Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-024-00551-8

ORIGINAL PAPER

Systematic hierarchical analysis of requirements for critical systems

Asieh Salehi Fathabadi[1] · Colin Snook[1] · Dana Dghaym[1] · Thai Son Hoang[1] · Fahad Alotaibi[1] · Michael Butler[1]

Received: 25 September 2023 / Accepted: 2 February 2024

Abstract

Safety and security are key considerations in the design of critical systems. Requirements analysis methods rely on the expertise and experience of human intervention to make critical judgements. While human judgement is essential to an analysis method, it is also 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. In this paper, we present a hierarchical and incremental 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. The safety and security analysis at each level uses STPA-style action analysis to identify hazards and vulnerabilities. At each level, we verify that the design achieves the safety or 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 hazard and vulnerability analysis, together with refinement-based formal modelling and verification, to drive the design, replacing the system level requirements with component requirements. In doing so, we decompose critical system-level requirements down to component-level requirements, transforming them from abstract system level requirements, towards concrete solutions that we can implement correctly so that the hazards/vulnerabilities are mitigated.

Keywords Safety · Security · Hazards · Vulnerabilities · Requirements · STPA · Event-B

1 Introduction and motivation

In an increasingly connected world of intelligent devices, safety and security have become key issues in the development of embedded components and cyber-physical systems. In the avionics domain, for example, standards such as ED202A [1] and ED203A [2] mandate a series of security risk assessments to be carried out at different stages of design. Security and safety are, in many cases, interrelated. Systems Theoretic Process Analysis (STPA) [3] is a method for analysing safety hazards in systems while STPA-Sec [4] adapts STPA for analysis of security vulnerabilities.

STPA is methodical in the sense that it provides a systematic approach to identification of hazards through analysis of the ways in which control actions might cause failures. STPA lacks the rigour of formality in the sense that it relies on human judgement to assess the effect of control actions. Formal techniques such as Event-B [5], on the other hand, are not methodical per se in that they rely on human judgement to make modelling choices, but can then provide a rigorous analysis of the properties of the model through formal verification.

In previous work [6–9], we have explored the combination of STPA and STPA-Sec with formal modelling to exploit the synergy between methodical 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 incre-

✉ Asieh Salehi Fathabadi
a.salehi-fathabadi@soton.ac.uk

Colin Snook
cfs@soton.ac.uk

Dana Dghaym
d.dghaym@soton.ac.uk

Thai Son Hoang
t.s.hoang@soton.ac.uk

Fahad Alotaibi
f.a.alotaibi@soton.ac.uk

Michael Butler
m.j.butler@soton.ac.uk

1 Electronics and Computer Science, University of Southampton, Southampton SO171BJ, UK

Published online: 12 March 2024

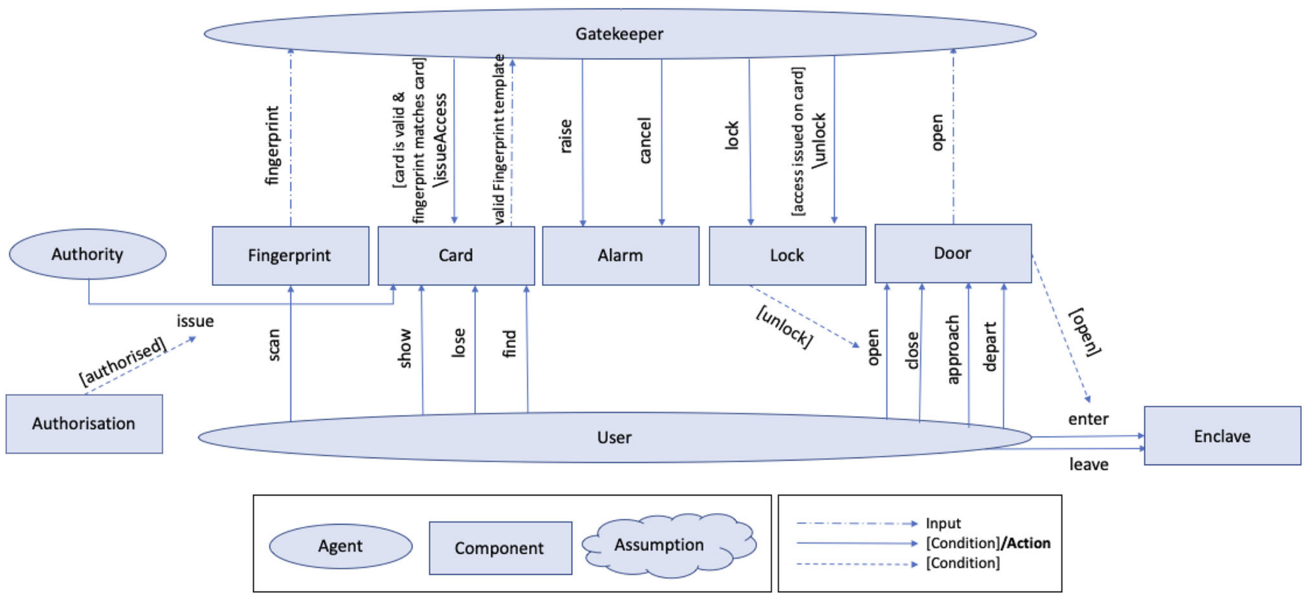


Fig. 1 Fingerprint component, control abstraction diagram

mental 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 hazard analysis part of our previous STPA/Event-B approach lacks systematic support for *incremental* analysis of a complex range of system hazards, in particular, it lacks support for abstraction and refinement in the analysis of control actions.

The current state of art lacks a methodical, rigorous and scalable single approach to analysis of safety and security. In this paper, we address this gap by adopting an abstraction-based incremental approach to hazard analysis. We call the approach Systematic Hierarchical Analysis of Requirements for Critical Systems (SHARCS). SHARCS addresses the rigorous formality challenges associated with application of STPA, the methodical challenges associated with application of Event-B, and scalability challenges of the existing STPA/Event-B combination [6–8].

To give the reader a flavour of what we mean by abstraction and refinement in action analysis, consider Fig. 1. This shows a Systematic Hierarchical Analysis of Requirements for Critical Systems (SHARCS) *control abstraction diagram* for a system whose purpose is to ensure that only authorised user are able to access a secure enclave. Details of the secure enclave case study and of the diagrams will be presented shortly. For now, the reader should observe of Fig. 1 that the solid arrows represent actions (scan, show, etc.) performed by agents (Authority, User, Gatekeeper) on components (Fingerprint, Card, etc). The model in Fig. 1 involves 3 agents, 7 components, and 16 actions. Now consider Fig. 5 which is abstraction of Fig. 1 containing 2 agents, 2 components, and 3 actions. Rather than performing hazard analysis across all the actions of the more complex Fig. 1, we commence the analysis on the smaller number of actions of Fig. 5. We then incrementally consider additional components and associated actions (e.g. the Secure Door component in Fig. 9) and analyse those additional actions for potential hazards. The choice of the abstraction of Fig. 5 is guided by the desire to commence the analysis with a model that succinctly captures the main purpose of the secure enclave system, i.e. ensure that only authorised users may enter the secure enclave. The minimal collection of agents, components and actions of Fig. 5 is sufficient to capture this purpose.

To our knowledge, an abstraction-based incremental approach to STPA control action analysis has not previously been considered. Our first attempt towards developing the SHARCS approach was presented in [10], which investigates the proposed idea on a case study but does not include the key concept of control abstraction diagram introduced here. In this paper, we introduce control abstraction diagrams, extend the security-critical enclave case study, and introduce a safety-critical railway level-crossing case study. We also systematically present the general development flow of the SHARCS approach. The artefacts from the case studies are available to download from https://doi.org/10.5258/SOTON/D2957.

The contributions described in this paper are as follows:

- For developers of critical systems, a method for rigorous and traceable analysis that flows down critical system requirements to derived component requirements and provides evidence that security and safety properties are addressed in the design.

- For safety/security analysts, a new (but somewhat 'STPA-like') abstraction-based, hierarchical component analysis method that uses formal models and proof to provide rigorous verification to the analysis at and between each refinement level.
- For users of refinement-based formal methods, a methodical approach to the difficult problem of finding an effective refinement strategy by selecting abstractions and requirements that should be modelled in each refinement.
- For stakeholders such as domain experts, progressive validation via scenarios executed on abstract models at different levels of requirements.
- A new diagram type, *control abstraction diagrams*, that illustrates control actions and constraints on them at different abstract levels.

The paper is structured as follows: Sect. 2 overviews the SHARCS approach, outlining its application to our secure enclave case study which is based on the Tokeneer access control system [11]. Section 3 presents the analysis and modelling steps for the highest abstraction level using the case study to demonstrate the steps in some detail. (Note: background information is provided in text boxes where needed to aid understanding of the paper.) Sect. 4 presents the analysis and modelling steps for individual refinement levels and how they relate to the corresponding higher abstraction level. Section 5 provides a specification of the overall workflow of the SHARCS process. We deliberately present the details of the analysis steps prior to specifying the overall workflow as this makes it easier for the reader to appreciate why the workflow is as specified. Section 6 discusses related and previous work. Detailed contributions of the proposed approach is presented in Sect. 7. Finally Sect. 8 concludes and describes future work.

2 An introduction to SHARCS approach and its outcomes

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. We focus on analysis of discrete control actions within a cyber-physical system. The controlled system may contain physical components that exhibit continuous behaviour but we assume the system handles continuous variables by discretisation. Event-B utilises a formal notion of abstraction and refinement where properties (e.g. the absence of security failures) can be expressed at a high level of abstraction and incremental refinements can be made to derive more detailed properties and verify that these respect the previously specified properties.

There is an established and documented taxonomy [12] used in the literature when discussing dependability including safety and security (which may impact safety and adds confidentiality). At the system level, a failure could result in a security *loss* or an unsafe *accident*. Hence a security loss is a failure that results in a security property being violated and an accident is a failure that results in a safety property being violated. Since we provide a generic approach for both safety and security, we simply use the term *failure*. In a hierarchical approach there are advantages if the process and terminology are self-similar throughout the levels, especially since system or component boundaries may be relative to the viewpoint. A system comprises a set of interacting components, each of which may be considered to be a system in its own right. The term, failure, works at any system boundary where we can consider the failure of that system to provide a service that it is responsible for. Faults in the design of a system allow it to get into a state that could lead to a failure. A fault may be a hazard that could lead to an accident or a vulnerability that could lead to a loss. Hence, we use *fault* as a generic term that includes hazards and vulnerabilities of a system.

In our hierarchical approach, for each system (or component sub-system), an STPA-style analysis identifies potential faults or vulnerabilities that could lead to service failures. STPA does this by considering a *control structure diagram* that identifies the structure of controlling agents and controlled components and the control actions that can be performed. However, we are working at multiple abstraction levels and our initial abstract models may abstract away from designed control agents, e.g. Fig. 5 abstracts away from the *Gatekeeper* control agent of Fig. 1. Instead, we propose an alternative style of diagram, appropriate to the level of abstraction, which shows the actors involved and in what ways they are required to constrain each others behaviour even if the mechanisms of that control are omitted. We call these diagrams *control abstraction diagrams*.

Control abstraction diagrams help us to identify the control actions involved in the set of behaviours that make up the service at the current abstraction level, and examine what effect a fault in that service would have on the system and what constraining conditions are needed to avoid it. This helps us to assess a proposed design consisting of interacting abstract components, each providing derived service functions. This assessment aims to mitigate service failure by strengthening the derived requirements placed on the components. The control abstraction diagram is an outline of the system at that level of abstraction and provides a good first step to making the corresponding Event-B model refinement. Validation of the Event-B model ensures that it accurately reflects the desired system-level service function. To do this, the model must also embody an abstraction of the system's environment, in particular, any external behaviour that could affect the service function. Applying formal verification to the Event-B model helps to identify faults in the design, which could cause or allow failures to manifest at the system level. Identification of design faults leads us to modify the design or strengthen the specifications of the service functions provided by the components. This strengthening will typically involve constraining the conditions under which control actions may be taken and hence strengthening guards in the Event-B events. Alternatively, in some cases we may revise our assumptions about the environment including user behaviour. The verification is repeated until no such design faults remain at this level of abstraction. We then move to the next level, analysing the sub-components as systems in their own right, with rigorous assurance of their derived requirements. In the remainder of this paper *requirements* means the specified functionality of service functions provided by a system and *derived requirements* means the specified functionality of service functions provided by the individual components of the system.

