery in Safety-Critical Systems: Exception handling is essential for robust system behaviour in safety-critical applications, where maintaining a safe state during abnormal conditions is paramount. Julliand

and Perrouin [9] discuss the complexities of exception handling in formal methods, emphasizing the challenges of fault tolerance when designing for systems that must adhere to stringent safety standards. This work underscores the importance of domain expertise in designing tailored recovery mechanisms that respond effectively to error events. Unlike general-purpose error handling, safety-critical applications require a transactional approach to return systems to a consistent state, even under exceptional conditions. Our work builds on this by incorporating formal methods to model transactional behavior that can systematically manage exceptions in a capability-based hardware environment.

Formal Methods for Exception Handling and Safety Assurance: Formal methods, particularly Event-B, have proven valuable in the verification of safety-critical systems by enabling mathematical rigor in system design and error detection. However, traditional Event-B lacks explicit support for exception handling and fault recovery. Snook and Butler's [15] UML-B framework extends Event-B with UML-like state machines, allowing high-level modeling of system behavior while maintaining consistency in the presence of exceptions. Abdallah et al. [3] further adapt Event-B for handling exceptions, proposing a model for safe exception handling that ensures safety-critical systems can reliably transition to safe states. Our approach builds on these advancements by using UML-B to model system consistency and integrating exception-handling transactions that respond to non-completions and error states, contributing to the formal analysis of capability-based hardware systems.

Transactional Models and Consistency Recovery: Maintaining a consistent system state is crucial for safe exception handling in transactional models, particularly in distributed and embedded systems. Lamport's [10] classic work on distributed systems provides a foundational understanding of time and event ordering, which underpins transactional recovery in complex systems. Lynch and Tuttle [11] introduce the input/output automata model, highlighting the importance of input/output synchronization for achieving reliable consistency in safety-critical applications. These foundational models inform our approach by providing a theoretical basis for handling transactions and error states within closed systems, which we implement in our UML-B framework to manage exception recovery effectively.

Automatic Code Generation for Safe Exception Handling: Finally, translating formal models into executable code is a significant step toward implementing safe exception handling in practice. Abrial [4] outlines techniques for generating code from Event-B models, which can facilitate a direct path from formal design to application. Dalvandi [7] proposes SEB-CG tool for extensible automatic code generation from Scheduled Event-B (SEB), an extension of Event-B that augments models with control structures, to executable code in a target language. Mendes and Bensalem [12] extend this to safety-critical applications, demonstrating that automated code generation can preserve the integrity of exception-handling logic across various system states. Our work contributes to this body of research by developing an implementation of our UML-B model in C, with the goal of enabling future automatic code generation for exception handlers in CHERI-based systems.

Summary: The existing literature demonstrates the feasibility of using formal methods for exception handling in safety-critical systems, though there remains a gap in capability-specific recovery models that address the unique needs of memory-safe hardware like CHERI. By leveraging UML-B state machines and transactional recovery modeling, our approach advances the formal analysis of exception handling mechanisms tailored for capability-based systems, ultimately contributing to safer, more reliable embedded applications.

9 Future Work

We intend to develop a code generation tool that will convert our UML-B models into C code with the exception handling functions automatically produced and populated based on the transaction and exception transition detail in the models. In previous work we have developed more general Event-B to C code generation tools based on our Eclipse/Rodin plug-in tool framework. These tools can be extended to be more specific to UML-B state-machines with exception handling. We would also like to develop better tool support for the modelling proposed here. For example, special features within UML-B to model transaction states and exception transitions would make the modelling easier as well as providing better support for the code generation.

The methods discussed here assume the use of POSIX signals and associated Unix-based exception-handling infrastructure. The target hardware for the case study was a CHERI Morello PC running a variant of the BSD operation system. We would also like to support embedded systems which run on smaller real-time operating systems. We are now investigating a new case study using the Sonata development board which is based on an FPGA implementation of the Cheriot processor running CheriRTOS. A significant difference is that the POSIX signal infrastructure is not present in such systems and the exception handling concepts are closer to hardware device level.

So far we have not considered compartmentalisation in our methods. Cheri compartments enhance the memory-safe capabilities of the hardware providing better detection of suspicious behaviours. Compartmentalisation may require further modelling features/techniques and improved code generation. Compartments also provide the basis of a hierarchical unwinding of un-handled exceptions (analagous to unwinding of the call stack in some typical exception handling languages). We imagine this could be modelled using hiearchical statemachines.

