View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Heriot Watt Pure

A Rigorous Approach to Combining Use Case Modelling and Accident Scenarios

Rajiv Murali, Andrew Ireland, and Gudmund Grov

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh, UK

Abstract. We describe an approach to embedding a formal method within UML use case modelling while extending this requirements capture technique to also consider safety concerns. Our motivation comes from interaction with systems and safety engineers who routinely rely upon use case modelling during the early stages of defining and analysing system behaviours. Our chosen formal method is Event-B, which is refinement based and consequently has enabled us to exploit natural abstractions found within use case modelling. By underpinning informal use case modelling with Event-B, we are able to provide greater precision and formal assurance when reasoning about concerns identified by safety engineers as well as the subsequent changes made at the level of use case modelling. To achieve this we have extended use case modelling to include the notion of an *accident case*. Our approach is currently being implemented, and we have an initial prototype.

Keywords: Formal modelling, use cases, hazard analysis, model based, refinement, Event-B

1 Introduction

UML *use cases* are an informal notation for modelling the required behaviour of a system with respect to its operational environment. They are widely used and highly accessible. Our interest in use cases has developed through interactions with systems and safety engineers at BAE Systems[1]. Use case modelling provides a basis on which initial system behaviours can be defined and analyzed by safety engineers. Moreover, safety concerns that are identified by safety engineers are mitigated via changes to the use cases, e.g. corrections, inclusion of additional behaviours, etc. The lack of formality of use case modelling means that the process of analysis is typically review-based, and thus lacks the rigour that comes from formal methods, i.e. systematic identification of ambiguities, inconsistencies and incompleteness. Moreover, use case modelling does not provide any special mechanisms for representing the concerns of safety engineers, such as accident scenarios.

We present an approach which adds rigour to use case modelling via the Event-B [1] formal modelling notation. We also extend use case modelling to

[1] http://www.baesystems.com

include the notion of an *accident case*, which provides a way of representing accident scenarios. We selected Event-B because it promotes a layered style of formal modelling, where a design is developed as a series of abstract models – level by level concrete details are progressively introduced via provably correct refinement steps. Sometimes referred to as posit-and-prove, this style of modelling can increase the clarity of design decisions as well as simplify the complexity of the verification task. We argue that use case modelling exhibits a series of natural abstract models. Our approach exploits this mapping. That is, for a given use case we automatically generate a skeleton Event-B development. The completion of the development relies upon the user formalizing the details of their use case, e.g. constants, variables, pre- postcondition, invariants, assignments. Our prototype tool allows the user to specify their informal and formal descriptions of the a use case side-by-side. As a consequence inconsistencies and defects identified by formal verification can be mapped back onto the informal level.

The paper is structured as follows. Section 2 provides background on both use case modelling and Event-B, along with the details of a simple control system that we use to illustrate our approach. In Section 3 we introduce our notion of an accident case while in Section 4 the formalization of use case modelling is described. Section 5 deals with the mapping of our extended notion of use case modelling onto an Event-B development. Section 6 focuses on the benefits that formal verification at the level of Event-B brings to use case modelling and describes a prototype tool support. Related work and conclusion is described in Sections 7 and 8 respectively.

2 Preliminaries

2.1 Water Tank Controller Case Study

A case study of a water tank controller is used to describe our approach on formalising UML use cases. The design intent for the *controller* is to maintain the *water level* between the high (H) and low (L) limits of a water tank, as seen in Fig. 1. To achieve this intent, the controller communicates with two external components: sensor system and pump. The sensor system monitors the water level in the tank with respect to the high threshold (HT) and low threshold (LT) sensor readings. Based on these readings, the controller either activates or deactivates the pump. When the pump is active, its motor is switched *on* which increase the water level in the tank. When the pump is inactive the motor is switched *off* and the water level gradually decreases in the tank. The additional components *drain* is later introduced in Sect. 3.

2.2 Use Case Modelling

A use case model [2] is composed of a collection of *use cases* and *actors* that are of interest to the system being designed. A *use case diagram* is used to show the relationships between actors and use cases within a system, where-each use case

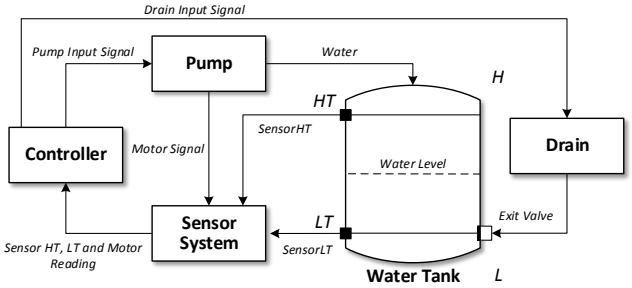
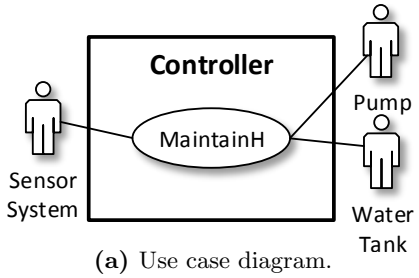


Fig. 1. A description of the Water Tank system.

captures the intended function of the system while the associated actors play a *role* in achieving them. An actor may represent a human, software or hardware component external to the system.

