Designing Exception Handling using Event-B

Asieh Salehi Fathabadi[0000−0002−0508−3066], Colin Snook[0000−0002−0210−0983], 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.
{a.salehi-fathabadi, cfs, t.s.hoang, robert.thorburn, m.j.butler, l.aniello, vsassone}@soton.ac.uk

Abstract. The design of exception handling is a complex task requiring insight and domain expertise to ensure that potential abnormal conditions are identified and a recovery process is designed to return the system to a safe state. Formal methods can address this complexity, by supporting the analysis of 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. In this paper, we use UML-B state machines to support the modelling of normal behaviour assisting the identification and handling of exceptions. This is followed by verification of exception handler recovery mechanisms using the built-in model checker and provers of the Event-B tool-set.

1 Introduction

Programming languages offer exception handling for responding to detected failures. Exception handling is a complex and error prone activity, and systematic reasoning is needed to identify and characterise exceptions. The formal analysis of the exceptional control flow provides a means to validate the exception handling design [6]. However, formal methods lack explicit support to specify exception handling behaviour. This paper proposes an approach to systematic reasoning about exception handling at the design level using the Event-B formal method.

Event-B [2] 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 Event-B toolkit, there is no explicit support for it. We use UML-B [9] to visualise the normal expected behaviour of a system and add support for handling exceptions in safety/security systems from the design level to the implementation. Our work is influenced by considering implementations on *capability hardware* which provides hardware level protection against incorrect memory access [11]. 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, e.g., out of bounds array access, or malicious attacks, e.g., 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 use of untrusted libraries.

We illustrate our approach using a Smart Ballot System (SBB) [1], 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 model in [8].

The paper, is structured as follows. Section 2 introduces Event-B and the SBB case study. Our proposed approach is outlined in Section 3 followed by application of our approach in the SBB case study, Section 5 and Section 5. Finally Section 6 concludes including summarising related works and directions for future works.

2 Background

Event-B [2] 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[1] tool set [3], 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 [10] extend the Rodin. UML-B [9] provides a diagrammatic modelling notation for Event-B in the form of state machines and class diagrams, which automatically generate the Event-B data elements.

SBB (Smart Ballot Box) [1] is to inspect a ballot paper by detecting a barcode, decode it and evaluate if the decoded contents verifies the paper. If the ballot is valid, then it can be cast into the storage box. Otherwise, the SBB rejects the paper, that will be ejected. The key function of the SBB is to ensure that only valid countable summary ballot documents that can be tabulated later are included in ballot boxes.

[1] The formal modelling tools used, are available as bundled installation packages via https://www.uml-b.org/Downloads.html.

3 Overview of approach

This section gives an outline of our suggested approach to analyse exceptions, and their associated recovery mechanisms, during systems modelling.

To consider exceptions at the formal design level, we propose the steps below:

- Build a state-machine to model normal behaviour (without exceptions). External controlled machinery can be modelled by other state-machines that synchronise via guards and synchronised transitions. Additional (ancillary) variables are added to model details maintained by the control system. Safety, security and other consistency properties are expressed as state invariants. Typically, these are expected values of machinery and ancillary variables in each control state.
- verify the normal-behaviour in the absence of exceptions (i.e. prove the invariant properties about the system).
- For each state in the state-machine, identify potential abnormal behaviour resulting in exceptions.
- For each exception, specify a corresponding recovery state to go to when the exception occurs in that occurrence state.
- Attempt to verify that the system invariants still hold even with abnormal behaviour.
- When an invariant cannot be verified it may be because the recovery state is inappropriate. For example, does not allow external machinery to return to an appropriate state. If so, change the recovery state or introduce new recovery states specifically to address this.
- When an invariant cannot be verified it may be because changes to ancillary variables need to be reverted due to the aborted process. If so, add roll-back actions to the exception handler for these specific cases.

These steps are presented through the SBB case study: Section 5 presents the normal behaviour model of the SBB as the first step and Section 5 presents the proposed exception handling approach within the rest of the above steps.

4 Case study: SBB normal-behaviour

Utilising UML-B, we model the SBB normal behaviour (without exceptions) as a state-machine (Figure 1).

