

Mechanised Assurance Cases with Integrated Formal Methods in Isabelle

Yakoub Nemouchi, Simon Foster, Mario Gleirscher, and Tim Kelly

University of York

`firstname.lastname@york.ac.uk`

Abstract Assurance cases are often required as a means to certify a critical system. Use of formal methods in assurance can improve automation, and overcome problems with ambiguity, faulty reasoning, and inadequate evidentiary support. However, assurance cases can rarely be fully formalised, as the use of formal methods is contingent on models validated by informal processes. Consequently, we need assurance techniques that support both formal and informal artifacts, with explicated inferential links and assumptions that can be checked by evaluation. Our contribution is a mechanical framework for developing assurance cases with integrated formal methods based in the Isabelle system. We demonstrate an embedding of the Structured Assurance Case Meta-model (SACM) using Isabelle/DOF, and show how this can be linked to formal analysis techniques originating from our verification framework, Isabelle/UTP. We validate our approach by mechanising a fragment of the Tokeneer security case, with evidence supplied by formal verification.

1 Introduction

Cyber-physical systems (CPS) control critical socio-technical processes prone to faults and other critical events with potentially undesired consequences. Such systems include autonomous vehicles, traffic flow control, patient monitoring, surgical robot assistants, and building security automation. Real-time concurrency of physical events and computation poses tough challenges in achieving high levels of assurance in verification and validation. Consequently, the benefits of CPS can only be harnessed if they acquire consumer trust and regulatory acceptance.

Safety cases [20, 23], and more generally assurance cases, are structured arguments, supported by evidence, intended to convince a regulator that a system is acceptably safe for application in a specific operating environment [18]. They are recommended by several international standards, such as ISO26262 for automotive applications. An assurance case consists of a hierarchical decomposition of requirements, through appropriate argumentation strategies, into further claims, and eventually supporting evidence. Several languages exist for expressing assurance cases, including the Goal Structuring Notation [23] (GSN), and the closely related Structured Assurance Case Metamodel¹ (SACM).

¹ Structured Assurance Case Metamodel: <http://www.omg.org/spec/SACM/>

Assurance case creation can be supported by model-based design, which utilises architectural and behavioural models over which requirements can be formulated [18]. However, safety cases can suffer from undermining logical fallacies and lack of evidence [17]. A proposed solution is formalisation in a machine-checked logic to enable verification of consistency and well-foundedness [26]. As confirmed by avionics standard DO-178C supplement DO-333, the evidence gathering process can also benefit from the rigour of formal methods. At the same time, we acknowledge that, (1) assurance cases are intended primarily for human consumption, and (2) that formal models must be validated informally [19]. Consequently, assurance cases will usually combine informal and formal content, and any tool must support this.

Our vision is a unified framework for machine-checked assurance cases, and with evidence provided by a number of integrated formal methods [16]. Such a framework can have a transformative effect in the field of assurance by harnessing results from automated formal verification to produce assurance cases undergirded by specific mathematical guarantees of their consistency and adequacy of the evidence. Moreover, it can provide a potential route to regulatory acceptance, through the production of mathematically verified safety certificates.

The contributions of this paper make a first step in this direction: (1) an implementation of SACM in the Isabelle interactive theorem prover [24], (2) a machine-checked domain-specific assurance language, and (3) integration of formal evidence from our verification framework, Isabelle/UTP [14]. Isabelle provides a sophisticated executable document model for presenting a graph of hyperlinked artifacts, like definitions, theorems, and proofs. The document model provides automatic and incremental consistency checking, and change analysis, where updates to model artifacts trigger rechecking. Such capabilities can support efficient maintenance and evolution of model-based assurance cases [20].

Moreover, the document model allows management of both informal and formal content, and access to a vast array of automated verification tools [29]. In particular, our own verification framework, Isabelle/UTP [14, 15], harnesses Hoare and He's Unifying Theories of Programming [21] (UTP) to provide verification facilities for a variety of programming and modelling languages with paradigms as diverse as concurrency, real-time, and hybrid computation. We validate our approach by mechanising an assurance case for the Tokeneer system [1], including the underlying formal model and verification of security functional requirements².

