Designing Critical Systems Using Hierarchical STPA and Event-B

Asieh Salehi Fathabadi(✉), Colin Snook, Dana Dghaym, Thai Son Hoang, Fahad Alotaibi, and Michael Butler

ECS, University of Southampton, Southampton, UK
{a.salehi-fathabadi,cfs,d.dghaym,t.s.hoang, F.A.Alotaibi,m.j.butler}@soton.ac.uk

Abstract. In the design of critical systems, it is important to ensure a degree of formality so that we reason about safety and security at early stages of analysis and design, rather than detect problems later. Influenced by ideas from STPA we present a hierarchical analysis process that aims to justify the design and flow-down of derived critical requirements arising from safety hazards and security vulnerabilities identified at the system level. At each level, we verify that the design achieves the safety/security requirements by backing the analysis with formal modelling and proof using Event-B refinement. The formal model helps to identify hazards/vulnerabilities arising from the design and how they relate to the safety accidents/security losses being considered at this level. We then re-apply the same process to each component of the design in a hierarchical manner. Thus we use ideas from STPA, backed by Event-B models, to drive the design, replacing the system level requirements with component requirements. In doing so, we decompose critical requirements down to components, transforming them from abstract system level requirements, towards concrete solutions that we can implement correctly so that the hazards/vulnerabilities are eliminated.

Keywords: Event-B · Hierarchical · STPA · Safety · Security

1 Introduction and Motivation

Safety and security are key considerations in the design of critical systems. Systems Theoretic Process Analysis (STPA) [11] is a method for analysing safety of systems that involve control components to identify potential hazards. STPA-Sec adapts STPA for use in systems to identify potential security losses.

STPA is methodical but not rigorous in that it provides systematic guidance on what to consider but relies on human judgment to assess the effect of incorrect actions. Formal techniques such as Event-B [2], on the other hand, are not methodical in that they rely on human expertise about modelling choices, but can then provide a rigorous assessment of the properties of the model through formal verification. In previous work [4,8,9] we have explored the combination

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
U. Glässer et al. (Eds.): ABZ 2023, LNCS 14010, pp. 220–237, 2023.
https://doi.org/10.1007/978-3-031-33163-3_17

of STPA and STPA-Sec with formal modelling methods to exploit the synergy between informal analysis and rigorous formal verification. While this combination is both methodical and rigorous, its scalability is limited by the lack of systematic support for an incremental approach. An incremental approach supports scalability by allowing developers to factorise the analysis of complex systems in stages rather than addressing the analysis in a single stage. Event-B already supports incremental formal development through abstraction and refinement in formal modelling. However, the STPA part of the combined STPA/Event-B approach lacks systematic support for incremental informal analysis of safety and security.

In this paper we address the limitation on scalability of the STPA/Event-B combination by adopting an abstraction-based incremental and hierarchical approach to informal analysis of critical requirements. We call the approach Systematic Hierarchical Analysis of Requirements for Critical Systems (SHARCS). Previous works present the combination of STPA and STPA-Sec with Event-B and support requirements analysis at a single abstraction level, while SHARCS is inspired by STPA and proposed a novel incremental approach. To our knowledge, an abstraction-based incremental and hierarchical approach to STPA control structure analysis has not previously been considered.

While STPA requires consideration of a complete closed system, it is based on the concrete design of the system. In contrast, by shifting the boundaries of the component sub-system being considered, we abstract away from the lower level internal details and analyse the constraint requirements of control abstractions before refining these with the next level of sub-component design.

We utilise the Event-B modelling language and the Rodin tool set for formal modelling to verify and validate the SHARCS analysis. Event-B with its associated automatic verification tools, is ideal for the detailed modelling of each level because it supports abstract modelling of systems with progressive verified refinements. One of the most difficult tasks in constructing an Event-B model consisting of several refinements is finding useful abstractions and deciding the progressive steps of refinement; the so-called *refinement strategy*. From an Event-B perspective therefore, SHARCS helps the modeller by providing a method to guide the refinement strategy. Although the Event-B supports refinement-based modelling, the modeller needs to make decisions about which system requirements to model at different stages of refinement. SHARCS helps the modeller to derive the requirements for different refinement levels; the requirements are driven by the incremental introduction of system components into the analysis.

Our aims are twofold. Firstly the hierarchical approach to the analysis introduces component sub-systems that are designed to address and mitigate insecure control actions that have been revealed by the analysis of the parent component. As a result we provide an analysis method for deriving component sub-system level requirements from parent system level requirements. Secondly the analysis provides a traceable argument that the design satisfies the higher level requirements while addressing safety hazards and security vulnerabilities. For example, consider a high-security enclave consisting of several components including a

secure door, a card reader and a fingerprint reader. The system-level security requirement is that only authorised users are allowed to access the enclave; a derived requirement on the fingerprint component is that it should determine whether a user fingerprint corresponds to the fingerprint stored on an access card. Figure 1 illustrates the derivation of the component requirements from the system-level requirement in a hierarchical manner. The abstraction-based hierarchical approach is a key contribution of this paper.