10 Conclusions

Whereas the focus in hardware design is on generic mechanisms for detecting unusual potentially erroneous or suspicious behavior, the design of safe exception handling after the detection, is application or domain specific and therefore generic solutions are unattainable. Application engineers need supporting methods and tools to help them design and verify that recovery mechanisms do not violate the safety or security of the system. We provide a formal model-based analysis approach to achieve this by first modelling and verifying the system in the absence of exceptions and then adding the exceptional behaviour and appropriate recovery mechanisms. The modelling approach is based on discovering transactions which then suggest the necessary rollback of variables that were involved in the transaction. Usually we promote the use of safety-preserving refinement to incrementally develop the details of a system. However, a limitation of the approach is that exceptions cannot be added as a refinement stage and instead must be seen as a second stage within a single refinement. However, the first and second stages are a relatively simple/methodical progression of the same refinement level and therefore the consequences in terms of verifiability are not excessive. Furthermore, the detail of the system can be expanded as several refinements that each contain the two-stage approach. We have also demonstrated that an implementation can be derived from the formal models. While this is handwritten for now, it would be relatively straightforward to write a tool to automatically generate the code using our Eclipse-based code generation frameworks.

Acknowledgement:

This work is supported by HD-Sec project, which was funded by the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

References

1. CheriBSD website. https://www.cheribsd.org/, accessed: 2025-02-20
2. Galois and Free & Fair. The BESSPIN Voting System. https://github.com/GaloisInc/BESSPIN-Voting-System-Demonstrator-2019, accessed: 2024-02-07
3. Abdallah, A., et al.: A formal model for safe exception handling in safety-critical systems using event-b. International Journal of Critical Computer-Based Systems **7**(1), 64–85 (2017)
4. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
5. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International journal on software tools for technology transfer **12**(6), 447–466 (2010)
6. Brito, P.H.S., de Lemos, R., Rubira, C.M.F., Martins, E.: Architecting fault tolerance with exception handling: Verification and validation. J. Comput. Sci. Technol. **24**(2), 212–237 (2009)
7. Dalvandi, M., Butler, M.J., Fathabadi, A.S.: SEB-CG: Code Generation Tool with Algorithmic Refinement Support for Event-B **12232**, 19–29 (2019)
8. Dghaym, D., Hoang, T.S., Butler, M.J., Hu, R., Aniello, L., Sassone, V.: Verifying system-level security of a smart ballot box. In: Raschke, A., Méry, D. (eds.) Rigorous State-Based Methods - 8th International Conference, ABZ 2021, Ulm, Germany, June 9-11, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12709, pp. 34–49. Springer (2021)
9. Julliand, J., Perrouin, G.: Exception handling and fault tolerance in formal methods: From theory to practice. Formal Aspects of Computing **27**(3), 497–509 (2015)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)
11. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2**(3), 219–246 (1989)
12. Mendes, M., Bensalem, S.: Automatic code generation for safety-critical applications. IEEE Transactions on Software Engineering **42**(7), 650–666 (2016)
13. Salehi Fathabadi, A., Snook, C., Hoang, T.S., Thorburn, R., Butler, M., Aniello, L., Sassone, V.: Designing exception handling using event-b. In: Bonfanti, S., Gargantini, A., Leuschel, M., Riccobene, E., Scandurra, P. (eds.) Rigorous State-Based Methods. pp. 270–277. Springer Nature Switzerland, Cham (2024)
14. Sewell, P., et al.: Cheri instruction-set architecture. Technical report, University of Cambridge (2019)
15. Snook, C.F., Butler, M.: Uml-b: Formal modeling and design aided by uml. ACM Transactions on Software Engineering and Methodology (TOSEM) **15**(1), 92–122 (2006)
16. Snook, C.F., Butler, M.J.: UML-B: formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. **15**(1), 92–122 (2006)

17. Snook, C.F., Butler, M.J.: UML-B: A Plug-in for the Event-B Tool Set. In: Börger, E., Butler, M.J., Bowen, J.P., Boca, P. (eds.) Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5238, p. 344. Springer (2008)
18. Watson, R.N.M., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N.H., Davis, B., Gudka, K., Laurie, B., Murdoch, S.J., Norton, R.M., Roe, M., Son, S.D., Vadera, M.: CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA. pp. 20–37. IEEE Computer Society (2015)
19. Watson, R.N.M., et al.: Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In: IEEE Symposium on Security and Privacy. pp. 20–37. IEEE (2015)