In §2 we outline preliminary materials: SACM, Isabelle, and the Isabelle/DOF ontology framework. In §3 we describe the Tokeneer system, and how it is assured and verified. In §4 we begin our contributions by describing the implementation of Isabelle/SACM and our assurance DSL in Isabelle. In §5 we describe how we model and verify Tokeneer using our verification framework, Isabelle/UTP. In §6 we describe the mechanisation of the assurance case for Tokeneer in Isabelle/SACM. In §7 we highlight related work, and in §8 we conclude.

² Supporting materials, including Isabelle theories, can be found on [our website](#)

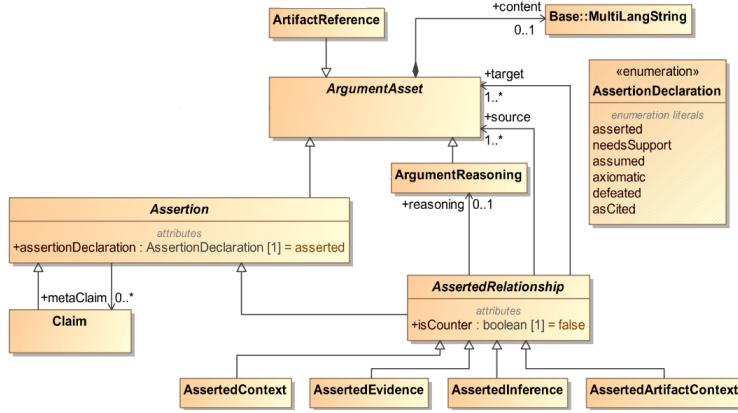


Figure 2: SACM Argumentation Meta-Model

2 Preliminaries

SACM. Assurance cases are often presented using a notation like GSN [23] (Figure 1), that shows the claims that are made, the argumentation strategies, the contextual elements, assumptions, justifications, and eventually evidence. SACM is an OMG standard meta-model for assurance cases [20, 28]. It aims at unifying and refining a variety of predecessor notations, including GSN [23] and CAE (Claims, Arguments, and Evidence), and is intended to be a definitive reference model.

SACM has three crucial concepts: arguments, artifacts, and terminology. An argument consists of a collection of claims, evidence citations, and inferential links between them. Artifacts manifest evidence, such as models, techniques, results, verification activities, and participants. Terminology is used to fix formal terms for the use in claims. Normally, claims are textual, but in SACM they can also contain structured expressions, which allows integration of formal languages.

The argumentation meta-model is shown in Figure 2. The base class is `ArgumentAsset`, which groups the argument assets, such as `Claims`, `ArtifactReferences`, and `AssertedRelationships` (which are inferential links). Every asset may contain a `MultiLangString` that provides a description, potentially in multiple natural and formal languages, and corresponds to contents of the shapes in Figure 1.

`AssertedRelationships` represent a relationship that exists between several assets. They can be of type `AssertedContext`, which uses an artifact to define context; `AssertedEvidence`, which evidences a claim; `AssertedInference` which describes

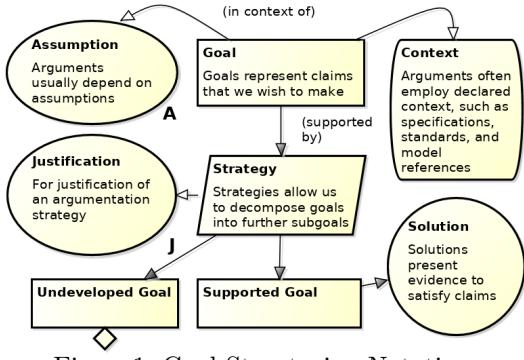


Figure 1: Goal Structuring Notation

explicit reasoning from premises to conclusion(s); or `AssertedArtifactSupport` which documents an inferential dependency between the claims of two artifacts.