We demonstrate our SHARCS approach in an access control system, Tokeneer. The artifacts from the case studies are available to download from https://tinyurl.com/SHARCS-dataset.

The paper is structured as follows: Sect. 2 provides background on STPA, the Event-B formal modelling language, applied tools and introduction to our case study: the Tokeneer access control system. Section 3 presents an overview of applying the approach to the case study. Section 4 and Sect. 5 present the approach in more detail, using our experience of applying it to the Tokeneer case study. Section 6 discusses related and previous work. Finally Sect. 7 concludes and describes future work.

2 Background

2.1 Systems Theoretic Process Analysis (STPA)

STPA [11] is a hazard analysis method which can be applied to systems involving control structures. The hazardous conditions are identified by considering the absence, presence or the improper timing of control actions. The process is followed by identifying causal factors for unsafe control actions.

While STPA is used for safety problems, STPA-Sec [17] extends STPA to include security analysis. Similar to STPA, STPA-Sec identifies losses and system hazards, or in this case, system vulnerabilities. STPA-Sec also examines the system control structure and identifies the insecure control actions instead of the unsafe actions.

2.2 Event-B

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 s, constants c, and axioms $\mathsf{A(c)}$ that constrain the carrier sets and constants. Machines contain variables v, invariant predicates $\mathsf{I(v)}$ 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. In general, an event e has the following form, where t are the event parameters,

$G(t, v)$ is the guard of the event, and $v := E(t, v)$ is the action of the event: $e ==$ **any** t **where** $G(t,v)$ **then** $v := E(t,v)$ **end**

An Event-B model is constructed by making progressive refinements starting from an initial abstract model which may have more general behaviours and gradually introducing more detail that constrains the behaviour towards the desired system. This is done by adding or refining the variables of the previous abstract model and modifying the events so that they use the new variables. Each refinement step is verified to be a valid refinement of the previous step. That is, the new behaviour must have been possible in the abstract model according to the given relationship between the concrete and abstract variables. Event-B is supported by the Rodin 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, and validating models with simulation-based approaches.

In this paper we make extensive use of the animation plug-in tools that extend the Rodin toolset; ProB [10] is an animator and model checker for the Event-B. Scenario checker [15] is an animation tool that we developed for validating systems by recording and replaying scenarios. It extends ProB to support two new functionalities: a 'run to completion' style execution of controller events, and a record/replay style user interface for running test scenarios.

2.3 Tokeneer Case Study

Our case study in this paper is the Tokeneer system. The Tokeneer system [14] consists of a secure enclave and a set of system components, some housed inside the enclave and some outside. The ID Station interfaces to four different physical devices: fingerprint reader, smartcard reader, door and visual display. The primary objective is to prevent unauthorised access to the Secure Enclave. The requirements include (1) authenticating individuals for entry into an enclave and (2) controlling the entry to and egress from an enclave of authenticated individuals. The door has four possible states: the cross-product of *open*/*closed* and *locked*/*unlocked*. A card identifies a particular user using a fingerprint mechanism. If a user holds a card that identifies them via fingerprint matching, they are permitted in the enclave. Hence cards should only be issued to permitted users. A successful scenario involves: arrival of a permitted user at the door who then presents a card on the card reader and a matching finger print at the fingerprint reader. The system will then unlock the door allowing the user to open it and enter the enclave.

3 Overview of Systematic Hierarchical Analysis of Requirements for Tokeneer

Our approach is based on the use of a control action analysis (that borrows some ideas from STPA) in conjunction with formal modelling and refinement (using Event-B) to analyse the safety and security of cyber-physical systems by

flowing down system-level requirements to component-level requirements. Since we propose a generic approach for both safety and security, we simply use the term *failure*. SHARCS approach consists of three phase: system level analysis and abstract modelling (Sect. 4), component level analysis and refinement modelling (repeated for each identified sub-system, Sect. 5), and consolidation phase. In this section we presents the outputs from the the final *consolidation* stage (Figs. 1 and 2) of the SHARCS process. We believe they give a good overview of the steps used in the analysis and presented in the next two sections.

The hierarchical component design of the Tokeneer system is illustrated in Fig. 1. Starting from the system level, the analysis of that system leads us to the outline design of the next level in terms of sub-components and their purpose. Some of these components require further analysis (those shown with title and purpose) while others (shown with only a title) are assumed to be given, and are therefore only analysed in so far as they are used by their sibling components.