In Fig. 2a, a use case diagram for the water tank controller captures a use case MaintainH which denotes the desired functionality of the controller to maintain the water level below the H limit. The pump, sensor system and water tank are represented as actors that play a role in achieving the functionality of this use case. Each use case can be further detailed in a *use case specification*[2]. The use case specification for MaintainH can be seen in Fig. 2b.



(a) Use case diagram.

Use case: MaintainH
Pre-condition: Water level above HT.
Post-condition: Water level between L and HT.
Invariant: Water level between L and H.
Main Flow:
Trigger: Water level above HT.
B1. Sensor System activates sensor HT.
B2. Controller deactivates pump.
B3. Pump switches its motor off.
B4. Water level in tank gradually reduces.

(b) MaintainH - Use Case Specification.

Fig. 2. A Use Case Model of the Boiler System.

The use case specification captures a sequence of *action steps*, where each step is a discrete unit of interaction between an actor (human or machine) and the system. This sequence of steps is known as a flow and every use case has one *main flow*. The main flow describes a *sunny day scenario* where there are no failures or exceptions. The flow can initiate if its *trigger* condition is enabled. The specification also captures the *pre-*, *post-condition* and *invariant* for the use case. These can be described as a *contract* specified by the designer where given

[2] There is no standard template for use case specifications. The one used in the paper is kept simple as possible and is in common use in industry [3], with the exception of the invariant field.

the pre-condition, if the main flow of the use case executes and completes, then the result described by the post-condition must be achieved. The invariant must be maintained throughout the execution of the use case's flow.

2.3 Event-B

Event-B [1] is a refinement-based formalism for system-level modelling and analysis. An Event-B model is composed of *contexts* and *machines* where a context expresses the static information about the model while a machine represents the dynamic aspects. A machine models the state space by variables v, and state transition are modelled by events. The state variables v are constrained by laws specified by invariants $I(v)$. An event *evt* is of the following form:

$$evt \ \widehat{=} \ \textbf{\textit{where}} \ G(v) \ \ \textbf{\textit{then}} \ S(v, v') \ \textbf{\textit{end}}$$

In event *evt*, $G(v)$ specifies the enableness conditions of the event while $S(v, v')$ defines the state transition associated with the event. A dedicated *initialisation* event with no guards defines the states of the variables v at start-up. $S(v, v')$ contains several assignments that are supposed to happen simultaneously. Each assignment may take following forms: $v := E(v)$, $v :\in E(v)$, or $v : |P(v, v')$. The first form deterministically assigns the values of expression $E(v)$ to v, the second form non-deterministically assigns to v some value from $E(v)$. The last assignment form is the most general as it assigns to v some value satisfying the before-after predicate $P(v, v')$. A machine is consistent if its invariants hold at any given time. In practice, this is guaranteed by proving that the invariant is established by the *initialisation* and maintained by all its events.

An Event-B model supports refinement and allows details to be added in a stepwise manner. This helps manage the complexities of design and improve the degree to which verification can be automated. Correctness of the refinement is ensured by a set of generated proof obligations. The Rodin platform provides tool support for Event-B. It is based on the Eclipse framework and is further extensible via plugins.

3 Accident Scenarios in Use Case Modelling

Accidents or *losses* are considered early in the development of safety-critical systems. An accident can be defined as *"an undesired or unplanned event that result in a loss, including loss of human life or human injury, property damage, environmental pollution, mission loss, etc"* [4]. In the water tank controller case study, an accident (A1) that results in damage to the water tank is considered:

A1: Water level in tank exceeds the high (H) limit (physical damage to tank).

It is necessary for hazardous causal scenarios that lead to an accident to be considered along side the proposed system behaviour in order to agree upon appropriate design recommendations that may help mitigate them. We argue

that if the design intent, i.e. the expected behaviour of a system, can be captured and conveyed via use cases (e.g. MaintainH), then it should be possible for the unexpected scenarios that result in an accident to be conveyed in a similar manner. To our knowledge, this has not yet been considered for use cases and we have extended use case modelling to incorporate a use case type known as *accident case*.

3.1 Accident Case

Leveson [4] describes the cause of an accident as follows:

Hazard (Action) + Environmental Conditions (State) ⇒ Accident (Event)

A hazard is a system action that together with a particular set of worst-case environmental conditions, constitutes an accident. What constitutes a hazard depends on where the *boundaries* of the system are drawn. The use case model establishes the actors and system boundary via the use case diagram which determines what the designer has control over. If one expects the designer to create systems that eliminate or control hazards, then those hazards must be in their design space.