Both `Claims` and `AssertedRelationships` inherit from `Assertion`, because in SACM both claims and inferential links are subject to argumentation and refutation. SACM allows six different classes of assertion, via the attribute `assertionDeclaration`, including `axiomatic` (needing no further support), `assumed`, and `defeated`, where a claim is refuted. An `AssertedRelationship` can also be flagged as `isCounter`, where counterevidence for a claim is presented.

Isabelle. Isabelle/HOL is an interactive theorem prover for higher order logic (HOL) [24], based on the generic framework Isabelle/Isar [30]. The former provides a functional specification language, and a large array of facilities for proof and automated verification. The latter has an interactive, extensible, and executable document model, which describes Isabelle theories. An Isabelle theory contains a sequence of executable markup commands with a semantics given in the meta-language SML.

Figure 3 gives an overview of the document model. The first section for *context definition* describes *imports* of existing theories, and *keywords* which give extensions to the concrete syntax. The second section is the body enclosed between *begin-end* which is a sequence of commands. Isabelle commands have a concrete syntax consisting of pre-declared top-level keywords (in blue), such as the command `ML`, followed by a “semantics area” enclosed between `<...>`. The keywords can be associated with optional attribute keywords (in green). The processing of the concrete syntax and any extensions is performed by SML code.

An Isabelle session is as an acyclic graph grouping several theories and their dependencies. When an edit is made to a document in a session, it is immediately processed and executed, with feedback provided to the user. For example, whenever the dependency structure of a document changes due to the removal, addition, or alteration of artifacts, Isabelle reruns the associated code and any dependencies. This feature makes Isabelle ideal for assurance cases, which have to be updated with every increment in system development

In addition to formal content, Isabelle theories can also contain informal commentary. The `text <...>` command is a processor for textual markup content containing a mixture of informal content, and links to formal document entities through *antiquotations* of the form `@{aqname ...}`. Antiquotations trigger a series of checks, for example the antiquotation `@{thm <HOL.ref1>}` checks if the theorem `HOL.ref1` exists within the underlying theory context, and if so inserts a hyperlink.

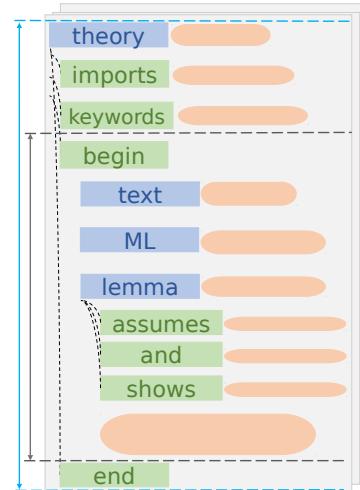


Figure 3: Document Model

Plugins, such as Isabelle/HOL, HOL-TestGen [3], and Isabelle/DOF [2] contain document models and conservative extensions, following the LCF approach. Isabelle/DOF [2] is a plugin that the Isabelle/Isar document model implemented with support for ontologies. The result is a machine-checked document model with formal hyperlinks between document instances of the modelled ontology.

The central component of Isabelle/DOF is the Isabelle Ontology Specification Language (IOSL), which describes the content of documents in terms of several document classes. Document classes can be linked to form a class model. We refer to [2] for examples to model document content within Isabelle/DOF. A document class is the main entity in IOSL and it is represented using the command `doc_class`, which creates a new class with a number of typed attributes. The attributes of `doc_class` can refer both to the standard HOL types such as `string`, `bool`, and also internal Isabelle meta-types such as `thm`, `term`, or `typ`, which represent theorems, logical terms, and types, respectively. This is because DOF ontologies sit at the meta-logical level, and so they can freely mix formal and informal content. This is our motivation for its use in mechanising SACM.

3 Running Example: Tokeneer

To demonstrate our approach, we use the Tokeneer Identification Station (TIS)³ illustrated in Figure 4, a system that guards entry to a secure enclave by ensuring that only authorised users are admitted.