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 2 ancilliary variables which are not shown in the state-machine but contained in the Event-B model. These are a count of the votes cast by the user (incremented by the transition USER_cast) and a count of the papers accepted by the roller (incremented by the transition ROLLER_accept_paper). The Waiting state contains two invariant properties which are expected to hold when the SBB is in the Waiting state:

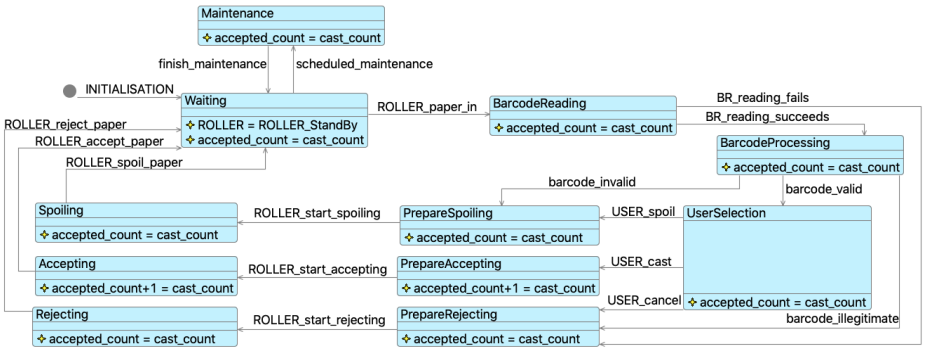

Fig. 1. State Machine, normal-behaviour SBB

– The roller should be in the state ROLLER_Standby so that it is ready to take another paper.
– The count of votes cast by the user should be the same as the count of papers accepted by the roller.

The invariants in the other states are needed to help the provers prove the second of these invariants. The proofs are automatically discharged by the Rodin provers.

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 which is encoded via a generated partition axiom as below. Each state (Waiting, BarcodeReading, ...), is specified as a *constant* and the set of states, SBB_STATES, are specified as an axiom using *carrier sets*:

```
@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 ROLLER_paper_in when @grd1: SBB =Waiting @grd2: ROLLER =ROLLER_StandBy
then @act1: SBB :=BarcodeReading @act2: ROLLER :=ROLLER_PaperIn end

event BR_reading_succeeds when @grd1: SBB =BarcodeReading @grd2: Exceptions =Normal
then @act1: SBB :=BarcodeProcessing end
```

5 Case study: SBB exceptional-behaviour

We consider two types of exception; an invalid memory access which could be caused by a security attack and a timeout when an external actor or machine does not provide a response. These exceptions are detected by the following interrupt signals:

– SIGPROT: a memory protection exception can be generated by capability hardware [11] when a pointer is used outside of its protected range (representing a possible memory attack).

– SIGALARM: timeout exception can be raised when an expected response from the environment fails to occur within a time limit.

The table below outlines these potential exceptions and their handling mechanisms within the SBB system:

Exception	Signal	States	Handling	Rollback
Memory protection error	SIGPROT	BarcodeReading, BarcodeProcessing, UserSelection	reject ballot	-
User does not enter selection	SIGALRM	UserSelection	reject ballot	-
Roller jammed	SIGALRM	Accepting	maintenance	cast count
Roller jammed	SIGALRM	Spoiling, Rejecting	maintenance	-

An attack on the software resulting in a SIGPROT interrupt is most likely to occur when the barcode reading subsystem is active and the safe handling response is to cancel and reject the ballot. When the user does not respond with a decision a SIGALRM interrupt is generated and again the safe handling response is to cancel and reject the ballot. It is also possible for the roller mechanism to malfunction and not confirm its completion, resulting in a SIGALRM interrupt. However, the ballot cannot be rejected as a response to this exception because it would involve the faulty roller. Instead a maintenance mode is entered to allow human intervention to correct the roller and reject the ballot. In the case where the roller was in the process of accepting a ballot an additional rollback action is needed because the users cast decision has already been counted but the ballot will now be rejected. To maintain consistency (and the invariant of the waiting state), the cast count must be decremented as part of the exception handling before the maintenance state is entered.