For the water tank controller, the designer has control over the increase and decrease of the water level in the tank (albeit not directly). A hazardous action would be for the water level to continue increasing in the water tank even after the water level has exceeded the high threshold (HT) limit and the controller has deactivated the pump. The cause of the accident for A1 can be written as follows:

Water level increases (Hazard) + Water level above HT and Pump signal is Off (Env. condition)
⇒ Water level exceeds H limit (Accident)

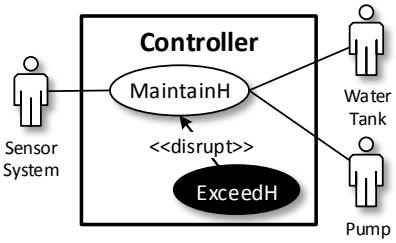
It is the role of the safety engineer to apply hazard analysis in order to determine the *hazardous causal scenarios* that can lead to the hazardous action, given the environmental conditions. We extend UML use cases with an *accident case* to help capture this hazardous causal scenario while relating them to use cases via a *disrupt* relationship. An accident case is defined as follows:

Definition 1 (Accident Case) *An accident case is a sequence of actions that a system or other entity can perform that result in an accident or loss to some stakeholder if the sequence is allowed to complete.*

The specification of an accident case contains the accident flow, where its trigger captures the *environmental condition* for the cause of an accident, while its final step captures the *hazardous control action*. The preceding steps capture the *hazardous causal scenario* identified from hazard analysis. The accident case may *disrupt* a use case by providing an alternate route resulting in an accident if allowed to complete.

The use case model of the water tank system is updated to introduce the accident A1 via an accident case ExceedH (Fig. 3a). It *disrupts* the MaintainH

use case by introducing its accident flow. The accident flow's trigger captures the environmental condition where the water level is above HT and the pump is *off*. The final step in the flow captures the hazardous control action of increasing the water level, while the preceding step capture a causal scenario leading to the hazardous action, i.e. where the motor remains switched on (Fig. 4b).



(a) Use Case Diagram.

Accident Case: ExceedH
Accident Flow:
Trigger: Water level above HT and Pump signal is Off.
F1. Pump's motor remains switched on.
F2. Water level in the tank increases.

(b) Accident Case Specification.

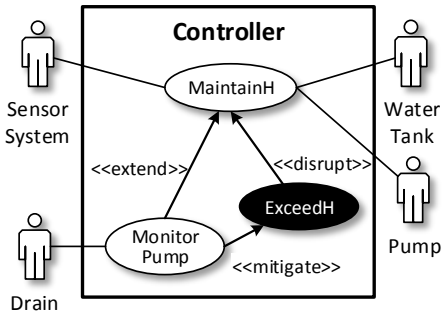
Fig. 3. Updated Use Case Model with Accident Case.

3.2 Safety Guided Design

The purpose of the accident case is to provide a means to communicate appropriate design recommendations after hazard analysis between system and safety engineers. One of the aims of a safety engineer is to ensure that no single fault or failure may result in an accident. In order to strengthen the safety of the the water tank, the system is extended with the additional component *drain* as seen in Fig. 1. The controller may activate the drain if it detects a *fault* where the motor remains switched *on* even after the pump has been deactivated. When the drain is active it opens an exit valve in the water tank which can drain to the water level to the low threshold (LT) limit.

In use case modelling *exceptional* behaviour can be introduced to the system via an *extension use case* [2]. An extension use case is used to describe how a system can respond to when things do not go as expected. The structure of extension use case specification is the same as a regular use case, however it is dependant on the base use case it extends. It places an *extension-point* between the steps of the base use case and if its trigger condition is *true* then its flow will initiate.

An extension use case MonitorPump is introduced in the use case model of the water tank controller (see Fig. 4a). It *extends* the MaintainH use case by mitigating the accident flow provided by ExceedH. An extension-point is placed between steps F1 and F2 (as seen in Fig. 6a) of the accident flow. This captures a relationship, *mitigate*, between the extension use case and accident case. The mitigate relationship ensures that if the accident case triggers then the extension use case will prevent its accident flow from completing. The extension-point captures a return-after step that returns the extension use case flow after the final step of the use case MaintainH.



(a) Updated Use Case Diagram.

Accident Case: ExceedH
...
F1. Pump motor remains switched on.
extension-point: **MonitorPump** [return-after: B4]
F2. Water level in the tank increases.

(b) Updated Accident Case Specification.