2.1 Overview of Tokeneer outcomes: requirements and failures hierarchies

Background: Tokeneer Case Study: Our main case study in this paper is the Tokeneer system (security-critical case study). The Tokeneer system [11] consists of a secure enclave and a set of system components, some housed inside the enclave and some outside (Figure 2). 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 by 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 to enter 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.

The Tokeneer system consists of several interlinked components and includes several hazards that can impact functionality; this makes it a practical case study to demonstrate the SHARCS, therefore we use it as the main case study in this paper.

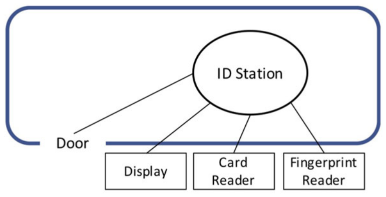


Fig. 2 Tokeneer Secure Enclave

The hierarchical component design process as applied to the Tokeneer system is illustrated in Fig. 3. Starting from the system level purpose, 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.

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.

Failures at the immediate sub-component level could cause a failure at the higher level. Hence, in line with the hierarchical component design (Fig. 3), starting from the top level system failures, we have derived a hierarchy of failures as illustrated in Fig. 4. The left side of Fig. 4 presents the relations between failures leading to a breach of the system-level security constraint. For example, if an unauthorised user

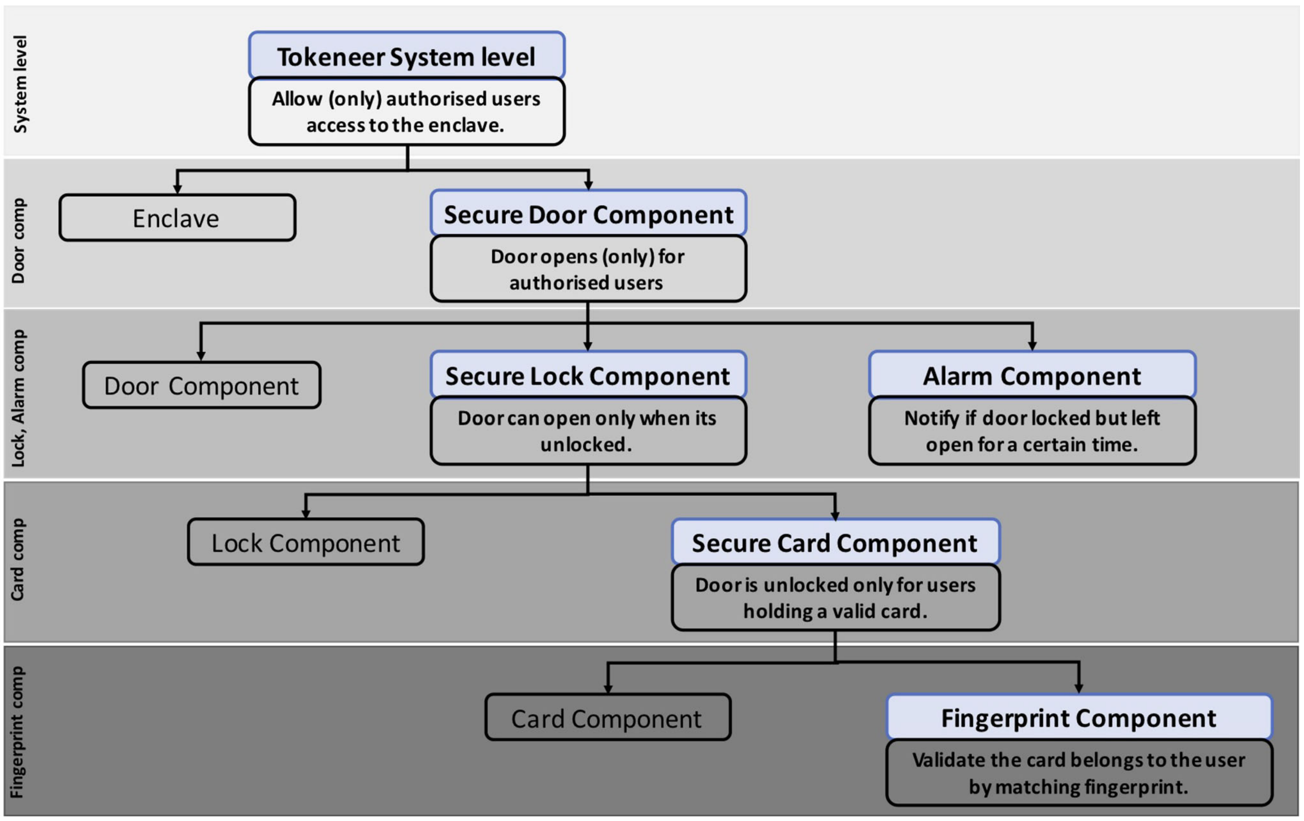
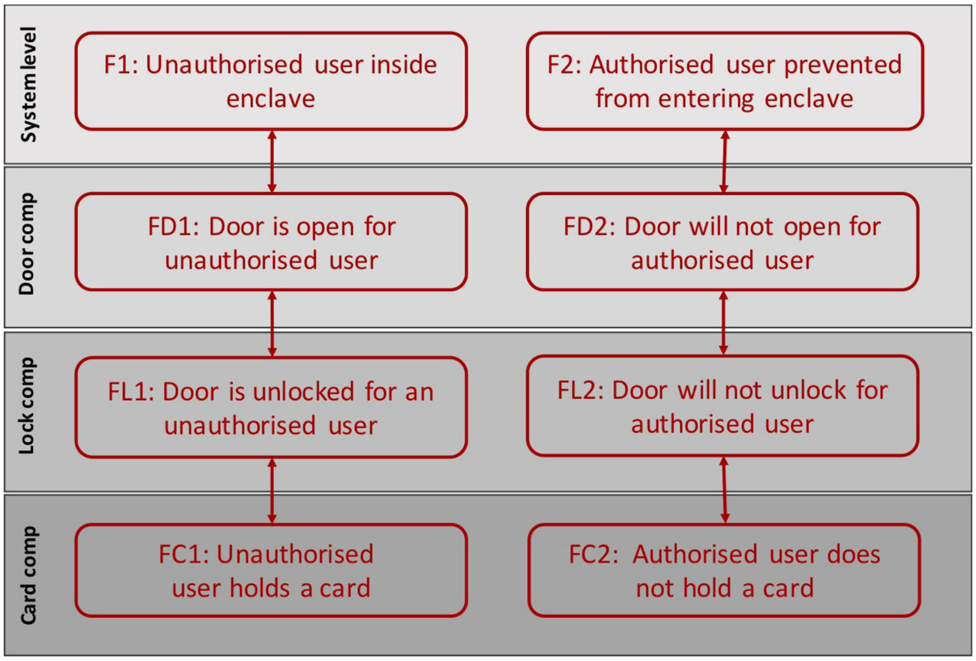

System level
Tokeneer System level
Allow (only) authorised users access to the enclave.
Door comp
Enclave
Secure Door Component
Door opens (only) for authorised users
Lock, Alarm comp
Door Component
Secure Lock Component
Door can open only when its unlocked.
Alarm Component
Notify if door locked but left open for a certain time.
Card comp
Lock Component
Secure Card Component
Door is unlocked only for users holding a valid card.
Fingerprint comp
Card Component
Fingerprint Component
Validate the card belongs to the user by matching fingerprint.

Fig. 3 Tokeneer: hierarchical component design, flow down requirements

Fig. 4 Tokeneer: hierarchical failures


System level
F1: Unauthorised user inside enclave
F2: Authorised user prevented from entering enclave
Door comp
FD1: Door is open for unauthorised user
FD2: Door will not open for authorised user
Lock comp
FL1: Door is unlocked for an unauthorised user
FL2: Door will not unlock for authorised user
Card comp
FC1: Unauthorised user holds a card
FC2: Authorised user does not hold a card

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 leading to a loss of functionality are illustrated on the right hand side on Fig. 4. For example, if an authorised user loses their card (FC2), the enclave door is prevented from unlocking (FL2) and opening (FD2) and hence an authorised user is prevented from entering the enclave (F2). Note that our design into sub-components does not always make the failures disappear. We drive them down, transforming them from an abstract system level failure, towards a concrete problem that we can address by introducing controls at an appropriate level so that they do not manifest.

We also applied our SHARCS approach to a railway level crossing case study. A level crossing is an intersection where a road crosses a railway line at the same level (i.e. without a bridge). We use this case study to show that our analysis approach can be used on a safety-critical system. The safety property is that cars and trains should not use the crossing at the same time. The hierarchical component design, hierarchy of failures, and the last level control abstraction diagram for the level crossing are included in the Appendix and the full development of the level crossing, including the STPA analysis tables and Event-B models, is available to download from https://doi.org/10.5258/SOTON/D2957. At the level of abstraction of the Event-B modelling, the security analysis and safety analysis are similar. In both case studies, failure is a result of the combination of undesirable user behaviour and vulnerabilities in the design. Security and safety are often treated as requiring different analysis methods. We applied our approach to both case studies to demonstrate that the same methods can be applied to both safety and security cases.

In the next sections, we use the Tokeneer case study to present various steps of the SHARCS approach in more detail. Note that Figs. 3 and 4 are produced as outputs from the final *consolidation* stage of the process shown later, Fig. 17. We have presented them here because they give a good overview of the steps used in the analysis.

3 System level analysis

In this section, first we describe the system-level analysis phase consisting of six steps through application to the Tokeneer case study.

An introduction to control abstraction diagrams:
The diagrams (see example diagrams in Figures 5 and 9) show *agents*, *components* and *assumptions* as nodes and *actions*, *conditions* and *input* as links.

- **Agents** are entities in the system that can act on a component. They could be part of the control system or part of the environment (e.g. User). Actions are shown as a link from the agent to the component and labeled with the name of the action. Agents may constrain each others actions by making a condition link to the constrained action (labelled with the constraint condition in square brackets). As shorthand, when they constrain their own actions the constraint condition is annotated on the action link. Input links show where agents utilise information contained in a component.
- **Components** are entities that are acted upon but do not take actions themselves. However, their state may constrain an agent's actions via condition links and may depend on information in other components via input links. Components may be physical objects in the system (e.g. enclave) or may be conceptual domain entities (e.g. authorisation). They may also represent abstract entities at this level of the analysis (e.g. secure door). In the latter case they will be replaced in future refinement levels by subsystems made up of agents and components, revealing how they are able to impose the constraints shown at this abstract level.
- **Assumptions** allow us to explicitly annotate properties or behaviours that the designed control relies on in order to operate correctly. Assumptions can constrain actions via condition links in a similar way to components.

Step 1: Control analysis The system level control abstraction diagram for Tokeneer is shown in Fig. 5 and is a reflection of the system requirement: "Allow only authorised users to enter the enclave". In the diagram an agent, *User*, performs the *enter* and *leave* actions on the physical component *enclave*. We also include a domain component to cater for the concept of *Authorisation* that imposes a constraint, *[authorised]*, on the enter action. An Authority agent can grant and revoke this user authorisation.

Background: Systems Theoretic Process Analysis (STPA):

STPA [3] 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. The STPA process includes the following steps:

- Identification of accidents and hazards.
- Providing a control structure diagram to identify the major components and control actions.
- Identification of unsafe control actions and consequently generating safety constraints.
- Causal factors identification to determine how unsafe actions could occur.

While STPA is used for safety problems, STPA-Sec [4] 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. It differs from STPA in the addition of intentional actions when identifying causal scenarios.

STPA and STPA-Sec are intended as methods for analysing a control system. It is assumed that a control structure diagram can be constructed. That is, the interfaces and safety constraints within the control system and between the controller and its environment are known and understood.

Step 2: Action analysis A system failure is a violation of the system purpose so we identify failures by essentially negating the purpose. For the Tokeener system negation of purpose leads to the two failures presented in Fig. 6: F1 represents a breach of the required security property and F2 represents a denial of functionality.