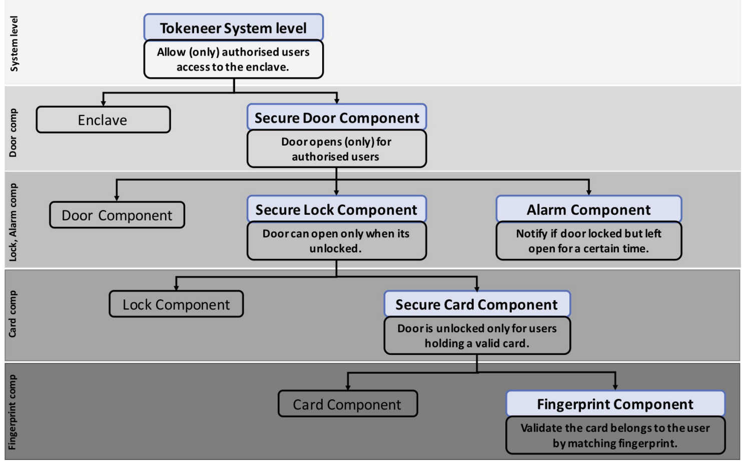

Fig. 1. Tokeneer: hierarchical component design, flow down requirements

The purpose of the Tokeneer system is to allow only authorised users to enter an enclave. Users may also leave the enclave. High level analysis of this system leads us to the design decision that, to achieve the system purpose, we need some kind of secure door whose purpose is to only open for authorised users. (Note that the prefix *secure* implies that this door has some extra functionality beyond a normal door that we have yet to design). Analysis of the secure door in turn leads to the decision to use an ordinary (i.e. unintelligent) door and a secure lock to achieve the functionality of the secure door. However, the analysis of the secure door also revealed a risk that the door may be left open by a user, leading to a decision to introduce an alarm component at the same level. The secure lock and alarm components are at the same conceptual level but functionally independent and can be analysed individually in consecutive analysis levels.

The alarm component analysis does not lead to any further sub-components and the derived requirements of this component are therefore used as input to its implementation (or validation in case of a given component). The secure lock is further decomposed into an ordinary lock and a secure card component which in turn is decomposed into an ordinary card and a fingerprint component. In summary, there are five control components in the Tokeneer design structure (over three levels): secure door, secure lock, alarm, secure card and fingerprint. There are four passive environment objects that are controlled by the Tokeneer control system: door, lock, card reader, fingerprint reader.

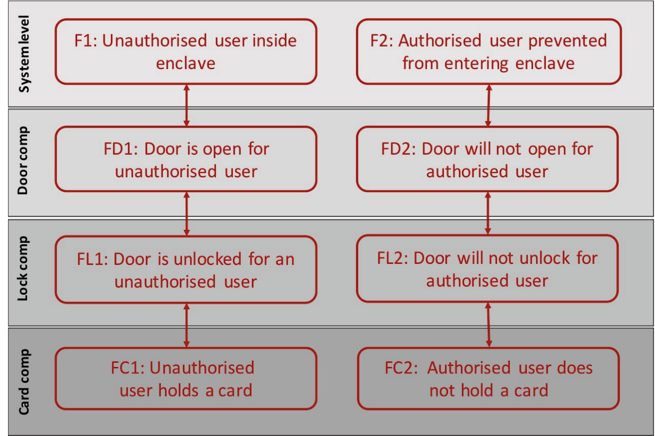

Fig. 2. Tokeneer: hierarchical failures

Failures at the immediate sub-component level could cause a failure at the previous level. Hence, in line with the hierarchical component design (Fig. 1), starting from the top level system failures, we have derived a hierarchy of failures as illustrated in Fig. 2. The left side of Fig. 2, presents the relations between failures arising from a breach of the system-level security constraint. For example, if an unauthorised user holds a card (FC1) this can result in the door unlocking for the unauthorised user (FL1) followed by the door opening (FD1) where upon the unauthorised user can enter the enclave (F1). Security attacks may also target denial of functionality which is sometimes omitted in safety analysis (i.e. a system that does nothing is often considered safe). Relations between security failures related to a loss of functionality are illustrated on the right hand side on Fig. 2. For example, if an authorised user loses their card (FC2) it prevents the enclave door from unlocking (FL2) and opening (FD2) and hence an authorised user is prevented from entering the enclave (F2).

In the next two sections we use the Tokeneer case study to illustrate the process steps for the first two phase of SHARCS: system level (Sect. 4) and component level (Sect. 5).

4 System Level

In this section we describe the system-level phase. The system-level phase itself consists of five steps.

Step 1, Action Analysis: The system level action analysis is presented for Tokeneer in the table in Fig. 3. The main purpose of the Tokeneer system is to allow authorised users to enter the enclave and prevent unauthorised users from entering. At this level, a failure is a violation of the system purpose so we identify failures by negating the purpose leading to the two failures presented in Fig. 3: F1 represents a breach of the required security property and F2 represents a 'denial' of functionality.

Following the STPA approach, we analyse the control actions with respect to system level failures that could result from the actions. At this level (Fig. 3), there are two identified actions to enter and leave the enclave. Action analysis considers whether lack of execution of the action, or execution under the wrong conditions, timing or ordering, could result in one or more of the identified failures.

System level

Purpose: Allow (only) authorised users access to the enclave.
Actions: Users can enter and leave enclave.