We extend our Event-B model to include the abnormal behaviour of exceptions as follows: SIGNAL is a set consisting of the types of interrupting signals (SIGPROT and SIGALRM). For each signal type, we specify a handling state that should be entered in order to recover from each state the signal could occur in. This is a constant partial function from SBB_STATES to SBB_STATES: A further constant function, signalHandling, gives the Handling function to be used for each signal. The values of these signal handling functions for the SBB are defined directly as axioms:

```
axm3: SIGPROT_Handling ={
    BarcodeReading ↦PrepareRejecting,
    BarcodeProcessing ↦PrepareRejecting,
    UseSelection ↦PrepareRejecting}
```

```
axm4: SIGALRM_Handling ={
    UseSelection ↦PrepareRejecting,
    Spoiling ↦Maintenance,
    Accepting ↦Maintenance,
    Rejecting ↦Maintenance}
```

The event exception_handler represents the occurrence of an exception:

```
event exception_handler any s
where @grd1: s ∈dom(signalHandling) @grd2: SBB ∈dom(signalHandling(s))
then @act1: SBB :=signalHandling(s)(SBB) end
```

Since the new event changes the state of the state-machine, the tools generate proof obligations to ensure that the state invariants concerning cast_count and accepted_count are respected. Most of these can be discharged by guiding the prover to show that there are no cases that enter the state containing that invariant or that the property was already true in the occurrence state. (We split the event into the two cases of s (s=SIGPROT and s=SIGALRM) and added theorems concerning the possible values of SBB and signalHandling(s)(SBB) for that case. This enabled the proofs to be automatically discharged). However, there remained one case (s=SIGALRM, SBB=Accepting) that was not proved and this corresponds to the case where we need to add a rollback of the cast_count. Hence the Event-B verification identifies any missing rollback actions and discovers the exact case where they are needed.

Other state invariants may identify inappropriate handling recovery states. For example, initially, we specified a recovery from a SIGPROT exception occuring in BarcodeReading directly to Waiting and imagine a transition from BarcodeReading to Waiting), the ROLLER would be left in the state ROLLER_PaperIn violating the safety invariant. We could not prove this unsafe design (the proof obligation could not be discharged) and we discovered a counter-example using the ProB model checker. Since the Roller is an external system it cannot be easily changed like the cast_count. Changing the recovery state from BarcodeReading to PrepareRejecting allows the ROLLER subsystem to reject the paper before the controller returns to Waiting, thereby maintaining a verified safe system.

6 Conclusion and future direction

This paper outlines an approach to analysing systematic exception handling and recovery at a formal systems design level. The proposed approach utilises UML-B state machines augmented by systematic identification and handling of exceptions in Event-B. We extend normal behavioural modelling and formal verification to address exceptional behaviour and recovery responses to bring the system back to a safe state according to system invariant properties. By considering exception handling in an abstract formal model of the complete system (i.e. a closed incorporating the controller and its environment including the controlled subsystems) we are able to verify that the chosen recovery mechanisms do not violate any safety properties. If we were to leave this verification to an implementation level (e.g. code) it would be more difficult to provide this level of verification since the controlled external environment would not be represented in a format that can be analysed.

To address the exception handling mechanism in different domains, related attempts have been presented before. [7] extends ERS (Event Refinement Structure) to introduce the interrupt and retry operators in Event-B. [5] and [4] formally define BPEL (Business Process Execution Language) compensation mechanisms using Event-B, focusing on the role of Event-B invariants during refinement. The research work presented in this paper, elaborates the existing state machine feature and automatic transformation to Event-B model to support explicit exception handling.

It is important to assure that the verified model is reflected in an implementation. Our future aim is to generate C code to implement the application functionality and exception handling based on signals as defined in UNIX systems. The implementation is derivable from our UML-B/Event-B models in a straightforward methodical way that could be mechanised with tool support.

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. Galois and Free Fair. The BESSPIN Voting System. https://github.com/GaloisInc/BESSPIN-Voting-System-Demonstrator-2019, accessed: 2024-02-07
2. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
3. 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)
4. Aït-Sadoune, I., Ameur, Y.A.: Formal Modelling and Verification of Transactional Web Service Composition: A Refinement and Proof Approach with Event-B. In: Thalheim, B., Schewe, K., Prinz, A., Buchberger, B. (eds.) Correct Software in Web Applications and Web Services, pp. 1–27. Texts and monographs in symbolic computation, Springer (2015)
5. Babin, G., Ameur, Y.A., Pantel, M.: Web Service Compensation at Runtime: Formal Modeling and Verification Using the Event-B Refinement and Proof Based Formal Method. IEEE Trans. Serv. Comput. **10**(1), 107–120 (2017)
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. Dghaym, D., Butler, M.J., Fathabadi, A.S.: Extending ERS for Modelling Dynamic Workflows in Event-B. In: 22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017. pp. 20–29. IEEE Computer Society (2017)
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. 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)
10. 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)
11. 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)