Once the control abstraction diagram has been constructed, we analyse the control actions with respect to system level failures that could result from the actions. This is presented for Tokeneer in Fig. 6 for both user actions of Fig. 5—*enter* and *leave*. 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. This analysis is shown in the action analysis table in Fig. 6. The entries in the action analysis table identify conditions under which non-occurrence, occurrence, or incorrect timing of the action would cause a failure. For example, the entry labelled **A12** describes the condition under which occurrence of the *enter* action would cause failure **F1**.

Background: Event-B:

Event-B [5] 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, $\mathsf{G(t,v)}$ is the guard of the event, and $\mathsf{v := E(t,v)}$ is the action of the event.

$$\mathsf{e} == \textbf{any}\ \mathsf{t}\ \textbf{where}\ \mathsf{G(t,v)}\ \textbf{then}\ \mathsf{v := E(t,v)}\ \textbf{end}$$

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. The detailed mathematical reasoning processes of Event-B modelling requires expertise in the language and concepts of proof, but this is achievable by most engineers. More challenging is the choice of useful abstractions that can be arranged in a sequence of steps to form a *refinement strategy* that introduces coherent issues in manageable steps and is amenable to the automatic theorem provers.

Event-B is supported by the Rodin [1]1 tool set [13], 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 following plug-in tools that extend the Rodin toolset:

3.0.1 ProB

[14] is an animator and model checker for the B-Method. It also supports Event-B and can be installed within the Rodin tool set. The ProB model checking facility complements Event-B theorem proving for verifying Event-B models by finding invariant violations and deadlocks. ProB also enables the validation of the model behaviour by exploring execution traces, which can be constructed by the manual selection of enabled events.

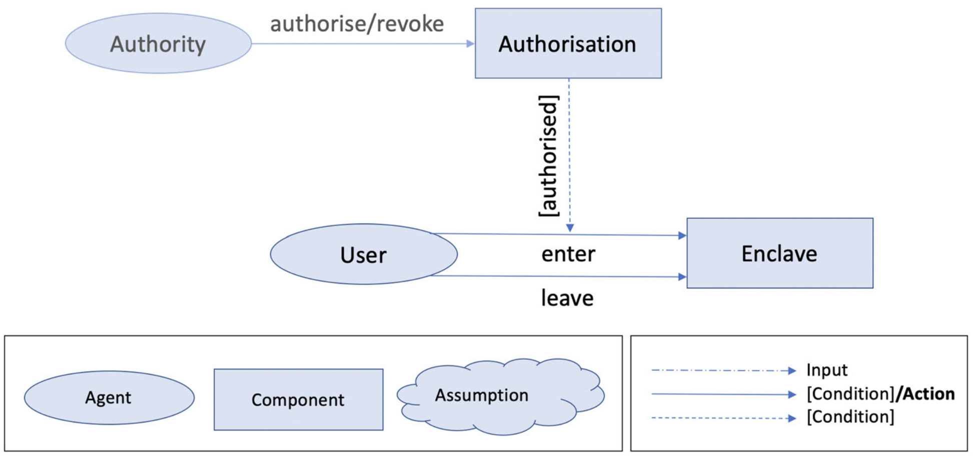
3.0.2 Scenario Checker

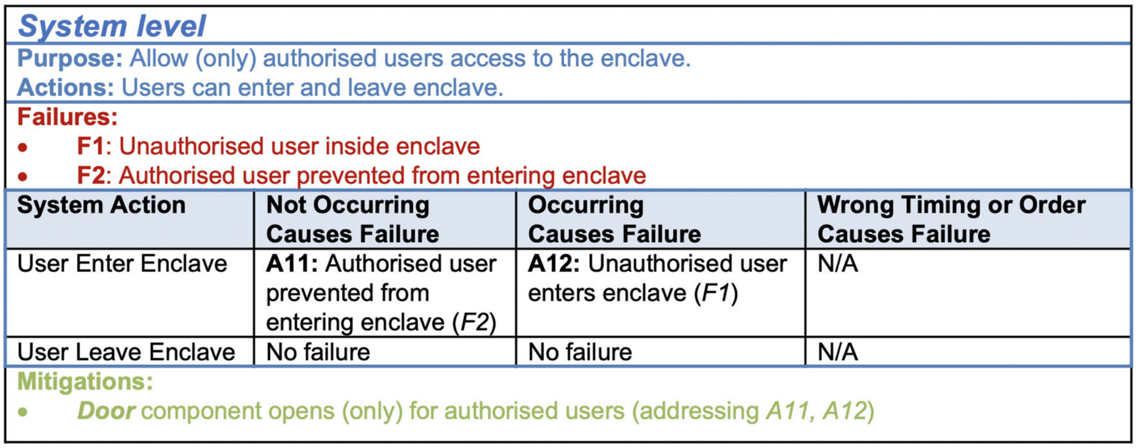
[15] is an animation tool that we developed for validating systems by recording and replaying scenarios. It extends ProB to support two new functionalities: a 'run to completion' style execution of controller events, and a record/replay style user interface for running test scenarios. In Event-B we model a closed system of interacting components including the environment and any controlling device without distinguishing between the different kinds of events. However, an environment event may trigger a sequence of controller events representing the controllers response. These internal controller steps are not usually explicitly specified by scenarios. Hence, to support scenario animation, we distinguish between external environment events and internal controller events. To simplify the scenario execution, the scenario checker records and replays the external events automatically firing any enabled internal events until none are enabled. Then the scenario checker will wait for the next external event to be selected by the user or by replaying the recorded scenario.

Step 3: Formal modelling We now construct a formal model to capture the behaviour of the identified control

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

Fig. 5 System level, control abstraction diagram





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. 6 System level, action analysis table

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. 7). 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 as follows:

$$@inv1 : inEnclave \subseteq authorisedUser$$

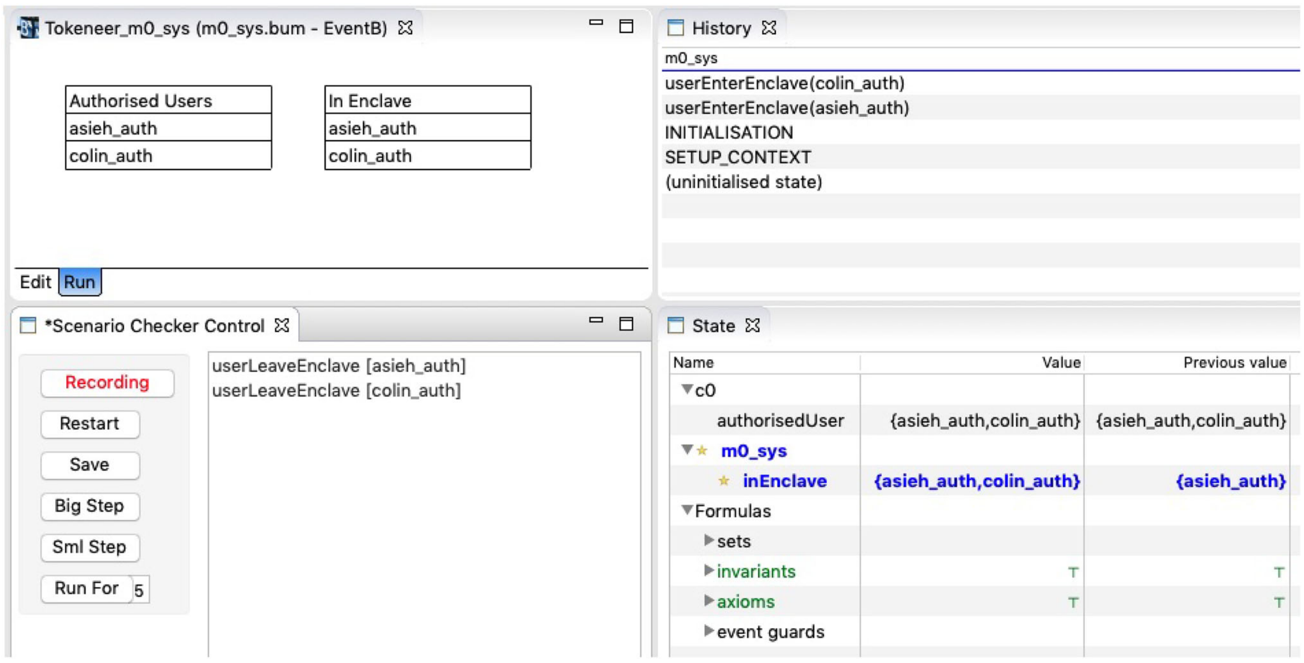
(This invariant property is from our Event-B model of the system.)

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 execute and the action act1 updates the variable inEnclave by adding the instance user. The action analysis in Fig. 6 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).

```
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. 7 (part of) Event-B model for system level



Authorised Users
asieh_auth
colin_auth

In Enclave
asieh_auth
colin_auth

```
m0_sys
userEnterEnclave(colin_auth)
userEnterEnclave(asieh_auth)
INITIALISATION
SETUP_CONTEXT
(uninitialised state)
```