Failures:
- **F1**: Unauthorised user inside enclave
- **F2**: Authorised user prevented from entering enclave

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
User Enter Enclave	**A11:** Authorised user prevented from entering enclave (*F2*)	**A12:** Unauthorised user enters enclave (*F1*)	N/A
User Leave Enclave	No failure	No failure	N/A

Mitigations:
- ***Door*** component opens (only) for authorised users (addressing *A11, A12*)

Fig. 3. System level, action analysis table

Step 2, Formal Modelling: We now construct a formal model to capture the behaviour of the identified control actions as well as the environment around the control system and any invariant properties capturing the purpose of the system. The two identified actions are specified as abstract events in the system-level Event-B model (Fig. 4). We choose to model the system state using a set inEnclave of the users that are in the enclave. Another set authorisedUsers specifies which users are authorised to enter the enclave. Formally, we can express the security constraint as an invariant property; the set of users in the enclave is a subset of the authorised users:

$$@inv1: inEnclave \subseteq authorisedUser$$

```
event userEnterEnclave
any user
where
  @grd1: user ∉ inEnclave
  @grd2: user ∈ authorisedUser
then
  @act1: inEnclave := inEnclave ∪ {user}
end

event userLeaveEnclave
any user
where
  @grd1: user ∈ inEnclave
then
  @act1: inEnclave := inEnclave \ {user}
end
```

Fig. 4. (part of) Event-B model for system level

The userEnterEnclave event has one parameter, user, and two guards. The first guard grd1 represents an assumption that the user is not already in the enclave, while grd2 ensures that the user is authorised to enter the enclave. If both guards are satisfied then the event is allowed to fire and the action act1 updates the variable inEnclave by adding the instance user. The action analysis in Fig. 3 helps us to identify the need for grd2 of userEnterEnclave: this guard addresses failure F1, since lack of this guard results in failure of a security constraint (an unauthorised user enters enclave).

Step 3, Formal Validation and Verification: In formal models, we distinguish between safety properties (something bad never happens) and liveness properties (something good is not prevented from happening). Occurrence of failure F1 would represent a violation of safety since it would result in violation of invariant inv1. Failure F2 is a denial of service failure and, in the formal model, this failure represents a violation of liveness. We use the scenario checker tool in the Rodin tool for manual validation of liveness. Figure 5 shows the scenario checker tool being used to check the F2 failure scenario; the scenario involves two authorised users entering the enclave and the scenario checker demonstrates that both users can enter the enclave sequentially. Animation of the abstract model is a useful way for a modeller (or domain expert) to use their judgement to validate that the model accurately captures the security requirements. Model checking and animation can identify potential violations of the security invariant and violations of liveness, i.e., denial of entry for authorised users.

Once the model is determined to be a valid representation of the system, we use automatic theorem provers to verify security constraints (such as F1 expressed as the invariant inv1). The embedded theorem prover of the Rodin tool discharges the invariant preservation proof obligation for the userEnterEnclave

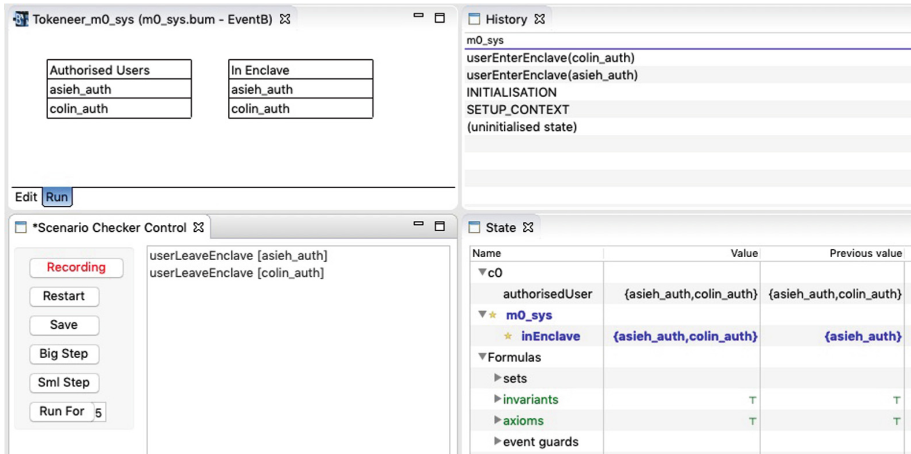

Fig. 5. Scenario checker tool applied at system level

event, verifying that it preserves the specified invariant. Note that grd2 is necessary to prove that the userEnterEnclave event preserves invariant inv1.

Step 4, Adjust the Analysis and Models: In the case that the scenario checking or verification identifies problems with the formal model, we make adjustments in order to remove the problems. These might be problems with the formalisation or might be due to problems in the informal analysis. The analysis and formalisation of Tokeneer at this abstract level is straightforward and does not reveal any problems. In the next section we demonstrate how the need to formally verify the correctness of the refined model incorporating the secure door component leads us to revisit and clarify our assumptions about the potential tailgating by unauthorised users.

Step 5, Mitigation and Outline Design for Next Phases: The system level requirements specify the desired behaviour but do not say how it will be achieved. That is, unauthorised users are prevented from entering but we do not specify how. Next we need to take a design step and introduce some subcomponents that take responsibility for this behaviour. Domain knowledge (and common practice) provides a suggestion for the next level design (mitigation): the introduction of a door component. The mitigation represents the identified next level component(s) and derived requirement(s) for that component(s), which address the control actions identified in Step 1, that could lead to failures. Each mitigation can address more than one failure. The door component here addresses both identified failures: the door opens so that authorised users can enter the enclave but does not open for users that are not authorised.