Extension Use Case: MonitorPump
Pre-condition: Pump is deactivated and motor remains switched on.
Post-condition: Water level at LT.
Extension Flow:
Trigger: Pump is deactivated and motor remains switched on.
E1. Controller activates the drain.
E2. Drain opens the exit valve.
E3. Water level in the tank is drained.

(c) Extension Use Case Specification.

Fig. 4. Use case model updated with MonitorPump extension use case.

4 A Formal Use Case Specification

We aim to perform formal verification of use cases by automatically transforming its specification to Event-B. To do so, we first introduce a *formal use case specification* to represent the informal use cases MaintainH (Fig. 2b), ExceedH (Fig. 4b) and MonitorPump (Fig. 4c) formally written with Event-B's mathematical language.

Use case: UC	
A: $i :: T(i)$ $j :: T(j)$	**Pre-condition:** $P(i)$ **Post-condition:** $Q(i)$ **Invariant:** $I(i)$
	Main-flow: *Trigger* : $R(i,j)$ $S_1 .\ E_1(i,j,j')$ $\vdots$ $S_n .\ E_n(i,i',j)$

(a) Generic use case UC.

Use case: MaintainH	
Water Tank: $wl :: wl \in \mathbb{N}$ $H :: H > HT$ $HT :: HT > LT$ $LT :: LT > L$ $L :: L < H$ $DEC ::$ $DEC \in (H - HT)..(HT - L)$ **Sensor System:** $senHT :: senHT \in BOOL$ **Controller:** $pump :: pump \in BOOL$ **Pump:** $motor :: motor \in BOOL$	**Pre-condition:** $wl > HT$ **Post-condition:** $wl \geq L \wedge wl \leq HT$ **Invariant:** $wl \in L..H$
	Main-flow: *Trigger* : $wl > HT$ $B1.\ senHT := TRUE$ $B2.\ pump := FALSE$ $B3.\ motor := FALSE$ $B4.\ wl := wl - DEC$

(b) MaintainH.

Fig. 5. Formal use case specifications.

Fig. 5a shows a formal use case specification of a generic use case, UC, where its pre-condition, post-condition, and invariant describe constraints on the variables i. These variables i model a state space associated to the domain of actor, A, that plays a role in UC. Variables that are written in capital indicate that they are 'static', i.e. they cannot be modified by the use case's flow. The variables along with their types $T(i)$ for each associated actor is provided on the left-hand side of the formal use case specification. This is done to help guide the designer when detailing the right-hand side of the formal use case specification which provides the pre-condition, post-condition, invariant and flow.

The flow contain steps $S_1..S_n$ that capture *actions* that describe state transition to the variables of UC. The flow is expected to reveal more of the state

space, which we model by variables j. The variables i and j are kept distinct as the steps S_1 to S_{n-1} capture actions, $E(i, j, j')$, that describe some state transitions to variable j. However, we work on the assumption that there will be some step, S_n, that will capture the necessary action, $E(i, i', j)$, that modifies the variable i in order to achieve the post-condition $Q(i)$.

This template is applied to the use cases of the water tank controller. The formal use case specification can be seen for MaintainH (Fig. 5b), ExceedH (Fig. 6a) and MonitorPump (Fig. 6b). In MaintainH, the actor, Water Tank, captures the variable wl that denotes the water level in the tank. Its state space represents a numerical value hence the type $wl \in \mathbb{N}$. The pre-, post-condition and invariant capture the necessary constraint on wl with respect to the limits of the water tank H, HT, LT, L. The limits are written in capital indicate that they are 'static' and they cannot be modified by the use case's flow. The static variable DEC is a discrete representation of the decrease in the water level by the water tank.

Accident Case: ExceedH	
Pump: $INC :: INC \in (LT-L)..(H-LT)$	**Main-flow:** *Trigger* : $wl > HT \wedge Pump = FALSE$ $F1.\ motor := TRUE$ *extension-point*: MonitorPump [return-after: B4] $F2.\ wl := wl + INC$

(a) ExceedH.

Extension Use Case: MonitorPump	
Controller: $drain :: drain \in BOOL$ **Drain:** $valve :: valve \in BOOL$ $DRN :: DRN = LT$	**Pre-condition:** $motor = TRUE \wedge pump = FALSE$ **Post-condition:** $wl = LT$
	Main-flow: *Trigger* : $motor = TRUE \wedge pump = FALSE$ $E1.\ drain := TRUE$ $E2.\ valve := TRUE$ $E3.\ wl := DRN$

(b) MonitorPump.

Fig. 6. Formal use case specifications.

The flow of MaintainH reveals more of the state space which are modelled by variables *pump*, *senHT* (sensor reading for high threshold limit) and *motor*. They are of type $BOOL$, where $TRUE$ indicates 'active' while $FALSE$ indicate 'inactive', i.e. when $pump = TRUE$ denotes the pump is active. The steps B1 to B3 modify these variable, while the final step B4 captures the necessary modification to the variable wl that should achieve the post-condition. The formal use case specification helps bring the benefit of precision and clarity when detailing the use case while enabling the designer to relate informal and formal notation.