```
userLeaveEnclave [asieh_auth]
userLeaveEnclave [colin_auth]
```

Name	Value	Previous value
▼c0		
authorisedUser	{asieh_auth,colin_auth}	{asieh_auth,colin_auth}
▼☆ m0_sys		
☆ inEnclave	{asieh_auth,colin_auth}	{asieh_auth}
▼Formulas		
▶sets		
▶invariants	⊤	⊤
▶axioms	⊤	⊤
▶event guards		

Fig. 8 Scenario checker tool applied at system level

Step 4: Formal verification and validation: 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.

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

We use the scenario checker tool in the Rodin tool for manual validation of liveness. Figure 8 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.

Step 5: 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 Sect. 4 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 potential tailgating by unauthorised users.

Step 6: Mitigation and outline design for next phases The system level requirements specify the desired behaviour but do not say how they will be achieved. That is, unauthorised users are prevented from entering but we do not specify how they are prevented. Next we need to take a design step and introduce some sub-components 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 action analysis of Fig. 6 helps us derive the requirements on the door component in that the door should mitigate the conditions (A11, A12) under which the actions could cause failures F1 or F2. This leads to the requirement that the door opens so that authorised users can enter the enclave but does not open for users that are not authorised. The derived requirement for the door component is also shown at the bottom of Fig. 6. In Sect. 4, we will describe further analysis of the door component leading in turn to the identification of further components and analysis of those components to derive their requirements.

The interplay between the (informal) analysis in Steps 1–2 and the formal modelling (Steps 3–4) is important. The analysis identifies key properties, actions and conditions under which actions may cause failures. These guide the construction of the formal modelling in Steps 3–4, 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. 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 5).

4 Component level analysis

In this section, we describe the steps of the component analysis phase of SHARCS through application to Tokeneer. The component analysis phase is subsequently repeated if we identify further sub-components. For example, Fig. 3 illustrates how failure analysis of the secure door component leads to identification of secure lock and alarm components. The steps involved in the component analysis phase are similar to those of the system-level analysis, which were explained in the previous section. Here we highlight the differences:

- *Step 1:* Elaborate the control abstraction diagrams to add the control action structure of the component(s) introduced in this phase.
- *Step 2:* 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 3: Refine* the abstract formal model to capture:
  - *Component* properties as invariants.
  - *Refined/new events* representing *component* level actions.
- *Step 4:* Use automated theorem proving and model checking to verify constraints including the *refinement proof obligations*.

Section 4.1 describes the steps involved in the analysis of a secure door component. Section 4.2 describes the steps involved in analysis of the secure lock and alarm components identified in Sect. 4.1. Section 4.3 describes the steps involved in analysis of the secure card component identified in Sect. 4.2. Section 4.4 describes the steps involved in analysis of the fingerprint component identified in Sect. 4.3.

4.1 Component level: door

The secure door component, Fig. 10, addresses two of the insecure conditions of the user actions, A11 and A12, from the previous level (see Fig. 6), which lead to the failures, FD2 and FD1, identified in the previous level.

Step 1: The refined control abstraction diagram for the door component is presented in Fig. 9. Here the secure door component is introduced with four actions for a user to open/-close the door and approach/depart the door. Compared to the system level control abstraction diagram (see Fig. 5), the role of checking authorisation is shifted to the secure door component which should open only for authorised users.

Step 2: Analysis of the door component's actions is presented in Fig. 10. Two failures (FD1 and FD2 in Fig. 10) are found by negating the purpose of the door component which was identified in the previous level (see Fig. 6). 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. 4).

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. 10 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. 10, (AD23: the user closes the door before entering) and (AD43: the user leaves door with the door still 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. 10 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.

Step 3–5: Fig. 12 presents the first refinement of the Tokeneer Event-B model to introduce the door component. There are two versions of this refinement, the initial refined model (Fig. 12a), where the security constraints are more rigidly enforced, and the adjusted model (Fig. 12b), where security relies partly on user behaviour. These two models are not refining each other. The adjusted model is a replacement of the initial refined model.

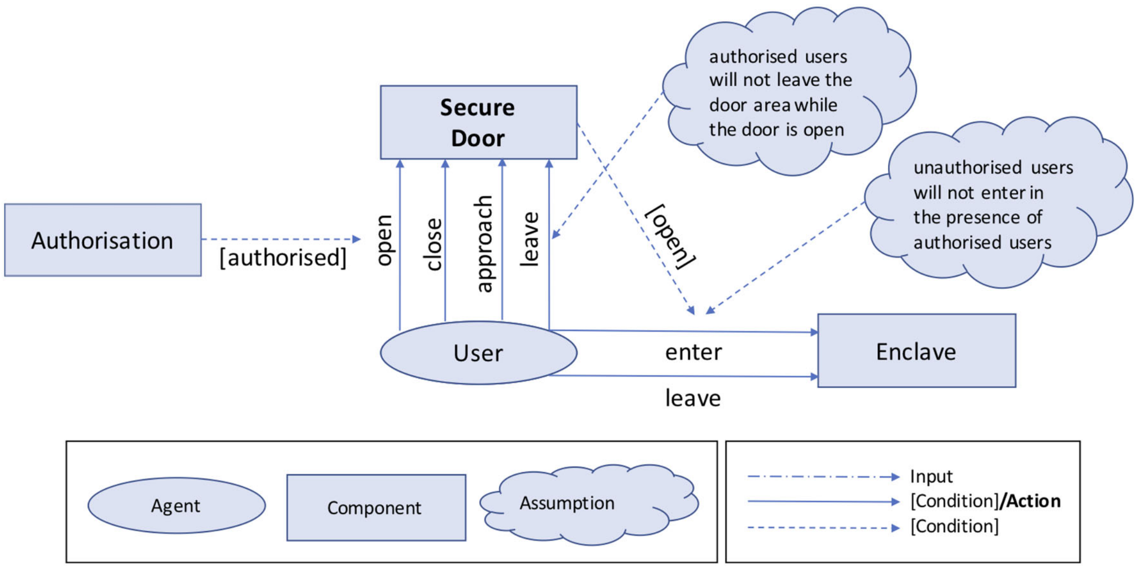
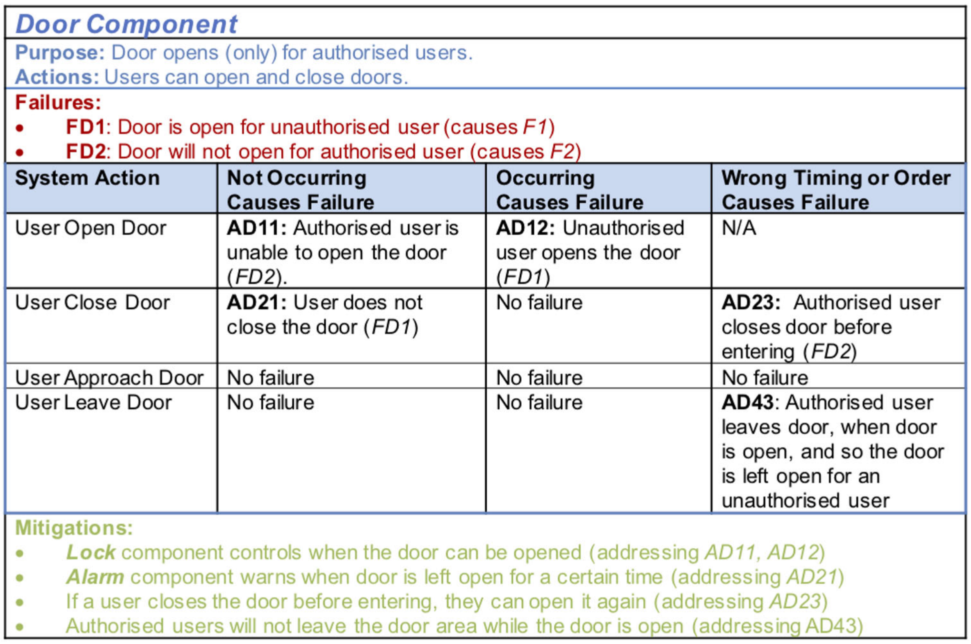


Fig. 9 Door component, control abstraction diagram

Fig. 10 Door component, action analysis table



Door Component

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

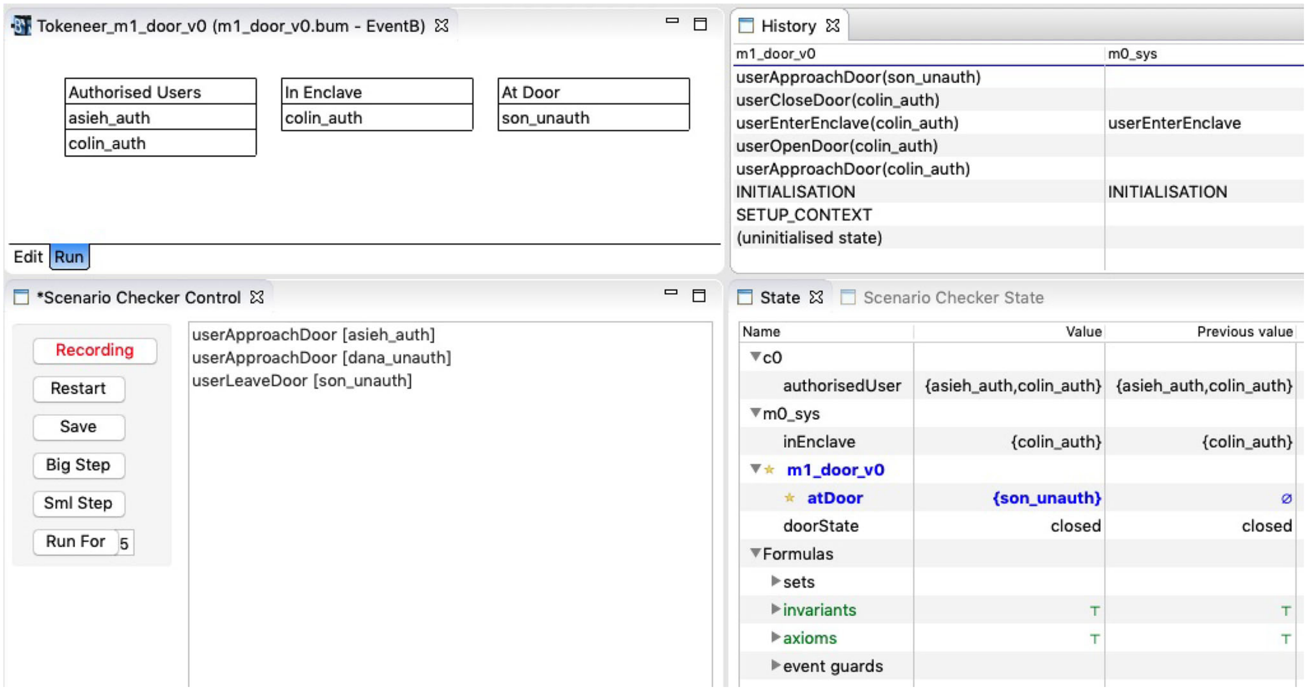
Failures:
- **FD1**: Door is open for unauthorised user (causes *F1*)
- **FD2**: Door will not open for authorised user (causes *F2*)

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)

In the initial refined model (Fig. 12a), 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,



Authorised Users	In Enclave	At Door
asieh_auth	colin_auth	son_unauth
colin_auth		

m1_door_v0	m0_sys
userApproachDoor(son_unauth)	
userCloseDoor(colin_auth)	
userEnterEnclave(colin_auth)	userEnterEnclave
userOpenDoor(colin_auth)	
userApproachDoor(colin_auth)	
INITIALISATION	INITIALISATION
SETUP_CONTEXT	
(uninitialised state)	

```
userApproachDoor [asieh_auth]
userApproachDoor [dana_unauth]
userLeaveDoor [son_unauth]
```

Name	Value	Previous value
▼c0		
authorisedUser	{asieh_auth,colin_auth}	{asieh_auth,colin_auth}
▼m0_sys		
inEnclave	{colin_auth}	{colin_auth}
▼☆ **m1_door_v0**		
☆ **atDoor**	**{son_unauth}**	∅
doorState	closed	closed
▼Formulas		
▶sets		
▶invariants	⊤	⊤
▶axioms	⊤	⊤
▶event guards		

Fig. 11 Scenario checker tool at the door level

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 event userLeaveDoor 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. 11) 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. 12a 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. 12b): 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 7, where the formal modelling informs the informal analysis. The assumption about tailgaters is modified: in the initial refined 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

Fig. 12 Event-B model for the door component