The interplay between the (informal) analysis, inspired by STPA, (Steps 1) and the formal modelling (Steps 2–3) is important. The analysis in Step 1 identifies key properties, actions and conditions under which actions may cause

failures. These guide the construction of the formal modelling in Steps 2, including invariants, events (corresponding to actions) and event guards (to prevent failures). The formal modelling in turn increases the degree of rigour in the analysis through the automated support for scenario checking, model checking and proof (Step 3). The formal modelling can identify gaps or ambiguities in the informal analysis resulting in the need to adjust the informal analysis and formal modelling to address these (Step 4).

The derived requirement for the door component is shown at the bottom of Fig. 3. In the next section, we will describe further analysis of the door component leading in turn to the identification of further components and analysis of those components.

5 Component Level

In this section we describe the component phase. The component phase is subsequently repeated if we identify further sub-components. For example, Fig. 1 illustrates how failure analysis of the secure door component leads to identification of secure lock and alarm components. The steps involved in the component level phase are similar to those of the system-level phase, which were explained in the previous section. Here we only highlight the differences:

- **Step 1:** Consider the *component* purpose, which has been identified as part of the previous level analysis and identify component failures (by negating the component purpose). For certification purposes, it is useful to record how the potential failures of this component *link*, via the control actions that this component addresses, to the previous level failures.
- **Step 2:** *Refine* the abstract formal model to capture:
  - *component* properties as invariants.
  - *refined/new events* representing *component* level actions.
- **Step 3:** Use automated theorem proving and model checking to verify constraints including the *refinement proof obligations.*

5.1 Component Level: Door

The secure door component, Fig. 6, addresses two of the insecure user actions, A11 and A12, from the previous level (see Fig. 3), which lead to the failures, FD2 and FD1, identified in the previous level.

Step 1: Analysis of the door component's actions is presented in Fig. 6. Two failures (FD1 and FD2 in Fig. 6) are found by negating the purpose of the door component which was identified in the previous level (see Fig. 3). The failures FD1, FD2 are linked to failures F1 and F2, respectively, from the previous level (for a broader illustration of the connection between failures, see Fig. 2).

Note that the actions of the previous level are still part of the system behaviour (and hence model) but are not analysed further at this level since their potential failures have been addressed by introducing the door sub-component

and delegating their responsibilities to the new actions of the door. The table in Fig. 6 identifies the scenarios under which the open door and close door actions may lead to failures.

Not all control action problems can be addressed by the design. Here mitigation is divided into two types: design mitigation, where there is a proposed design decision for the problem(s), and user mitigation, where the user can contribute to mitigating the problem. In the 'wrong timing or order' cases, Fig. 6, (AD23: the user closes the door before entering) and (AD43: the user leaves door when the door is open), these are user errors which cannot be prevented by the system. The provers detect such anomalies in temporal behaviour that violate the invariants and we fix the system by constraining the behaviour, either by making assumptions about the environment (including users) or by adding features to the control system. For these cases, Fig. 6 includes user mitigation to address AD23 (user opens the door again) and an assumption about user behaviour to address AD43 (user will not leave the door while the door is open). Thus there is no need to address these failures in the control system design.

Door Component

Purpose: Door opens (only) for authorised users.
Actions: Users can open and close doors.

Failures:
- **FD1**: Door is open for unauthorised user (causes *F1*)
- **FD2**: Door will not open for authorised user (causes *F2*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
User Open Door	**AD11:** Authorised user is unable to open the door (*FD2*).	**AD12:** Unauthorised user opens the door (*FD1*)	N/A
User Close Door	**AD21:** User does not close the door (*FD1*)	No failure	**AD23:** Authorised user closes door before entering (*FD2*)
User Approach Door	No failure	No failure	No failure
User Leave Door	No failure	No failure	**AD43**: Authorised user leaves door, when door is open, and so the door is left open for an unauthorised user

Mitigations:
- ***Lock*** component controls when the door can be opened (addressing *AD11, AD12*)
- ***Alarm*** component warns when door is left open for a certain time (addressing *AD21*)
- If a user closes the door before entering, they can open it again (addressing *AD23*)
- Authorised users will not leave the door area while the door is open (addressing AD43)

Fig. 6. Door component, action analysis table

Step 2-3-4: Figure 8 presents the first refinement of the Tokeneer Event-B model to introduce the door component. There are two versions of this refinement, the initial model (Fig. 8a), where the security constraints are more rigidly enforced, and the adjusted model (Fig. 8b), where security relies partly on user behaviour. These two models are not refining each other. The adjusted model is a replacement of the initial model.