The static variable DRN in ExceedH indicates the level at which the water level will be drained in the water tank. In this instance, the designer has set the drain to the low threshold limit (LT). Once the use cases are specified formally, it is possible to map their content to a formal model via refinement for purpose of verification.

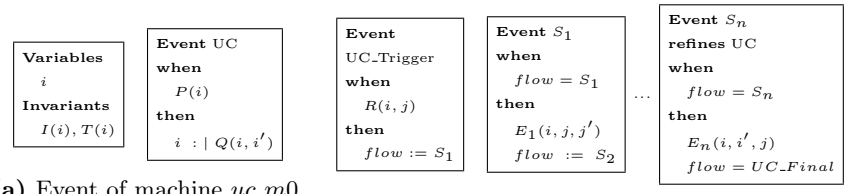
5 Mapping Use Cases to Event-B via Refinement

In this section we describe how the formal use case specification of UC is *mapped* to an Event-B model. We then apply this mapping to the MaintainH use case of the water tank controller. ExceedH and MonitorPump are also taken into account

as they are dependent on MaintainH by providing alternate scenarios to its flow. The verification performed for the use cases from their corresponding Event-B models is discussed in Sect. 6.

5.1 Generic Use Case

We consider the use case specification to have a *contract* and *body*. The contract is composed of the pre-condition, post-condition and invariant, while the body contains the flow. Our mapping introduces the contract of UC in an abstract Event-B machine, uc_m0, while the body is introduced by its refinement in uc_m1. The contract is the specification of the body and we use refinement to relate them accordingly.



(a) Event of machine uc_m0.

(b) Events of refined machine uc_m1.

Fig. 7. Refinement of UC's use case specification

UC's Contract: In *uc_m0*, we introduce a key abstraction of *what* is to be achieved by the use case without specifying *how*. UC's pre- and post-condition are modelled by an event, UC, as its guard and action, respectively (see Fig. 7a). This event introduces a state transition in the model where given the pre-condition it is possible for the post-condition to be achieved. The event does not reveal *how* this is done and emphasises only on *what* is achieved. The variables i associated to the pre- and post-conditions are introduced along with its type $T(i)$ and invariant $I(i)$. The invariants ensures that the state transition of event UC maintains these constraint.

UC's Body: The machine uc_m0 is refined to uc_m1 to introduce UC's flow which describes *how* the use case achieves and maintains its *contract*. Each step of UC's flow along with its trigger is mapped to a corresponding event in um_m1 (see Fig. 7b). The variables j (revealed by the flow) and its associated types $T(j)$ are introduced in this refinement. The flow is mediated between steps S_1 to S_n by an auxiliary variable s that act as a program counter. The event UC_Trigger initiates the use case's flow ($s := S1$) given the trigger condition $R(i, j)$. The steps S_1 to S_{n-1} capture actions that modify the variables j, however the final step S_n capture the necessary modification on variables i and refines the abstract event UC. By proving the refinement we show that the contract of is satisfied by the flow. Moreover, it allows to reason about the contract in the initial model while temporarily ignoring how they are implemented till refinement.

To illustrate the execution we use an *event-flow diagram* (see Fig. 8). The event-flow diagram is read from left-to-right, where each line is an execution of an

event in the formal model. The diagram helps to show the refinement of events, where any event in **bold** indicates it refines an event from the abstract model. Suppose if the name of the event in the abstract model is different to that of its concrete counter part, then we show the name abstract event in parenthesis, e.g. S_n (UC).

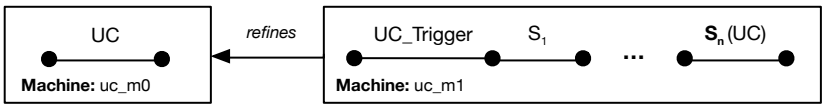


Fig. 8. Event-flow diagram of UC in Event-B.

5.2 Water Tank Controller

The mapping from UC is applied to MaintainH to produce two layers of refinement mh_m0 and mh_m1. The MaintainH use case has a disrupt relationship which introduces the accident flow of ExceedH in mh_m1. This accident flow captures a extension-point that introduces the extension use case MonitorPump. The mapping from UC is applied to MonitorPump to introduce its contract in mh_m1 while its body (containing the extension flow) is introduced via refinement in mh_m2. The event-flow diagram for MaintainH's Event-B model can be seen in Fig. 10.