```
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
```

16

(b) Adjusted model

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 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 6: 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.

4.2 Component level: lock and alarm

In this level, we introduce two components that need to be analysed: Secure Lock and Alarm. Since they are relatively independent, the order in which we analyse them is arbitrary; we chose to do the secure lock first. The control abstraction diagram from the final analysis stage, including the lock and the alarm, is shown in Fig. 1.

4.2.1 Component level: lock

The lock component, Fig. 13, addresses two of the insecure control actions, AD11 and AD12, from the previous level (see Fig. 10), which resulted in failures, FD2 and FD1 (resp.) of the previous level.

As with the previous door level, the lock component is modelled in two versions: a more rigidly enforced model, where the door unlocks only when all users by the door are authorised (specified as an invariant $\mathsf{doorLatch} = \mathsf{unlocked} \Rightarrow \mathsf{atDoor} \subseteq \mathsf{authorisedUser}$), and an adjusted model, where, if the door is unlocked, then there is either a user in the enclave (who, according to the security invariant, must be authorised) or at least one authorised user is by the door. In this version the invariant is changed to $\mathsf{doorLatch} = \mathsf{unlocked} \Rightarrow \mathsf{inEnclave} \neq \varnothing \vee (\mathsf{atDoor} \cap \mathsf{authorisedUser}) \neq \varnothing$.

4.2.2 Component level: alarm

An alarm is activated if the door is left open longer than the time needed for a user to enter. The alarm component, Fig. 14, addresses the insecure action, AD21, from the previous level (see Fig. 10), which resulted in failure FD1 of the previous level.

The assumption is that an unauthorised user would not enter the enclave while the alarm sounds (human behaviour assumption). If the alarm does not occur when it should, there is no notification that the door has been left open. If it occasionally occurs when it should not, there is no failure. However, if the alarm sounds too often for no good reason, it may eventually be ignored when the door has been left open.

We assume that the alarm component is designed to mitigate the failure conditions identified in Fig. 14 and do not decompose this component further.

4.3 Component level: card

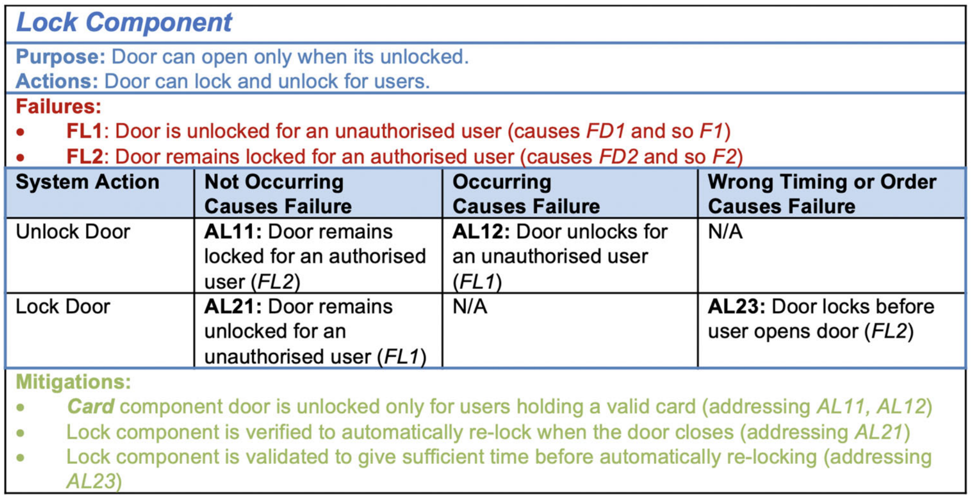
The card component, Fig. 15, addresses two of the insecure control actions, AL11 and AL12, from the previous level (see Fig. 13), which resulted in two of the failures, FL2 and FL1 (resp.), identified in the previous level. It does this by providing a mechanism for the lock to ascertain the authorisation of a user.

The actions considered in this phase, to issue a card and to lose a card, reveal further potential failures FC1 and FC2 where an unauthorised user holds a card or an authorised user has lost their card. An outline solution is proposed to solve this by introducing fingerprint detection to ensure that the current holder of the card is the one that it was issued to.

4.4 Component level: fingerprint

The fingerprint component, Fig. 16, addresses the insecure actions, AC12 and AC32 of the card component (see Fig. 15) which are both linked to the failure FC1.

However, it introduces a new failure if it incorrectly does not detect a match for a user with a valid card. This does not relate directly to a failure at the card component level because adding fingerprint matching introduces a new validation step. Sometimes this happens during the design. Also, the 'wrong order' action (AF13) is a problem of the lock component unlocking before the fingerprint has determined validity. In this case the mitigation requires verification of a component higher up the hierarchy to ensure it works correctly with the



Lock Component

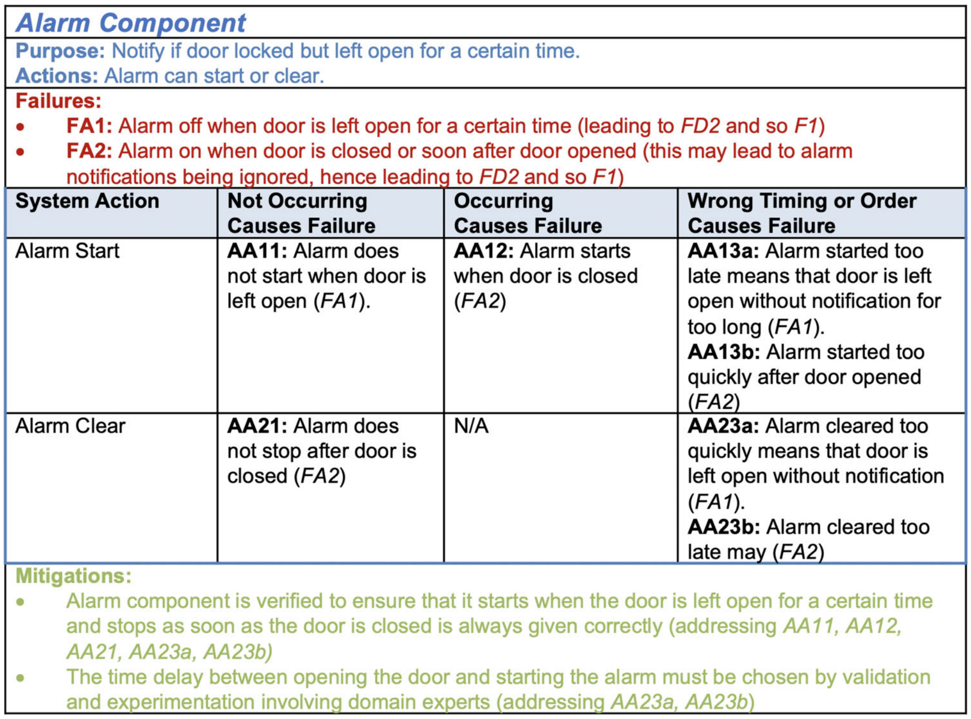
Purpose: Door can open only when its unlocked.
Actions: Door can lock and unlock for users.

Failures:
- **FL1**: Door is unlocked for an unauthorised user (causes *FD1* and so *F1*)
- **FL2**: Door remains locked for an authorised user (causes *FD2* and so *F2*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Unlock Door	**AL11:** Door remains locked for an authorised user (*FL2*)	**AL12:** Door unlocks for an unauthorised user (*FL1*)	N/A
Lock Door	**AL21:** Door remains unlocked for an unauthorised user (*FL1*)	N/A	**AL23:** Door locks before user opens door (*FL2*)

Mitigations:
- ***Card*** component door is unlocked only for users holding a valid card (addressing *AL11, AL12*)
- Lock component is verified to automatically re-lock when the door closes (addressing *AL21*)
- Lock component is validated to give sufficient time before automatically re-locking (addressing *AL23*)

Fig. 13 Lock component, action analysis table



Alarm Component

Purpose: Notify if door locked but left open for a certain time.
Actions: Alarm can start or clear.

Failures:
- **FA1:** Alarm off when door is left open for a certain time (leading to *FD2* and so *F1*)
- **FA2:** Alarm on when door is closed or soon after door opened (this may lead to alarm notifications being ignored, hence leading to *FD2* and so *F1*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Alarm Start	**AA11:** Alarm does not start when door is left open (*FA1*).	**AA12:** Alarm starts when door is closed (*FA2*)	**AA13a:** Alarm started too late means that door is left open without notification for too long (*FA1*). **AA13b:** Alarm started too quickly after door opened (*FA2*)
Alarm Clear	**AA21:** Alarm does not stop after door is closed (*FA2*)	N/A	**AA23a:** Alarm cleared too quickly means that door is left open without notification (*FA1*). **AA23b:** Alarm cleared too late may (*FA2*)

Mitigations:
- Alarm component is verified to ensure that it starts when the door is left open for a certain time and stops as soon as the door is closed is always given correctly (addressing *AA11, AA12, AA21, AA23a, AA23b)*
- The time delay between opening the door and starting the alarm must be chosen by validation and experimentation involving domain experts (addressing *AA23a, AA23b*)

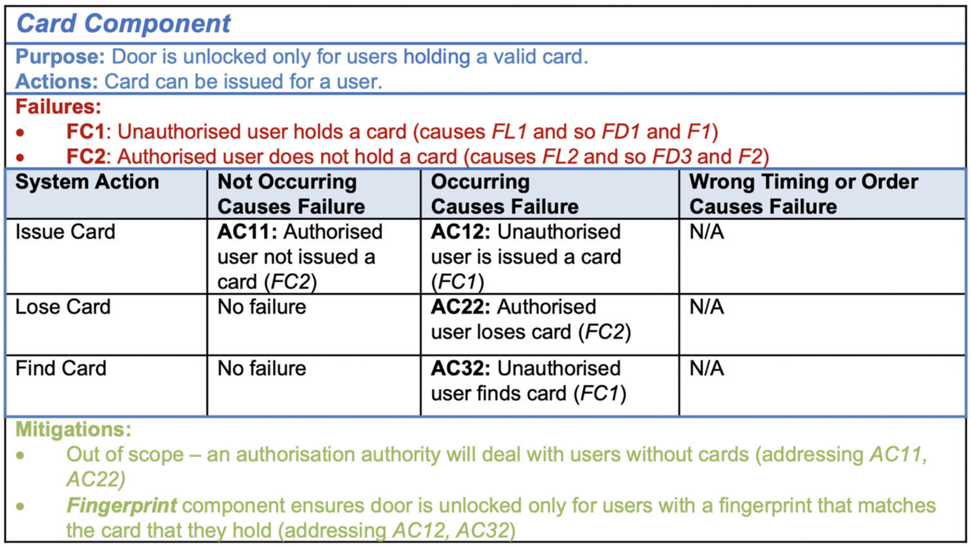
Fig. 14 Alarm component, action analysis table

component at this level. We stop at this point where we can implement the component rigorously.

The final control abstraction diagram, including all of the introduced components, is shown in Fig. 1. The control analysis is gradually built by introducing one component at each step. The design components were added, in the order shown from right to left: door, lock, alarm, card and finger print, with associated actions, conditions and inputs. The gatekeeper agent was introduced at the point when the lock component was introduced since this is the first component to have a physical control interface. As the control abstraction diagram is made more concrete, the checking of authorisation (authorised condition arrow) is shifted in each level depending on the role of the introduced component.

5 SHARCS workflow

In this section, we provide a specification of the workflow for the various steps of the SHARCS process, independently of the Tokeneer case study. Figure 17 illustrates the overall SHARCS process showing the main inputs and outputs of each level of analysis, while Fig. 18, described later, gives



Card Component

Purpose: Door is unlocked only for users holding a valid card.
Actions: Card can be issued for a user.

Failures:

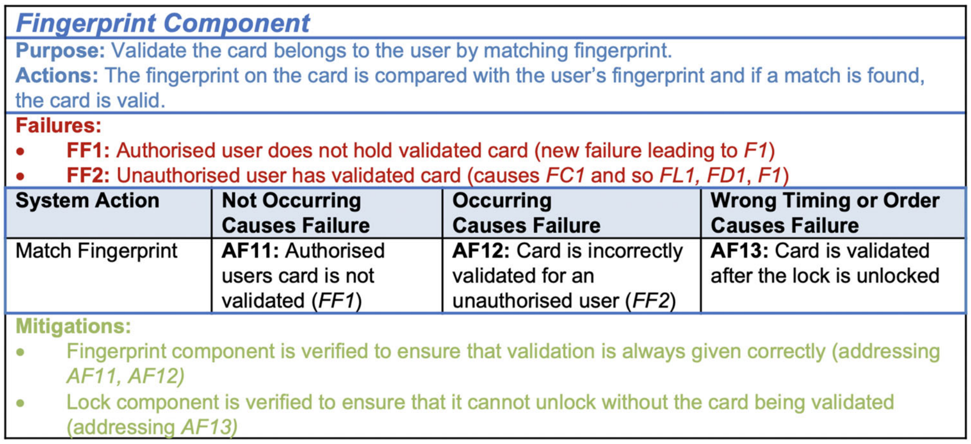
- **FC1**: Unauthorised user holds a card (causes *FL1* and so *FD1* and *F1*)
- **FC2**: Authorised user does not hold a card (causes *FL2* and so *FD3* and *F2*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Issue Card	**AC11:** Authorised user not issued a card (*FC2*)	**AC12:** Unauthorised user is issued a card (*FC1*)	N/A
Lose Card	No failure	**AC22:** Authorised user loses card (*FC2*)	N/A
Find Card	No failure	**AC32:** Unauthorised user finds card (*FC1*)	N/A

Mitigations:

- Out of scope – an authorisation authority will deal with users without cards (addressing *AC11, AC22*)
- ***Fingerprint*** component ensures door is unlocked only for users with a fingerprint that matches the card that they hold (addressing *AC12, AC32*)

Fig. 15 Card component, action analysis table



Fingerprint Component

Purpose: Validate the card belongs to the user by matching fingerprint.
Actions: The fingerprint on the card is compared with the user's fingerprint and if a match is found, the card is valid.

Failures:

- **FF1:** Authorised user does not hold validated card (new failure leading to *F1*)
- **FF2:** Unauthorised user has validated card (causes *FC1* and so *FL1, FD1, F1*)

System Action	Not Occurring Causes Failure	Occurring Causes Failure	Wrong Timing or Order Causes Failure
Match Fingerprint	**AF11:** Authorised users card is not validated (*FF1*)	**AF12:** Card is incorrectly validated for an unauthorised user (*FF2*)	**AF13:** Card is validated after the lock is unlocked

Mitigations:

- Fingerprint component is verified to ensure that validation is always given correctly (addressing *AF11, AF12)*
- Lock component is verified to ensure that it cannot unlock without the card being validated (addressing *AF13)*

Fig. 16 Fingerprint Component, action analysis table

more detail of the process within each level (i.e. expanding the two dashed boxes in Fig. 17)[2].

Figure 17 shows the three main phases of the process:

1. System level analysis and abstract modelling
2. Component level analysis and refinement modelling
3. Consolidation

The steps of Phase 1 were described in some detail in Sect. 3 and those of Phase 2 in Sect. 4. The main input to the system level analysis is a system requirements document. The requirements serve as the basis for construction of control abstraction diagrams and hence identification of system actions, potential failures, derived component and derived requirements. These are used as input to the next component level analysis. As demonstrated in Sect. 4, component level analysis is repeated for the various components that are introduced, hence the self loop on Phase 2 in Fig. 17.

Phase 3 is the consolidation phase of analysis where the outcomes are integrated to deliver the structured requirements and failures hierarchies, supported by illustrating scenarios, and a verified refinement chain of Event-B models. At the consolidation stage, we have enough design to draw a concrete control diagram which corresponds to the level at which a standard STPA analysis is typically undertaken.