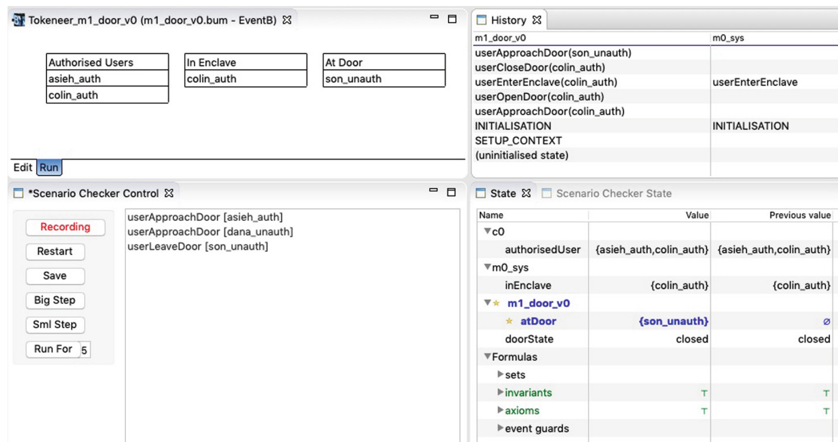

Fig. 7. Scenario checker tool at the door level

In the initial model (Fig. 8a), the userEnterEnclave abstract event (see previous section) is refined and the check that the user is authorised, specified in grd2, is replaced by checking the state of the door (a user can enter enclave only when the door is open). This guard replacement shifts the role of checking authorisation to the door. A proof obligation is generated by the Rodin tool since guards must not be weakened by refinement (i.e. the refined guard implies the abstract guard). To prove that the guard is not weakened we need an invariant property: when the door is open, then all users by the door must be authorised since any of them could enter the enclave. This is an example of how proof obligations associated with a formal model lead to the discovery of necessary assumptions. To model this assumption we introduced a variable atDoor to represent the subset of users by the door and the necessary invariant property (inv2a in the listing). To preserve this invariant, the userApproachDoor event also checks that the door is closed before allowing a new user to be added to the atDoor variable, act1. Specifying that a user will only approach the door when it is closed is a rather strong assumption and we re-visit this in our second model of the secure door.

The purpose of the door component is specified formally in the model by a combination of an invariant inv2a and a guard, grd3, of the event userOpenDoor. The invariant captures our assumption about users in the case that the door is open and the guard checks that all users by the door are authorised before allowing the user to open the door. The FD1 failure, *door opens for unauthorised user*, is prevented by grd3 of the userOpenDoor event which represents the requirement that the door has some, yet to be designed, security feature.

The guard grd2 of userLeaveDoor event is needed to prevent FD2, *Door does not close*. Without this condition an authorised user can open the door and then leave with the door open so that no other user can approach the door (because of our strong assumption that users approach the door when it is closed) which results in a deadlock. We demonstrated this (before adding grd2 of userLeaveDoor

event) by using the scenario checker to execute a scenario where an authorised user leaves the door without closing it. This scenario leads us to observe that the door must not be left open, meaning that we need to constrain (i.e. make assumptions about) user behaviour in our Event-B model in order to show that the system is secure.

Another scenario (shown in Fig. 7) demonstrates that when an authorised user is in the enclave, the presence of an unauthorised user by the door prevents the authorised user from opening the door to leave the enclave (trapped in the enclave).

The model in Fig. 8a includes the assumption that when the door is open, then all users by the door must be authorised. By making this assumption we are departing from the original specification of the Tokeneer system which has no such prevention/checking mechanism and relies instead on authorised users preventing tailgating. The experience gained from the scenario checking led us to change our assumption and relax the condition inv2a specified in the initial version of the model. Instead we make the assumption that the presence of authorised users will deter unauthorised ones from entering the enclave. In the adjusted model, inv2a is replaced by inv2b (Fig. 8b): when the door is open there is either a user in the enclave or at least one authorised user is by the door.

This illustrates Step 4, where the formal modelling informs the informal analysis. The assumption about tailgaters is modified: in the initial model, we assume there is no potential tailgater by an open door; while in the adjusted model we assume the authorised users will prevent tailgating. The adjusted version is more realistic but relies on stronger assumptions about user behaviour.

In order to be able to use scenarios to test whether the model prevents unauthorised users from entering we deliberately model the event that we hope to prevent. The abstract userEnterEnclave is split into two refining events: authUserEnterEnclave and unauthUserEnterEnclave. The guard of the latter event (which includes a conjunct that no authorised users are at the door) must never hold, thus preventing an unauthorised user from entering the enclave. A contradiction between inv2b and the guard of unauthUserEnterEnclave ensures that it is never enabled. This is an example of a negative scenario which we do not want to be possible in the system. These negative scenarios involve a check that some particular events are disabled at a particular state of the system. Note that disabledness is preserved by refinement since guards must not be weakened in refinement.

In this modified version of the model, grd3 of the userApproachDoor event is removed, so that a user can approach the door even when the door is open. Also grd3 of userOpenDoor is changed, so that the authorisation is only checked for the particular user that attempts to open the door (i.e. unauthorised users may also be in the vicinity of the door). These changes introduce more assumptions on human behaviour: an authorised user will prevent unauthorised users from entering the enclave.