Initial Model: In mh_m0, the contract of MaintainH use case is introduced. The event $MaintainH$ models the pre- and post-condition as its guard ($wl > HT$) and action ($wl :| wl' \geq L \ \wedge \ wl' \leq HT$), respectively. The event captures *what* the use case MaintainH achieves, by allowing the water level to be reduced between the L and HT limits if it exceeds the HT limit. The variable wl associated with the pre- and post-condition is introduced along with its type $wl \in \mathbb{N}$ and invariant $wl \in L..H$. The static variables (written in capitals) are captured in a context component of the Event-B model, which can be *seen* by the machine.

First Refinement: In mh_m1, the flow of MaintainH is introduced which describes *how* the water level is reduced when it exceed the HT limit. The flow's trigger and steps B1 to B4 are mapped to events (as seen in machine mh_m1 in Fig. 10). The variables *pump*, *senHT* and *motor* revealed by the flow is introduced in this machine along with their types. The events $B1$ to $B3$ perform actions that modify these variables according to the main flow. The final event $B4$ describes the decrease in the water level ($wl := wl - DEC$) after the motor has been deactivated. This final event $B4$ refines the abstract event $MaintainH$.

The accident flow of ExceedH is also introduced in this refinement due to the disrupt relationship with MaintainH. The accident flow of ExceedH is introduced by the events $ExceedH\_Trigger$, $F1$ and $F2$. The event $ExceedH\_Trigger$ may initiate the accident flow at any point during the flow of MaintainH, provided its trigger condition is *true*. If the accident flow is allowed to terminate, then the

water level would exceed above the H limit which would violate the invariant ($wl \in L..H$).

The extension-point between the steps F1 and F2 introduces the extension use case MonitorPump by two events: $MonitorPump$ and $MonitorPump\_False$. An auxiliary boolean variable ext is used to insert both these events between events $F1$ and $F2$ (see Fig. 9). The event $MonitorPump$ captures the contract of the extension use case. If the motor remains active while the pump has been deactivated, then the water level is reduced to the low threshold (LT) and the flow is returned after the final B4 of the main flow. The event $MonitorPump$ refines the abstract event $MaintainH$, since the post-condition describes a desired stated ($wl = LT$) on the abstract variable wl. On the other hand, $MonitorPump\_False$ captures the negation of the extension use case's precondition as its guards. Suppose if the extension use case's precondition is not *true*, then this event returns the flow back into the accident flow. This requires the right fault condition to be specified by the extension use case in order to prevent the accident flow from completing.

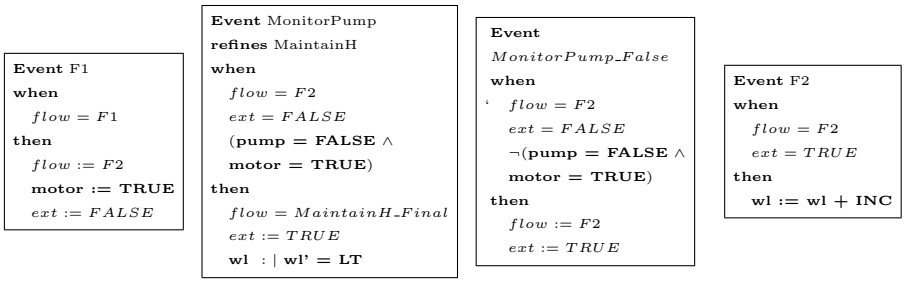


Fig. 9. Extension-point inserted between steps F1 and F2 of ExceedH.

Second Refinement: In the final refinement, we introduce the body of the extension use case MonitorPump. The body reveals more of the system by introducing the variables *drain* and *valve*. The flow of MonitorPump is mapped to events $MonitorPump\_Trigger$, $E1$, $E2$ and $E3$. It introduces the scenario of the controller activating the drain which opens an exit valve on the water tank. This is done to reduce the water level if the controller detects the motor remains active even after the pump being deactivated. The action of the final event $E3$ drains the water level in the tank, which refines the now abstract event $MonitorPump$.

6 Verification and Tool Support

6.1 Generic Use Case

UC's Contract: The main mathematical judgement in the initial model of the use case is to determine whether the invariant, $I(i)$, is guaranteed to be maintained by what is achieved, $Q(i, i')$, by event UC. Proving this ensures that the flow introduced by refinement must meet this contract.

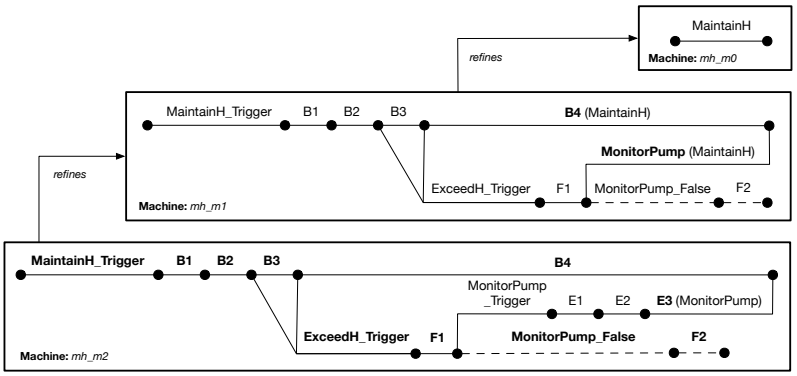


Fig. 10. Event-flow diagram of the MaintainH's corresponding Event-B model.