The relevant physical infrastructure consists of a door, a fingerprint reader, a display, and a card (token) reader. The main function of the TIS is to check the stored credentials on a presented token, read a fingerprint if necessary, and then either unlatches the door, or denies entry. Entry is permitted when the token holds at least three data items: (1) an ID certificate, which identifies the user, (2) a privilege certificate, which stores a clearance level, and (3) an identification and authentication (I&A) certificate, which assigns a fingerprint template. When the user first presents their token the three certificates are read and cross-checked. If the token is valid, then a fingerprint is taken, which if validated against the I&A certificate, allows the door to be unlocked once the token is removed. An optional authorisation certificate is also written upon successful authentication which allows the fingerprint check to be skipped.

The TIS has a variety of other functions related to its administration. Before use, a TIS must be enrolled, meaning it is loaded with a public key chain and certificate, which are needed to check token certificates. Moreover, the TIS stores audit data which can be used to check previously occurred entries. The

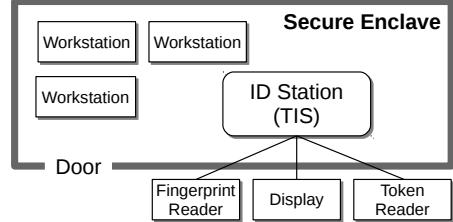


Figure 4: Tokeneer System Overview

³ Project website: <https://www.adacore.com/tokeneer>

TIS therefore also has a keyboard, floppy drive, and screen to configure it. Administrators are granted access to these functions. The TIS also has an alarm which will sound if the door is left open for too long.

Our objective is to construct an assurance case that argues that the TIS fulfils its security properties and complies to the Common Criteria (CC) standard [4]. CC supports a vendor in delivering a system compliant to a *security level* while a certification authority confirms compliance and further qualities. The standard defines *seven* Evaluation Assurance Levels (EALs), each a collection of Security Functional Requirements (SFRs) and Security Assurance Requirements (SARs) the system would have to meet. Formal Methods (FMs) are strongly recommended for EAL 5 and above. Now, one can either (a) use a pre-specified *set of general security properties*, or (b) develop an *application-specific set* with the potential of additional effort due to requirements analysis. The security of the TIS is assured according to (b) by demonstrating six SFRs [6], of which the first four are shown here and detailed below in section 5:

- SFR1** If the latch is unlocked by TIS, then TIS must be in possession of either a User Token or an Admin Token. The User Token must either have a valid Authorisation Certificate, or must have valid ID, Privilege, and I&A Certificates, together with a template that allowed TIS to successfully validate the user's fingerprint. Or, if the User Token does not meet this, the Admin Token must have a valid Authorisation Certificate, with role of "guard".
- SFR2** If the latch is unlocked automatically by TIS, then the current time must be close to being within the allowed entry period defined for the User requesting access.
- SFR3** An alarm will be raised whenever the door/latch is insecure.
- SFR4** No audit data is lost without an audit alarm being raised.

The *pioneering work* on the assurance of the TIS according to option (b) was carried out by Praxis High Integrity Systems and SPRE Inc. [1]. Barnes et al. performed security analysis, specification using Z, implementation in SPARK, and verification and test of the security properties. After independent assessment EAL 5 was achieved. This way, Tokeneer became a successful example of the use of FMs in assuring a system against CC.

4 Isabelle/SACM

In the following we encode SACM in Isabelle/DOF as an ontology, and then use it to provide a concrete syntax for our assurance case language. Our embedding implements assurance cases as meta-logical entities. We are not embedding assurance arguments in the HOL logic, as this would prevent the expression of informal reasoning and explanation. Rather, SACM is implemented as a datatype in SML, meaning that we can refer to entities like types, terms, and theorems as objects. Thus, certain claims can contain formal expressions, but others may have unstructured natural language. Thus, we faithfully represent the inherently semi-formal nature of assurance cases.