Note that, in our case studies, each analysis phase focuses on one single component. In principle, it is also possible to consider several components interacting in the same phase. However, to keep each stage simple we prefer not to do this. We introduce a single component in the refined model along with enough behaviour about its environment and other control system components that it interacts with, to be able to validate and verify it. There may be examples where a components interaction is so tightly coupled with another component that it makes sense to consider them together, but

[2] The notation used in Figs. 17 and 18 is a standard notation for describing work processes called 'solution-patterns', (see https://vvpatterns.ait.ac.at/about-vv-patterns/).

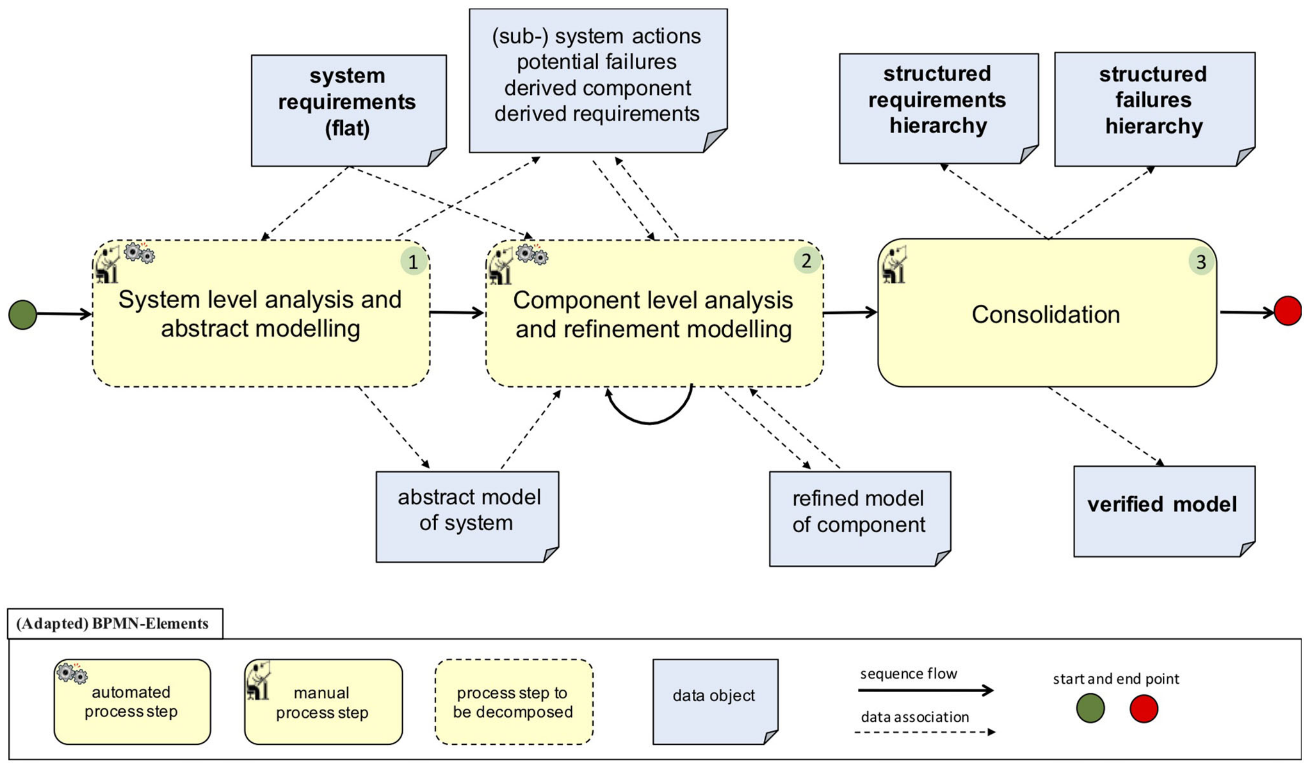

system requirements (flat)
(sub-) system actions potential failures derived component derived requirements
structured requirements hierarchy
structured failures hierarchy
1
System level analysis and abstract modelling
2
Component level analysis and refinement modelling
3
Consolidation
abstract model of system
refined model of component
verified model
(Adapted) BPMN-Elements
automated process step
manual process step
process step to be decomposed
data object
sequence flow
data association
start and end point

Fig. 17 SHARCS high level workflow

so far we envisage that our linear approach will deal with most cases.

5.1 Workflow for individual abstraction levels

Within each level of analysis (i.e. the box 1 (system level) and box 2 (component level) in Fig. 17), we perform the process shown in Fig. 18. There are slight differences for the system level and the component level process. The differences are outlined here and also highlighted previously in Sect. 4.

The inputs to the process for a level are 1) the system or sub-system (component) requirements, and for sub-systems only: 2) the control abstraction diagram and 3) formal model from the previous level.

The steps involved in system/component analysis (Steps 1 to 6 in Fig. 18) are as follows:

1. **Control analysis:** The system requirements and outline design from the previous phase (if any) are used to construct a control abstraction diagram showing the actors involved and the information flow and control between them. This diagram identifies the control actions needed in the next step. Except for the top-level system, the diagram is an elaboration of the previous level control abstraction diagram.
2. **Action analysis:** The control abstraction diagram is used to identify behavioural actions which are analysed to identify potentially insecure or unsafe actions. An action analysis table is constructed to analyse all the possible actions that can occur at the system/component level and identifies the resulting failures caused by that action occurring, not occurring or occurring with the wrong timing or order. This leads to identification of conditions for controls that are intended to prevent failures in the system.
3. **Formal modelling:** The actions identified in the control abstraction diagram are then formalised in an Event-B model. For sub-systems, this is done by refining the abstract model from the previous level and altering it to reflect the introduction of the component sub-system (the subject of this phase) in accordance with the outline design from the previous phase. Control actions are modelled by events that alter the state of the environment (including other components) and conditions are modelled as guards of these events. Note that, as well as modelling the component itself, we need to extend the model to exercise its interfaces with the environment including other component sub-systems. It is important to ensure that we introduce enough of the component's environment (including abstract models of other sub-systems) in order to validate and verify the component under analysis. Correctness of Event-B refinement ensures that the abstract properties are preserved by a refined model.

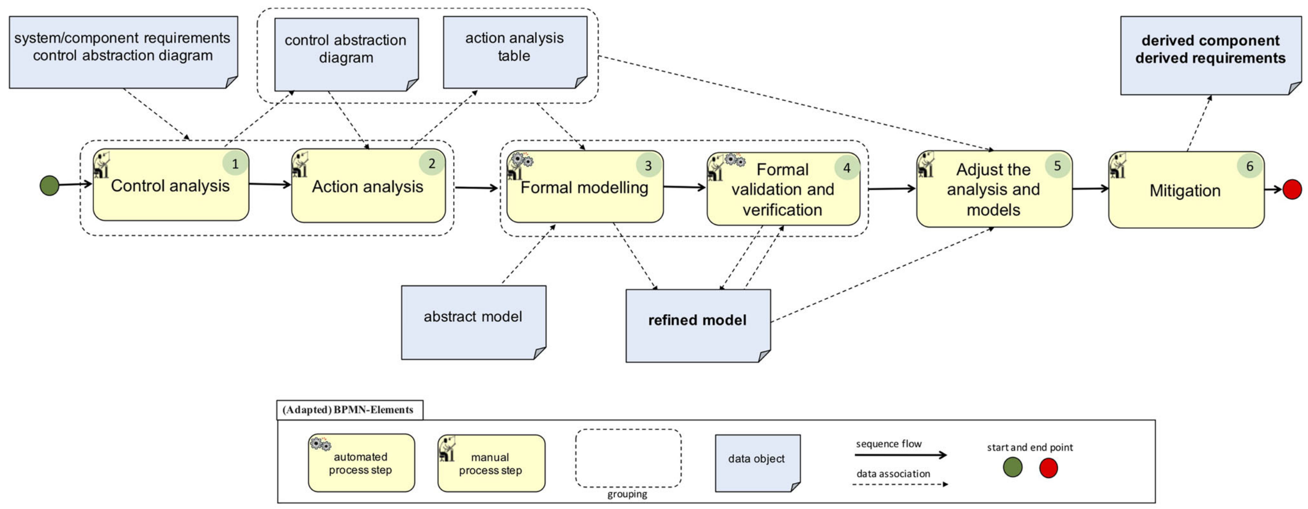


Fig. 18 Workflow for a single level of abstraction

4. **Formal validation and verification:**
   - **Validation using scenarios.** The formal model is validated using model animation, visualisation and scenario playing tools (ProB and the Scenario Checker described earlier). This validation stage ensures that the model behaves as expected by domain experts. The model should include all relevant external (environment) behaviour scenarios (including relevant external faults) as well as the internal system functionality.
   - **Verification using theorem provers.** When the model behaves in an appropriate way it is verified (using theorem provers supported by model checking) to ensure that the critical invariant properties of the system (representing safety and security properties for example) are maintained. Any remaining verification failures represent design faults. Note that some design faults will only cause system failures in the presence of environmental faults, which is why it is important to first validate the model to ensure it embodies such scenarios.
5. **Adjust the analysis and models:** Design faults are corrected and the model is revised accordingly so that verification failures are eliminated. In most cases, these improvements will involve strengthening the functionality, and hence derived requirements, allocated to component sub-systems, or in some cases introducing additional component sub-systems.
6. **Mitigation and outline design for next phases:** In this stage, each insecure or unsafe control action is considered and addressed as follows. Mitigating actions are proposed to prevent that action from leading to a failure. In some cases the mitigation might be to focus verification or validation activities in order to provide a particular safety or security argument. In other cases, the mitigation will be addressed by the next level design of the system. For these cases, an outline design is proposed to address the potentially insecure/unsafe control actions. The design may either be invented (in the pure case) or given (in the pragmatic case). The outline design identifies the sub-system components and provides their broad derived requirements which are used as inputs to the next phase when the relevant sub-system is analysed. Alternatively, we may have reached a stage where we feel this component does not need sub-components; it is manageable enough for us to implement without further sub-division. In this case the design consists of the derived requirements which have been validated and are verified to meet the safety and security constraints of this component.

5.2 Consolidation phase

In this phase, we collate the results of the hierarchical analysis to produce an overview of the derived control structure and the verified derived requirements. This results in the hierarchical component structure, Fig. 3, which has already been introduced and discussed in Sect. 2.1.

The final refined version of scenarios used in the hierarchical validation form an important output from the analysis as they illustrate scenarios that could potentially lead to a failure, and provide verification evidence that the design is robust enough to mitigate them. These are required items in ED203A [2], for example.

6 Related work

In this section, we discuss previous studies in the verification of Tokeneer (Sect. 6.1), informal approaches for developing safe and secure systems based on STPA (Sect. 6.2), and other approaches combining formal methods with STPA (Sect. 6.3).

6.1 Previous studies in verification of Tokeneer

The Tokeneer system is the subject of a case study by Capgemini (formerly Praxis) for developing high quality secure systems [16] using a correctness-by-construction approach. In particular, a requirements analysis process (called REVEAL) was used to produce a System Requirements Specification (SRS). An important reason for producing the SRS was to identify the system boundaries and an agreement on the system requirements with all stakeholders. In addition, the security target of the system was defined which is the basis for developing the system security properties. Based on the SRS and the security target, the formal system specification and design were written in the Z notation [17] and formal properties were verified in Z. In the approach of [16], the full SRS incorporating all the system components was constructed prior to the development of formal specifications. With our approach, we elide many of the components in the higher levels of abstraction in both requirements analysis and formal modelling, then introduce system components in an incremental and hierarchical manner, analysing component requirements in more depth and refining formal models in tandem. Thus, in our approach, the iterative interactions between formal modelling and requirements analysis help to derive the requirements of the sub-components from the design and high-level requirements.

Rivera et al. [18] remodelled the Tokeneer specifications in Event-B based on the Z specification of [16]. Rodin was used to discharge the proof obligations and the EventB2Java plug-in to Rodin was used to translate the final model into Java code. The work is focussed on assessing the capability of Rodin and its associated plug-ins to verify a formal model and generate executable code. Our work addresses the interactions between critical requirements analysis and formal modelling, and showing the link between properties of the components and system properties through a hierarchy of abstractions.

Similarly, Foster et al [19] relied on the security requirements of [16] and verified security properties against the functional formal specification of the Tokeneer system using Isabelle/SACM. The work focuses on the use of formal proof in building an assurance case for developing secure systems and utilises Goal Structure Notation (GSN) for integrating formal proof into the assurance case. Our consolidation phase plays a similar role by linking the flow of component requirements up to system requirements together with verified formal models at each abstraction level.

6.2 Safety and security in STPA

Young and Leveson [20] proposed an analysis method called an STPA-Sec, which is based on the top-down safety analysis method (STPA). This method uses an incorporative team involving security engineers, operations, and domain experts of a target system, in order to identify the potential security constraints for preventing a target system from entering vulnerable states that results in threats/losses. Although it is similar to traditional STPA, each control action is examined under the sort of conditions that identify a loss/threat scenario. Specifically, STPA-Sec focuses on identifying vulnerable states with a global system structure in order to prevent and avoid threats that might be exploited and eventually lead to losses.