In Event-B, ordering is specified implicitly by guards on the state conditions required for events to occur. For our model this is quite natural, e.g., the door

```
invariants
@inv2a: doorState = open ⇒
    atDoor ⊆ authorisedUser

events
event userEnterEnclave
refines userEnterEnclave
any user
where
@grd1: user ∈ atDoor
@grd2: doorState = open
then
@act1: inEnclave := inEnclave ∪ {
    user}
@act2: atDoor := atDoor \ {user}
end

event userApproachDoor
any user
where
@grd1: user ∉ atDoor
@grd2: user ∉ inEnclave
@grd3: doorState = closed
then
@act1: atDoor := atDoor ∪ {user}
end

event userLeaveDoor
any user
where
@grd1: user ∈ atDoor
@grd2: doorState = closed
then
@act1: atDoor := atDoor \ {user}
end

event userOpenDoor
any user
when
@grd1: doorState = closed
@grd2: user ∈ atDoor ∨ user ∈
    inEnclave
@grd3: atDoor ⊆ authorisedUser
then
@act1: doorState := open
end
```

(a) Inital model

```
invariants
@inv2b: doorState = open ⇒
    inEnclave ≠ ∅ ∨ (atDoor ∩
    authorisedUser) ≠ ∅

events
event authUserEnterEnclave
refines userEnterEnclave
any user
where
@grd1: user ∈ atDoor
@grd2: doorState = open
@grd3: user ∈ authorisedUser
then
@act1: inEnclave := inEnclave ∪ {
    user}
@act2: atDoor := atDoor \ {user}
end

event unauthUserEnterEnclave
refines userEnterEnclave
any user
where
@grd1: user ∈ atDoor
@grd2: doorState = open
@grd3: user ∉ authorisedUser
@grd4: atDoor ∩ authorisedUser =
    ∅
@grd5: inEnclave = ∅
then
@act1: inEnclave := inEnclave ∪ {
    user}
@act2: atDoor := atDoor \ {user}
end

event userApproachDoor
any user
where
@grd1: user ∉ atDoor
@grd2: user ∉ inEnclave
begin
@act1: atDoor := atDoor ∪ {user}
end

event userOpenDoor
any user
when
@grd1: doorState = closed
@grd2: user ∈ atDoor ∨ user ∈
    inEnclave
@grd3: user ∈ authorisedUser
then
@act1: doorState := open
end
```

(b) Adjusted model

Fig. 8. Event-B model for the door component

needs to be open for the user to enter, and thus the event for opening the door will have to have occurred before the user can enter. In addition, the scenario checking allows us to describe ordering explicitly and validate that the model allows that ordering.

Step 5: We now take further design steps to elaborate how this secure door works. We finish the door phase by suggesting a mitigation, an outline design solution, that will address the potential failures discussed in this phase. We will fit the door with a secure lock component to make sure that it can only be opened for authorised users (addressing insecure actions AD11 and AD12) and an alarm component to detect and warn when it is left open (addressing AD21). These new components are then analysed in the following phases.

In the rest of this section the remaining component levels are briefly described omitting detailed step descriptions, due to space limitation. However the full analysis is available here: https://tinyurl.com/SHARCS-dataset.

5.2 Component Level: Lock, Alarm, Card and Fingerprint

In this level, we introduce two components that need to be analysed: Secure Lock and Alarm.

The lock component, addresses two of the insecure control actions, AD11 and AD12, from the previous level (see Fig. 6), which resulted in failures, FD2 and FD1 (resp.) of the previous level. An alarm is activated if the door is left open longer than the time needed for a user to enter. The alarm component addresses the insecure action, AD21, from the previous level (see Fig. 6), which resulted in failure FD1 of the previous level. The card and finger print components addresses the insecure control actions from the previous levels (see Fig. 1 and Fig. 2).

6 Related Work

STPA has also been combined with other formal methods. In [1], Abdulkhaleq et al. propose a safety engineering approach that uses STPA to derive the safety requirements and formal verification to ensure the software satisfies the STPA safety requirements. The STPA-derived safety requirements can be formalised and expressed using temporal logic. Hata et al. [7] formally model the critical constraints derived from STPA as pre and post conditions in VDM++. Thomas and Leveson [16] have also defined a formal syntax for hazardous control actions derived from STPA. This formalisation enables the automatic generation of model-based requirements as well as detecting inconsistencies in requirements. Unlike our approach, these approaches do not support an incremental, hierarchical analysis approach.

Based on the hybrid methodology of STPA and NIST SP800-30 [6] proposed by Pereira et al. [13], Howard et al. [9] develop a method to demonstrate and formally analyse security and safety properties. The goal is to augment STPA with formal modelling and verification via the use of the Event-B formal method and its Rodin toolset. Identification of security requirements is guided by STPA,

while the formal models are constructed in order to verify that those security requirements mitigate against the vulnerable system states. Dghaym et al. [5] also apply a similar approach to [9] for generating safety and security requirements. Event-B has previously been combined with STPA by Colley and Butler [4] for safety analysis, again using STPA to guide the identification of safety requirements and Event-B to verify mitigation against hazardous states. Also, in our previous work [12] we utilised STPA and STPA-Sec for analysing the safety and security of autonomous systems. [4,5,9,12] only support requirements analysis at a single abstraction level rather than the hierarchical approach that we support.