We focus on the **ArgumentationPackage**⁴ from Figure 2, as this is most relevant for the TIS argument we develop in §6. Different types of evidences and context,

⁴ We model all parts of argumentation, base, artifact and terminology packages in Isabelle/DOF, but omit details about these for space reasons.

modelled by the `ArtifactPackage`, can support a claim. The class `ArgumentAsset` is represented in Isabelle/DOF as follows:

```
doc_class ArgumentAsset = ArgumentationElement +
  content_assoc:: MultiLangString
```

Here, `ArgumentationElement` is a base class which `ArgumentAsset` inherits from, but is not discussed further. The `content_assoc:: MultiLangString` is an attribute modelling an association between `ArgumentAsset` and `MultiLangString` from the `BasePackage`. It allows classes inherited from `ArgumentAsset` to include content expressed in multiple languages, and also structured expressions, using the `TerminologyPackage`. Our implementation of `MultiLangString` allows us to embed a variety of informal and formal content utilising the Isabelle term language.

The class `ArgumentAsset` is inherited by three classes: (1) `Assertion`, which is a unified type for claims and their relationships; (2) `ArgumentReasoning`, which is used to explicate the argumentation strategy being employed; and (3) `ArtifactReference`, that evidences a claim with an artifact.

In Isabelle/DOF, `ArgumentAsset` is inherited as follows:

```
datatype assertionDeclarations_t =
  Asserted|Axiomatic|Defeated|Assumed|NeedsSupport
doc_class Assertion = ArgumentAsset +
  assertionDeclaration::assertionDeclarations_t
doc_class ArgumentReasoning = ArgumentAsset +
  structure_assoc::"ArgumentPackage option"
doc_class ArtifactReference = ArgumentAsset +
  referencedArtifactElement_assoc::"ArtifactElement set"
```

Here, `assertionDeclarations_t` is an Isabelle/HOL enumeration type, `set` is the set type, and `option` is the optional type. The attribute `assertionDeclaration` is of type `assertionDeclarations_t`, which specifies the status of assertions. The attribute `structure_assoc` is an association to the class `ArgumentPackage`, which is not discussed here. Finally, the attribute `referencedArtifactElement_assoc` is an association to the `ArtifactPackage` allowing claims to reference artifacts.

The class `Claim` is a leaf child class and inherits from the class `Assertion`. This means that an instance of `Claim` has a `gid`, a `MultiLangString` description, and can be `Axiomatic`, `Asserted`, etc. The other child class for `Assertion` is:

```
doc_class AssertedRelationship = Assertion +
  isCounter::bool
  reasoning_assoc:: "ArgumentReasoning option"
```

Here, `isCounter` specifies whether the target of the relation is refuted by the source, and `reasoning_assoc` is an association to `ArgumentReasoning`, to elaborate the strategy. `AssertedRelationship` models the relationships between elements of type `ArgumentAsset`. In addition to the inherited attributes from parent classes, the relationship classes have the attributes `source` and `target`. The `source` attribute carries the supporting elements and the `target` carries the supported elements.

From the SACM ontology, we create a number of Isabelle commands that create elements of the meta-model. Our approach gives a concrete syntax for SACM in terms of Isabelle commands as follows. Instances of concrete leaf child classes in the metamodel have concrete syntax consisting of an Isabelle top-level command. Instances of attributes of a leaf child class, including the inherited ones, have a concrete syntax represented by an Isabelle (green) subcommand. Instances of associations between leaf child classes have a concrete syntax represented by an Isabelle subcommand. The command has the name of the represented association and has an input with the type of the association of the underlying instance. A selection of the commands for SACM is shown below.

```
CLAIM <gid> CONTENT <MultiLangString>
ASSERTED_INFERENCE <gid> SOURCE <gid>* TARGET <gid>*
ASSERTED_CONTEXT <gid> SOURCE <gid>* TARGET <gid>*
ASSERTED_EVIDENCE <gid> SOURCE <gid>* TARGET <gid>*
ARTIFACT <gid> VERSION <text> DATE <text> CONTENT <MultiLangString>
```

`CLAIM` creates a new claim with an identifier (`gid`), and content described by a `MultiLangString`. `ASSERTED_INFERENCE` creates an inference between several claims. It has subcommands `SOURCE` and `TARGET` that are both lists of elements. The command ensures that the cited claims exist, otherwise an error message is issued. `ASSERTED_CONTEXT` similarly asserts that an entity should be treated as context for another, and `ASSERTED_EVIDENCE` associates evidence with a claim. The `ARTIFACT` command creates an evidential artifact, with description, date, and content.