Friedberg et al. [21] developed an analysis methodology for both security and safety based on STPA [3] and STPA-Sec [20], called STPA-SafeSec. The main aim of STPA-SafeSec is to choose effective mitigation strategies in order to ensure system safety and security. Therefore, the unified form of safety and security would be identified while choosing possible mitigation strategies, e.g. a possible security-related vulnerability might not be resolved when a piece of system equipment cannot be replaced, and based on the vulnerability, the control system can enforce the related safety constraint and mitigate the violation of a loss/threat.

Pereira et al. [22] argue that STPA-Sec [20] and STPA-SafeSec [21] both have a lack of extensive experience in real case studies. Therefore, they propose an approach that combines STPA and guidelines obtained from the safety assessment framework called NIST SP800-30 [23]. In addition, they claim that this standard should be considered because many organisations in the United States align to it. More specifically, the aim of this approach is to address safety at the high-level of components (system), while threats and vulnerabilities would be considered at the low-level of components (subsystem). This way of viewing a system would help to identify the safety and security workflows and where they may need to overlap.

6.3 Combining formal methods with STPA

Based on the hybrid methodology of STPA and NIST SP800-30 [24] proposed by Pereira et al. [22], Howard et al. [8] 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. [25] also apply a similar approach to [8] for generating safety and security requirements. Event-B has previously been combined with STPA by Colley and Butler [6] for safety analysis, again using STPA to guide the identification of safety requirements and Event-B to verify mitigation against hazardous states. [6, 8, 25] only support requirements analysis at a single abstraction level rather than the hierarchical approach that we support.

STPA has also been combined with other formal methods. In [26], 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. For verification, a behaviour model corresponding to the controller's behaviour and constrained by the STPA requirements is constructed, which can be verified against the formalised STPA requirements using model checking. Hata et al. [27] formally model the critical constraints derived from STPA as pre and post conditions in VDM++. Thomas and Leveson [28] 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.

6.4 Combining design methods with Event-B

Event-B has been used together with other design methods to formally ensure the safety and security of systems. In [29], the authors proposed an approach where designs written in AADL are translated to Event-B models to prove their consistency, in particular, using refinement to break down the complexity of the systems. Compared to our approach, the method in [29] structured the refinement based on features of AADL, rather than the actual systems under development.

In [30], a lightweight approach for connecting SysML and Event-B is proposed. The approach includes a requirement interchange system that supports the development of SysML and Event-B models iteratively. Compared to this paper, refinement is not yet incorporated into the approach in [30] for the gradual introduction of the model details. Similarly, in [31], the authors present a monolithic framework containing two phases: (1) requirements analysis including threat modelling using STRIDE and (2) design and verification formally using Event-B. The main challenge with monolithic approaches like the one in [31] is scalability, i.e. managing the models when the systems become more complex. On the other hand, in [32], the authors propose a methodology for constructing safety cases from hazard analysis techniques and Event-B. However, the link between the hazard analysis techniques (e.g. HAZOP, FMEA, etc.) and formal modelling in Event-B is simply by the set of requirements (output of the hazard analysis techniques) and assumed to be complete. We showed here in our paper that the link between analysis techniques (here STPA) can be embedded deeply within the framework and benefit from the formal modelling steps as well.

7 Contributions

7.1 Hierarchical and traceable

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. The abstraction-based hierarchical approach is a key contribution of this paper. We believe that the analysis and resulting evidence may be suited to the certification demands of standards such as the EUROCAE ones cited above.

7.2 Incremental, control abstraction diagrams

While STPA requires consideration of a complete closed system (which may consist of a hierarchical control structure given in a control structure diagram), 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. To apply the analysis at an abstract level we need the ability to model control structures abstractly and we introduced 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.

The analysis is applied incrementally with different abstraction levels until reaching the complete concrete control structure, thus factorising the complexity of the analysis across multiple abstraction levels in an incremental manner.

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.

7.3 Refinement strategy

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. The control abstraction diagrams help visualise the formal Event-B model at a particular level.

7.4 Traceable evidence

Initially, we envisaged that SHARCS analysis could be performed in the early stages of a new design when no prior conception exists concerning the components that will be involved. In this situation the hierarchical analysis could be used to drive the design with suitable components being created at each level to address the requirements derived for that level. However, we imagine that this purist approach is rarely seen in practice and it is more likely that, by the time a detailed analysis such as the one we propose is performed, the system structure will at least have been outlined in terms of potential components and their requirements. For this, more pragmatic, situation we see the analysis as an essential verification process, *firming up* the allocation of derived requirements to components and providing rigorous justification for the design along with traceable evidence. For the case studies in the paper, requirements exist and already identify some of the components and what they will do. However, the analysis raises some issues with requirements and depending on the design choices made, this could result in revised requirements leading to different designs.

7.5 Evidence for certification

Validation and verification artefacts (system requirements, component requirements, action analysis, formal models, validation scenarios, proofs) and traceability links between them provide important evidence for certification purposes. The ED-202A / DO-326A standard [1] for specifying the airworthiness security process specification requires the security development activities to include security requirements generated as part of the system requirements. These system security requirements should be subject to the same development requirements and assurance actions as safety related mechanisms. Our hierarchical approach supports the derivation and verification of system security requirements using analysis and formal methods that are well known in the safety requirements assurance domain.

7.6 Human behaviour

Systems often involve significant interaction with human users who are expected to behave in a certain way. Consequently any safety or security analysis has to make assumptions about user behaviour. In our examples we illustrate how human behaviour can be incorporated into the STPA-based analysis and formalised in the Event-B models. Analysing and formalising user behaviour in this way, makes explicit the extent to which it is relied on and the consequences of unexpected or deviant behaviour. Again this is useful as justifying evidence for the certification of systems.

Our case study led us to consider security related to human behaviour, emphasising the need to make assumptions about user behaviour. In contrast to protocols where only machines communicate with each other and precisely follow the protocol specification, new opportunities for attacks arise when humans are involved [33]. It is possible that users do not understand what they should and should not do and even knowledgeable users may neglect to perform some protocol steps due to carelessness. We examined our formal model for the secure door component of Tokeneer in two ways: rigid conditions and relaxed conditions, where rigid conditions result in less assumptions about user behaviour and relaxed conditions rely on more assumptions about user behaviour to prevent attacks.

8 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.

Although STPA supports hierarchical control structures, it does not support an incremental approach through abstraction/refinement. In contrast, SHARCS provides a systematic incremental and hierarchical approach supporting abstractions followed by refining those abstractions. Investigating the hierarchy of control structure and associated failures using the STPA control diagrams is based only on the human judgement, while SHARCS control abstraction diagrams supports incremental introduction of components and associated failures based on the suggested mitigation(s) at the earlier stage. Also SHARCS control abstraction diagrams support extra notations, like *condition* arrows, to provide the base for rigour translation to the corresponding entities in the formal model, like action *guards* in the Event-B model.

Currently the proposed abstraction control diagrams are not interpreted formally; they help to represent the actors, components and the interactions/conditions between them in a succinct visual way, and the formality is provided in the Event-B model. In the future work, we intend to address a rigorous approach to translation of the diagrams into Event-B formal entities. We will also investigate partitioning of the diagrams to focus on relevant components to further support scalability. For this purpose, we believe we can benefit from the decomposition technique [34] already supported by the Event-B.