UC's Body: The model checks whether UC's flow achieves the post-condition $Q(i)$, given the pre-condition $P(i)$. We are required to prove that the pre-condition must be maintained before the execution of step S_1 to S_{n-1}. The following invariant is automatically introduced to help prove this:

$$\forall s \cdot s \in \{S_1, .., S_n\} \wedge flow = s \Rightarrow P(i) \tag{1}$$

The event S_n refines the abstract event UC. This refinement must prove that the abstract event's behaviour of what the use case achieves, corresponds to the concrete behaviour of its flow.

Mitigate: For any accident flow introduced via the disrupt relationship, there is expected to be a extension-point between its steps. As discussed in Sect. 3, the extension-point in an accident flow is required to ensure the steps after its point of insertion can never be executed. In Event-B, we introduce invariants that negate the guards of the events that model these steps. This allows the model to automatically prove that the steps can never be enabled, i.e. the accident case will be unable to complete.

6.2 Water Tank Controller

Initial Model In the initial model we are able prove what is achieved by the MaintainH use case's post-condition, i.e. the reduction of water level, is within constraints of its invariant $wl \in L..H$. The following proof is generated by the model and automatically proved:

$$wl' \geq L \wedge wl' \leq HT \vdash wl' \in L..H$$

First Refinement The model is refined to introduce the flow of MaintainH which ensures the contract introduced in the abstract model is preserved. The invariant (1) is applied to MaintainH to automatically produce the following invariant:

$$\forall s \cdot s \in \{B1, B2, B3, B4, F1, F2\} \wedge flow = s \Rightarrow wl > HT$$

This is used to ensure the pre-condition is *true* before the flow can execute. The event $B4$ refines the abstract event $MaintainH$. A proof obligation is generated to ensure the action of event $B4$ is maintained between L and HT limits, given that the water level is above HT. The following proof is generated and automatically discharged:

$$flow = B4 \vdash wl - DEC \geq L \wedge wl - DEC \leq HT$$

The event $MonitorPump$ (which represents the extension-point) also refines the abstract event $MaintainH$. The model checks what the extension use case achieves, i.e. the water level reduced to LT, corresponds to what is achieved by $MaintainH$ event in the abstract model. In addition, the model proves that the accident flow of ExceedH is not allowed to complete. The guards of the events after the extension-point, $MonitorPump\_False$ and $F2$, are negated and introduced as invariants (3) and (2) respectively.

$$\neg(flow = F2 \wedge ext = FALSE \wedge \neg(pump = FALSE \wedge motor = TRUE)) \quad (2)$$

The model is able to prove these events invariants as the accident flow introduces the necessary conditions for the extension use case, MonitorPump to trigger instead of MonitorPump_False.

$$\neg(flow = F2 \wedge ext = TRUE) \quad (3)$$

Second Refinement The model is able prove that the extension use case's flow which drains the water level correspond to MaintainH, as the drain is set to the LT limit of the water tank ($DRN = LT$) which is between the L and HT limits.

6.3 UC-B Tool Support

Our approach is currently being implemented as a tool UC-B[3] (Use Case Event-B) for the Rodin platform (Fig. 11). It supports the authoring and management of UML use case models with the inclusion of accident cases. It allows the use case specifications to be detailed with Event-B mathematical language and provides support for the automatic generation of Event-B models from a target use case. The generated Event-B models are immediately subjected Event-B's verification tools (syntax checks and provers) that run automatically providing an immediate display of problems. Our aim is for inconsistencies in the Event-B models to reflected back to their parent use case model. We have considered two other case studies, anti-lock braking system (ABS) and sense and avoid (SAA), that fit the same control pattern as the water tank controller.

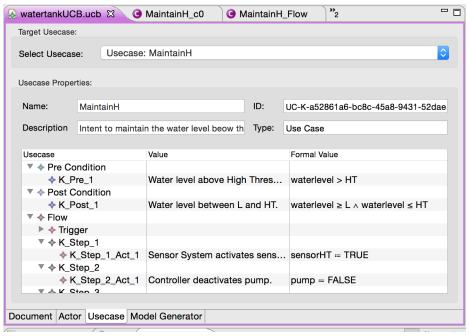


Fig. 11. UC-B tool on Rodin.