Each command also has an associated antiquotation, which can be used to reference the entity type in a claim string. This is illustrated in Figure 5, which shows the interactive nature of the assurance case language. It represents an inferential link between a strategy and a justification (cf. Figure 1). An asserted inference called `Rel_A` has been created that attempts to link existing claims `Claim_A` and `Claim_B`. However, `Claim_B` does not exist, and so the error message at the top of the screenshot is issued. A textual element is then created which references `Rel_A` using the antiquotation class `@{AssertedInference ...}`. This also leads to an error, shown at the bottom, since `Rel_A` does not exist.

We have now developed our interactive assurance case tool. In the next section we begin to consider assurance of the Tokeener system, first considering formal verification of the security properties.

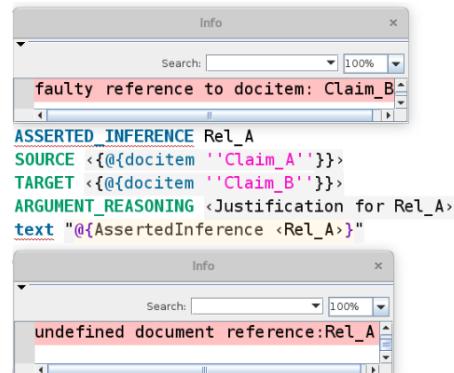


Figure 5: Relations in Isabelle/SACM

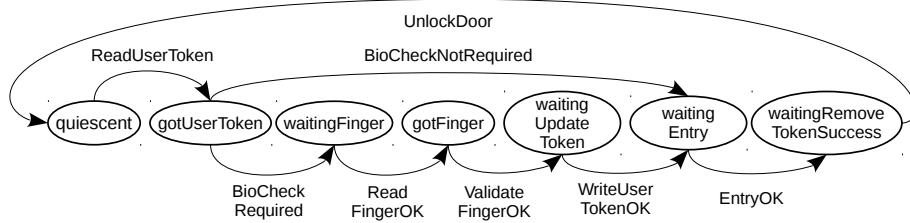


Figure 6: TIS Main States

5 Modelling and Verification of Tokeneer

In this section we formally model the TIS in Isabelle/UTP [14], in order to provide evidence for the assurance case. In [6], the six SFRs are argued semi-formally, but here we provide a formal proof. We focus on the formalisation and verification of the user entry part of SFR1, and describe the elements necessary for this.

The TIS behaviour, formalised by Praxis in the Z notation [5], uses an elaborate state space and a collection of relational operations. The state is bipartite, consisting of (1) the digital state of the TIS and (2) the variables shared with the real world, which are monitored or controlled, respectively. The TIS monitors the time, enclave door, fingerprint reader, token reader, and several peripherals. It controls the door latch, an alarm, a display, and a screen.

The specification describes a state transition system, illustrated in Figure 6 (cf. [5, page 43]), where each transition corresponds to an operation. Several operations are omitted due to space constraints. Following enrolment, the TIS becomes quiescent (awaiting interaction). `ReadUserToken` triggers if the token is presented, and reads its contents. Assuming a valid token, the TIS determines whether a fingerprint is necessary, and then triggers either `BioCheckRequired` or `BioCheckNotRequired`. If required, the TIS then reads a fingerprint (`ReadFingerOK`), validates it (`ValidateFingerOK`), and finally writes an authorisation certificate to the token (`WriteUserTokenOK`). If the access credentials are available (`waitingEntry`), then a final check is performed (`EntryOK`), and once the user removes their token (`waitingRemoveTokenSuccess`), the door is unlocked (`UnlockDoor`).