One of our motivations is to provide a rigorous security risk analysis method for the avionics domain. The Eurocae standard, ED203A [2] gives guidance on what is needed in security risk assessment methods to gain certification. The standard avoids mandating any particular method but gives STPA-SEC as an example (see Appendix G of [2]). It suggests that the intial steps of STPA, such as identification of losses and accidents, contribute to process activities required by ED202A [1], such as definition of security scope, identification of assets, definition of security perimeter and specification of security environment. The identification of insecure control actions, their causes, scenarios and mitigation address the ED202A process requirements for security risk assessment, threat condition evaluation and threat scenario identification. Our approach of supporting the STPA analysis with formal modelling strengthens these activities and hence should aid certification. Furthermore, by extending the analysis into sub-system components we provide a deeper explanation of a security threat in terms of its effect on the system, giving strong and traceable evidence of how the derived requirements address security threats. In the future work, we intend to illustrate in more detail how our proposed method relates to the required activities of ED202A.

One of our outputs from the consolidation phase is the structured failure hierarchy, similar to a fault tree. One difference of this hierarchy with a fault tree is that our structured failure hierarchy (e.g. Fig. 4) is derived in an incremental manner in tandem with the hierarchical analysis while the components are gradually introduced. A typical fault-tree analysis is applied to a detailed design where all of the components of the design and their interactions are already known before the analysis is performed. For future work, we will investigate how our approach may provide a incremental way of deriving fault trees.

We have evaluated the method using two case studies; one focussing on security and the other on safety. The hypothesis is that the method provides an effective analysis of critical systems to ensure that threats and hazards are mitigated in the design by derived component requirements. The case studies support this hypothesis. The main threat to validity, as with any case study, is whether the case studies are representative of problems in general and have fully tested the method. The case studies are realistic problems representative of the domain, but do they test the kinds of properties addressed in the method? The method covers two kinds of property; safety (i.e. something bad does not happen) and enabledness as a restricted form of liveness (i.e. some things should not be prevented, but we do not deal with eventuality). The Tokeneer case study covers both kinds of property; unauthorised users can not enter the enclave and authorised users are not prevented from entering the enclave. In the level crossing case study we only discussed the safety property, “avoid collisions between trains and cars”, but the full analysis also deals with enabledness since cars must be able to pass when there is no train coming. The case studies are both well known so the results may be affected by prior knowledge of the systems. In the future, we will seek to apply the method to less familiar problems.

For our case studies, influenced by the Event-B refinement method, we used a linear sequence of analyses, considering one component after another and making corresponding refinements to our formal model. While this formalisation and proof approach was effective, in future work, we will explore more flexible strategies. For example, the lock and alarm components could be analysed in parallel by different teams making separate refinements from the door level. These could be merged back to a single refinement (if certain conditions are met in the refinements) or left as independent analyses. Different case studies may be found where sibling components interact more strongly. In this case, it may still be possible to focus on one component first by providing sufficient abstraction of the other or in some cases the components may be so interdependent that separate analyses are not feasible. Existing techniques for composition of Event-B models [35] will be important in addressing this challenge.

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.

Appendix—Level crossing SHARCS analysis

See Figs. 19, 20 and 21.

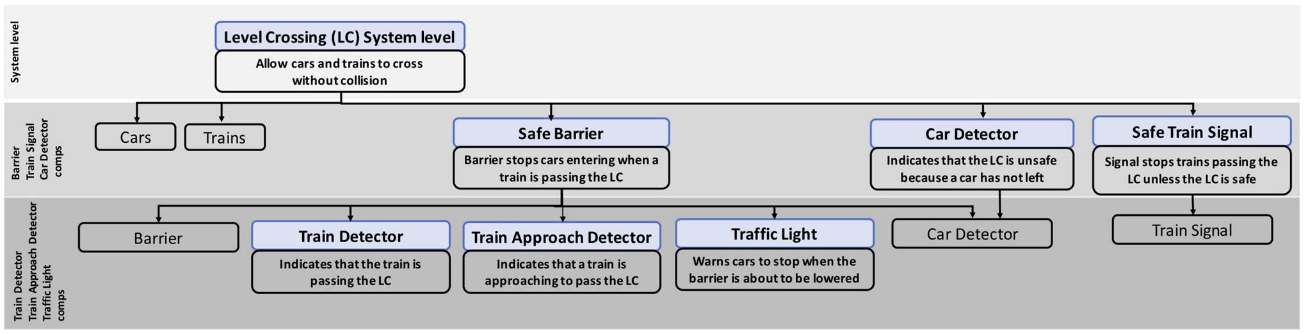
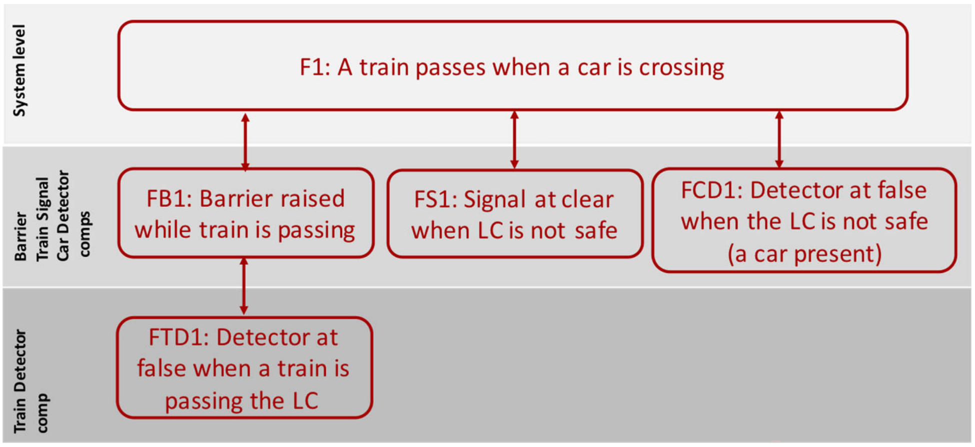

System level
Level Crossing (LC) System level
Allow cars and trains to cross without collision
Barrier Train Signal Car Detector comps
Cars
Trains
Safe Barrier
Barrier stops cars entering when a train is passing the LC
Car Detector
Indicates that the LC is unsafe because a car has not left
Safe Train Signal
Signal stops trains passing the LC unless the LC is safe
Train Detector Train Approach Detector Traffic Light comps
Barrier
Train Detector
Indicates that the train is passing the LC
Train Approach Detector
Indicates that a train is approaching to pass the LC
Traffic Light
Warns cars to stop when the barrier is about to be lowered
Car Detector
Train Signal

Fig. 19 Level crossing: hierarchical component design, flow down requirements

Fig. 20 Level crossing: hierarchical failures for one of the system level failures


System level
F1: A train passes when a car is crossing
Barrier Train Signal Car Detector comps
FB1: Barrier raised while train is passing
FS1: Signal at clear when LC is not safe
FCD1: Detector at false when the LC is not safe (a car present)
Train Detector comp
FTD1: Detector at false when a train is passing the LC

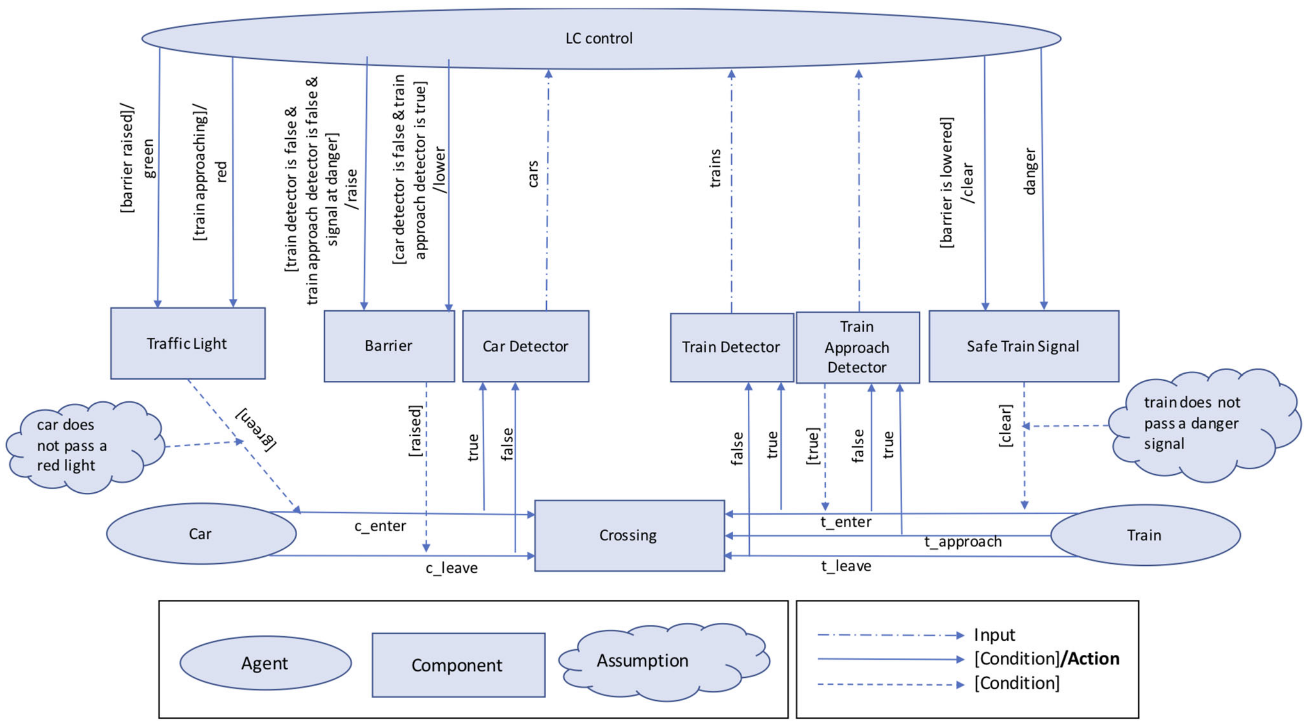


Fig. 21 Level crossing: last level abstraction control diagram

References

1. Eurocae (2014) ED-202A—Airworthiness Security Process Specification. https://eshop.eurocae.net/eurocae-documents-and-reports/ed-202a/
2. Eurocae (2018) ED-203A—Airworthiness Security Methods and Considerations. https://eshop.eurocae.net/eurocae-documents-and-reports/ed-203a/
3. Leveson NG, Thomas JP (2018) STPA handbook. Cambridge
4. Young W, Leveson NG (2014) 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
5. Abrial J-R (2010) Modeling in event-B: system and software engineering. Cambridge University Press, Cambridge
6. Colley J, Butler M (2013) A formal, systematic approach to STPA using event-B refinement and proof. In: 21th safety critical system symposium
7. Howard G, Butler MJ, Colley J, Sassone V (2017) 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
8. Howard G, Butler MJ, Colley J, Sassone V (2019) 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
9. Omitola T, Rezazadeh A, Butler M (2019) Making (implicit) security requirements explicit for cyber-physical systems: a maritime use case security analysis. Database and expert systems applications. Springer, Berlin
10. Fathabadi S, Snook C, Dghaym D, Hoang TS, Alotaibi F, Butler M (2023) Designing critical systems using hierarchical STPA and Event-B. In: ABZ 2023: rigorous state-based methods
11. Praxis: Tokeneer. https://www.adacore.com/tokeneer. Accessed May 2020
12. Avizienis A, Laprie J, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Depend Secure Comput 1(1):11–33
13. Abrial J-R, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in Event-B. Softw Tools Technol Transf 12(6):447–466
14. Leuschel M, Butler M (2008) ProB: an automated analysis toolset for the B method. Softw Tools Technol Transf (STTT) 10(2):185–203
15. Snook C, Hoang TS, Dghaym D, Fathabadi AS, Butler M (2020) Domain-specific scenarios for refinement-based methods. J Syst Archit
16. Barnes J, Chapman R, Johnson R, Widmaier J, Cooper D, Everett B (2006) Engineering the Tokeneer enclave protection software. In: Proceedings of IEEE international symposium on secure software engineering
17. Spivey JM (1989) The Z notation—a reference manual. Prentice Hall International Series in Computer Science
18. Rivera V, Bhattacharya S, Cataño N (2016) Undertaking the tokeneer challenge in event-B. In: Proceedings of the 4th FME workshop on formal methods in software engineering, FormaliSE@ICSE 2016. ACM
19. Foster S, Nemouchi Y, Gleirscher M, Wei R, Kelly T (2021) Integration of formal proof into unified assurance cases with Isabelle/SACM. Formal Aspects Comput 33:855–884
20. Young W, Leveson NG (2013) Systems thinking for safety and security. In: Annual computer security applications conference, ACSAC '13. ACM
21. Friedberg I, McLaughlin K, Smith P, Laverty DM, Sezer S (2017) STPA-SafeSec: safety and security analysis for cyber-physical systems. J Inf Secur Appl 34:183–196
22. Pereira DP, Hirata CM, Pagliares R, Nadjm-Tehrani S (2017) Towards combined safety and security constraints analysis. In: Computer safety, reliability, and security—SAFECOMP 2017

workshops, ASSURE, DECSoS, SASSUR, TELERISE, and TIPS. Springer

23. Blank RM, Secretary A (2011) Guide for conducting risk assessments
24. Group J.T.F.T.I.I.W (2012) SP 800–30 revision 1: guide for conducting risk assessments. Technical report, National Institute of Standards & Technology
25. Dghaym D, Hoang TS, Turnock SR, Butler M, Downes J, Pritchard B (2021) An STPA-based formal composition framework for trustworthy autonomous maritime systems. Saf Sci 136:105139
26. Abdulkhaleq A, Wagner S, Leveson N (2015) A comprehensive safety engineering approach for software-intensive systems based on STPA. Procedia Eng 128:2–11
27. Hata A, Araki K, Kusakabe S, Omori Y, Lin H (2015) Using hazard analysis stamp/STPA in developing model-oriented formal specification toward reliable cloud service. In: 2015 international conference on platform technology and service
28. Thomas J, Leveson N (2013) 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
29. Hadad ASA, Ma C, Ahmed AAO (2020) Formal verification of AADL models by event-B. IEEE Access 8:72814–72834
30. Thorburn R, Sassone V, Fathabadi AS, Aniello L, Butler MJ, Dghaym D, Hoang TS (2022) A lightweight approach to the concurrent use and integration of SYSML and formal methods in systems design. In: Kühn T, Sousa V (eds) Proceedings of the 25th international conference on model driven engineering languages and systems: companion proceedings, MODELS 2022, Montreal, Quebec, Canada, 23–28 Oct, 2022, pp 83–84. https://doi.org/10.1145/3550356.3559577
31. Seo J, Kwak J, Kim S (2023) Formally verified software update management system in automotive. In: Symposium on vehicles security and privacy (VehicleSec) 2023
32. Prokhorova Y, Laibinis L, Troubitsyna E (2015) Facilitating construction of safety cases from formal models in event-B. Inf Softw Technol. https://doi.org/10.1016/j.infsof.2015.01.001
33. Basin DA, Radomirovic S, Schmid L (2016) Modeling human errors in security protocols. In: IEEE 29th computer security foundations symposium
34. Silva R, Pascal C, Hoang TS, Butler MJ (2011) Decomposition tool for Event-B. Softw Pract Exp 41(2):199–208
35. Hoang TS, Dghaym D, Snook CF, Butler MJ (2017) A composition mechanism for refinement-based methods. In: 22nd international conference on engineering of complex computer systems, ICECCS 2017, Fukuoka, Japan, 5–8 Nov, 2017, pp 100–109 . https://doi.org/10.1109/ICECCS.2017.27