7 Conclusion and Future Works

We have presented an analysis method that starts from the top level system requirements and identifies potential failures that could lead to unsafe accidents or security losses. The informal STPA analysis is used in conjunction with formal modelling to systematically and rigorously uncover vulnerabilities in a proposed design that could allow external fault scenarios to result in a failure. The formal modelling gives precision and a better understanding of the behaviours that are involved and lead to these failures. The model verification and validation provide strong evidence to back up the analysis. The identified vulnerabilities then drive the process as we design sub-components that can address the threats. In this way we flow down the requirements to derived requirements. Our experience with the Tokeneer case study highlighted that assumptions about user behaviour are critical and can be incorporated into the analysis. The formal verification and validation processes are beneficial in making these assumptions and consequent reliance explicit and clear. We suggest that our analysis method provides rigorous evidence (i.e., precise with clear hierarchical links and formal arguments) of the the security or safety requirements and how they are achieved in the design.

We have evaluated the method using a security case study; However we believe it works equally beneficial for safety requirements too. As a future work, we are planning to apply the SHARCS to a safety case study. As a further direction to improve our method, we are working to introduce a new kind of diagram, *control abstraction diagrams*, that help visualise the entities involved at a particular abstraction level along with their information and control relationships and the constraints that they make on each others actions. A control system can be thought of as a system that makes constrained actions. Our new control abstraction diagrams make clear, what the necessary constraints on actions are and which entities in the system are responsible for making them. As we incrementally introduce the design of a system we replace abstract constraints by adding new components that take on that responsibility and implement the constraint in an equivalent way. This matches very closely with our approach to system refinement in Event-B.

Acknowledgements. This work is supported by the following projects:

– HiClass project (113213), which is part of the ATI Programme, a joint Government and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture.

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

References

1. Abdulkhaleq, A., Wagner, S., Leveson, N.: A comprehensive safety engineering approach for software-intensive systems based on STPA. Procedia Eng. **128**, 2–11 (2015). http://www.sciencedirect.com/science/article/pii/S1877705815038588. Proceedings of the 3rd European STAMP Workshop 5–6 October 2015, Amsterdam
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in event-B. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
4. Colley, J., Butler, M.: A formal, systematic approach to STPA using event-B refinement and proof (2013). https://eprints.soton.ac.uk/352155/. 21th Safety Critical System Symposium
5. Dghaym, D., Hoang, T.S., Turnock, S.R., Butler, M., Downes, J., Pritchard, B.: An STPA-based formal composition framework for trustworthy autonomous maritime systems. Saf. Sci. **136**, 105139 (2021). https://www.sciencedirect.com/science/article/pii/S0925753520305348
6. Group, J.T.F.T.I.I.W.: SP 800–30 revision 1: Guide for conducting risk assessments. Technical report, National Institute of Standards & Technology (2012)
7. Hata, A., Araki, K., Kusakabe, S., Omori, Y., Lin, H.: Using hazard analysis STAMP/STPA in developing model-oriented formal specification toward reliable cloud service. In: 2015 International Conference on Platform Technology and Service, pp. 23–24 (2015)
8. Howard, G., Butler, M.J., Colley, J., Sassone, V.: Formal analysis of safety and security requirements of critical systems supported by an extended STPA methodology. In: 2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, Paris, France, 26–28 April 2017, pp. 174–180. IEEE (2017). https://doi.org/10.1109/EuroSPW.2017.68
9. Howard, G., Butler, M.J., Colley, J., Sassone, V.: A methodology for assuring the safety and security of critical infrastructure based on STPA and Event-B. Int. J. Crit. Comput. Based Syst. **9**(1/2), 56–75 (2019). https://doi.org/10.1504/IJCCBS.2019.098815
10. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Softw. Tools Technol. Transf. (STTT) **10**(2), 185–203 (2008)
11. Leveson, N.G., Thomas, J.P.: STPA Handbook. Cambridge, MA, USA (2018)
12. Omitola, T., Rezazadeh, A., Butler, M.: Making (implicit) security requirements explicit for cyber-physical systems: a maritime use case security analysis. In: Anderst-Kotsis, G., et al. (eds.) DEXA 2019. CCIS, vol. 1062, pp. 75–84. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27684-3_11
13. Pereira, D., Hirata, C., Pagliares, R., Nadjm-Tehrani, S.: Towards combined safety and security constraints analysis. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10489, pp. 70–80. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66284-8_7

14. Praxis: Tokeneer. https://www.adacore.com/tokeneer. Accessed May 2020
15. Snook, C., Hoang, T.S., Dghaym, D., Fathabadi, A.S., Butler, M.: Domain-specific scenarios for refinement-based methods. J. Syst. Archit. (2020). https://www.sciencedirect.com/science/article/pii/S1383762120301259
16. Thomas, J., Leveson, N.: Generating formal model-based safety requirements for complex, software-and human-intensive systems. In: Proceedings of the Twenty-first Safety-Critical Systems Symposium, Bristol, UK. Safety-Critical Systems Club (2013)
17. Young, W., Leveson, N.: Inside risks an integrated approach to safety and security based on systems theory: applying a more powerful new safety methodology to security risks. Commun. ACM **57**(2), 31–35 (2014). https://www.scopus.com/inward/record.uri?eid=2-s2.0-84893411630&doi=10.1145%2f2556938&partnerID=40&md5=07efb2984b5cf13de1fe2cb1583b7d27