We mechanise the TIS model by first creating hierarchical state space types, with invariants adapted from the Z specification [5]. We define the operations using Dijkstra's guarded command language [10] (GCL) rather than the Z schemas directly, as GCL is easier to reason about and provides similar expressivity. Moreover, GCL is given a denotational semantics in UTP's alphabetised relational calculus [21], and so it is possible to prove equivalence with the corresponding Z operations. We use a variant of GCL that broadly follows the following syntax:

$$\mathcal{P} ::= \mathbf{skip} \mid \mathbf{abort} \mid \mathcal{P} ; \mathcal{P} \mid \mathcal{E} \longrightarrow \mathcal{P} \mid \mathcal{P} \sqcap \mathcal{P} \mid \mathcal{V} := \mathcal{E} \mid \mathcal{V} : [\mathcal{P}]$$

Here, \mathcal{P} is a program, \mathcal{E} is an expression or predicate, and \mathcal{V} is a variable. The language provides the usual syntax for sequential composition, guarded commands, non-deterministic choice, and assignment. We also adopt a framing

operator $a:[P]$ which states that P can refer only to variables in the namespace a , and all other variables remain unchanged [14, 15].

We now introduce the TIS state space, on which the state machine will act.

$$\begin{aligned} IDStation &\triangleq \left[\begin{array}{l} currentUserToken : TOKENTRY, currentTime : TIME, \\ userTokenPresence : PRESENCE, status : STATUS, \\ issuerKey : USER \rightarrow KEYPART, \dots \end{array} \right] \\ Controlled &\triangleq [latch : LATCH, alarm : ALARM, \dots] \\ Monitored &\triangleq [now : TIME, finger : FINGERPRINTTRY, \\ &\quad userToken : TOKENTRY, \dots] \\ RealWorld &\triangleq [mon : Monitored, ctrl : Controlled] \\ SystemState &\triangleq [rw : RealWorld, tis : IDStation] \end{aligned}$$

We define five state space types that describe the TIS state, the controlled variables , monitored variables, real-world, and the entire system, respectively. The controlled variables include the physical latch, the alarm, the display, and the screen. The monitored variables correspond to time (*now*), the door (*door*), the fingerprint reader (*finger*), the tokens, and the peripherals. *RealWorld* combines the physical variables, and *SystemState* composes the physical world and TIS.

Variable *currentUserToken* represents the last token presented to the TIS, and *userTokenPresence* indicates whether a token is currently presented. The variable *status* is used to record the state the TIS is in, and can take the values indicated in the state bubbles of Figure 6. Variable *issuerKey* is a partial function representing the public key chain, which is needed to authorise user entry.

We now specify a selection of the operations over this state space:

$$\begin{aligned} BioCheckRequired &\triangleq \left(\begin{array}{l} status = gotUserToken \wedge userTokenPresence = present \\ \wedge UserTokenOK \wedge (\neg UserTokenWithOKAuthCert) \end{array} \right) \\ &\longrightarrow status := waitingFinger; currentDisplay := insertFinger \\ ReadFingerOK &\triangleq \left(\begin{array}{l} status = waitingFinger \wedge fingerPresence = present \\ \wedge userTokenPresence = present \end{array} \right) \\ &\longrightarrow status := gotFinger; currentDisplay := wait \\ UnlockDoorOK &\triangleq \left(\begin{array}{l} status = waitingRemoveTokenSuccess \\ \wedge userTokenPresence = absent \end{array} \right) \\ &\longrightarrow UnlockDoor; status := quiescent; \\ &\quad currentDisplay := doorUnlocked \end{aligned}$$

Each operation is guarded by execution conditions and consists of several assignments. *BioCheckRequired* requires that the current state is *gotUserToken*, the user token is *present*, and sufficient for entry (*UserTokenOK*), but there is no authorisation certificate ($\neg UserTokenWithOKAuthCert$). The latter two predicates essentially require that (1) the three certificates can be verified against the public key store, and (2) additionally there is a valid authorisation certificate present. Their definitions can be found elsewhere [5]. *BioCheckRequired* updates the state

to *waitingFinger* and the display with an instruction to provide a fingerprint. *UnlockDoorOK* requires that the current state is *waitingRemoveTokenSuccess*, and the token has been removed. It unlocks the door, using the elided operation *UnlockDoor*, returns the status to *quiescent*, and updates the display.