[3] Tool information can be found at https://sites.google.com/site/rajivmkp/uc-b

7 Related Work

A majority of the related work in the area of capturing negative or forbidden scenarios for UML use case have focused on security concerns, whereas our focus has been towards considering safety concerns from potential accidents. Ellison et al. [5] introduce *intruders* and *intrusion scenarios* in their case study as part of a large-scale distributed health care system. The intrusion scenarios are similar to accident flow, but they do not provide a diagrammatic notation, a specification, or guidelines for what constitutes an intrusion scenario. McDermott and Fox [6] propose *abuse cases* which focus on security requirements and their relation to design and testing. They capture the abuse case and regular use cases in separate use case diagrams. This differs from our approach where we provide relationships between accident case and regular use cases in the same use case diagram. Potts [7] introduce *obstacles* from goal-oriented requirements engineering (KAOS). These obstacles are exceptional condition that prevent goal fulfilment. In relation to our work with UML use cases, a general goal can be related to a use case while an obstacle can be related to our use of accident case.

Several groups have investigated a rigorous approach to capturing UML use case [8–10]. In comparison the novelty of our approach comes from the use of refinement to introduce key abstractions that are captured naturally by the structure of the use case specification and its relationships to other use cases. In [8], Soussa and Russo provide a mapping from the flow of a use case to operations in B. They rely upon the flow to be written in accordance to a transaction pattern between the actor and the system as follows: (1) an actors request action, (2) a system data validation action, (3) a system expletive action, and finally (4) a system response action. We consider this pattern would require the designer to focus more on the solution rather than understanding the problem domain, which steps away from some of the benefits and simplicity of using UML use case. In [10], Whittle presents a precise notation using interaction overview diagrams (IOD) for specifying use cases based on three levels of abstraction: use case charts, scenario charts and interaction diagrams. The motivation to this approach is similar to ours which also consider the use of negative scenarios, however we have focused on adding rigour to the existing textual specification of use cases.

A rigorous approach to requirements capture techniques such as Problem Frames [11] and KAOS [12] have also been considered. However, these techniques are not a de-facto industry standard and their notations don't share the same popularity as UML use case, which we consider a major disincentive when convincing industry practitioners to adopt a more rigorous approach during requirements capture.

8 Conclusion and Future Work

The work presented here is part of an on-going effort to help in the industrial adoption of formal methods and of a more specific effort to consider safety concerns. We have extended UML use cases to consider potential accidents via the

use of accident cases, that is aimed to improve communication between system and safety engineers. For the purpose of formal analysis of use cases, we have provided a formal use case specification to detail use cases with Event-B's mathematical language. From this, we use the structure and relationship of use cases to derive a natural abstraction when mapping them to an Event-B model. Proof automation is possible which helps identify inconsistencies and defects in the formal model that can be mapped back onto the use case model. Our tool implementation supports the authoring and management of UML use case models on the Rodin platform while enabling automatic generation of Event-B models from a target use case.

For future work, we are investigating links between the hazard analysis techniques with our notion of an accident case. Our tool is currently being extended to support traceability between the generated Event-B model and its parent use case. Patterns of inconsistencies identified by proofs could be used to meaningfully guide an engineer while detailing use cases.

Acknowledgements: The first author was supported by an Industrial CASE studentship funded by the EPSRC and BAE Systems (EP/J501992), while the second and third authors was partially supported by EPSRC grant EP/J001058. We also would like to thank Benjamin Gorry, Rod Buchanan and Paul Marsland from BAE Systems.

References

1. Abrial, J.R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
2. Booch, G., Rumbaugh, J., Jacobson, I.: Unified modeling language. (1997)
3. Arlow, J., Neustadt, I.: UML 2 and the unified process: practical object-oriented analysis and design. Pearson Education (2005)
4. Leveson, N.G.: Safeware: system safety and computers. ACM (1995)
5. Ellison, R.J., Linger, R.C., Longstaff, T., Mead, N.R.: Survivable network system analysis: a case study. IEEE software **16**(4) (1999) 70–77
6. McDermott, J., Fox, C.: Using abuse case models for security requirements analysis. In: Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual, IEEE (1999) 55–64
7. Potts, C.: Using schematic scenarios to understand user needs. In: Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques, ACM (1995) 247–256
8. Russo Jr, A.G., de Sousa, T.: Starting b specifications from use cases. In: Abstract State Machines (ASM), Alloy, B and Z Conference. (2010)
9. Klimek, R., Szwed, P.: Formal analysis of use case diagrams. Computer Science **11** (2010) 115–131
10. Whittle, J.: Precise specification of use case scenarios. In: Fundamental Approaches to Software Engineering. Springer (2007) 170–184
11. Jackson, M.: Problem frames: analysing and structuring software development problems. Addison-Wesley (2001)
12. Ponsard, C., Dieul, E.: From requirements models to formal specifications in b. ReMo2V **241** (2006)