These operations act only on the TIS state space. During their execution monitored variables can also change, to reflect real-world updates. Mostly these changes are arbitrary, with the exception that time must increase monotonically. We therefore promote the operations to *SystemState* with the following schema.

$$UEC(Op) \triangleq tis:[Op] ; rw:[mon:now \leq mon:now' \wedge ctrl' = ctrl]$$

In Z, this functionality is provided by schema *UserEntryContext* [5], from which we derive the name *UEC*. It promotes *Op* to act on *tis*, and composes this with a relation that specifies changes to the real-world variables (*rw*). We specify this as a UTP relational predicate. The behaviour of all monitored variables other than *now* is arbitrary, and all controlled variables are unchanged. Then, we promote each operation, for example *TISReadTokenOK* \triangleq *UEC(ReadTokenOK)*. The overall behaviour of the entry operations is given below:

$$TISUserEntryOp \triangleq \left(\begin{array}{l} TISReadUserToken \sqcap TISValidateUserToken \\ \sqcap TISReadFinger \sqcap TISValidateFinger \\ \sqcap TISUnlockDoor \sqcap TISCompleteFailedAccess \sqcap \dots \end{array} \right)$$

In each iteration of the state machine, we non-deterministically select an enabled operation and execute it. We also update the controlled variables, which is done by composition with the following update operation.

$$\begin{aligned} TISUpdate \triangleq & rw:[mon:now \leq mon:now'] ; rw:ctrl:latch := tis:currentLatch ; \\ & rw:ctrl:display := tis:currentDisplay \end{aligned}$$

We also formalise the TIS state invariants necessary to prove SFR1:

$$\begin{aligned} Inv_1 &\triangleq \text{status} \in \left\{ \begin{array}{l} \text{gotFinger}, \text{waitingFinger}, \text{waitingUpdateToken} \\ \text{waitingEntry}, \text{waitingUpdateTokenSuccess} \end{array} \right\} \\ &\Rightarrow (\text{UserTokenWithOKAuthCert} \vee \text{UserTokenOK}) \\ Inv_2 &\triangleq \text{status} \in \{\text{waitingEntry}, \text{waitingUpdateTokenSuccess}\} \\ &\Rightarrow (\text{UserTokenWithOKAuthCert} \vee \text{FingerOK}) \\ TIS-inv &\triangleq Inv_1 \wedge Inv_2 \wedge \dots \end{aligned}$$

*Inv*₁ states that whenever the TIS is in a state beyond *gotUserToken*, then either a valid authorisation certificate is present, or else the user token is valid. *Inv*₂ states that whenever in state *waitingEntry* or *waitingUpdateTokenSuccess*, then either an authorisation certificate or a valid finger print is present. We elide the additional invariants that deal with the alarm and audit data [5].

Next, we show that each operation preserves *TIS-inv* using Hoare logic.

Theorem 5.1. $\{TIS-inv\} TISUserEntryOp \{TIS-inv\}$

Proof. Automatic, by application of Isabelle/UTP proof tactic `hoare-auto`.

This theorem shows that the state machine never violates the invariants, and we can assume they hold to satisfy any requirements. We use this to formalise and prove SFR1, which requires that we determine all states under which the latch will be unlocked. We can determine these states by application of the weakest precondition calculus [10]. Specifically, we characterise the weakest precondition under which execution of *TISUserEntryOp* followed by *TISUpdate* leads to a state satisfying *rw:ctrl:latch = unlocked*. We formalise this in the theorem below.

Theorem 5.2 (FSFR1 is satisfied).

$$\begin{aligned} & \left(\begin{array}{l} TIS\text{-inv} \wedge tis:\text{currentLatch} = \text{locked} \\ \wedge (TISEntryOp ; TISUpdate) \text{ wp } (rw:\text{ctrl}:latch = \text{unlocked}) \end{array} \right) \\ & \Rightarrow ((UserTokenOK \wedge FingerOK) \vee UserTokenWithOKAuthCert) \end{aligned}$$

Proof. Automatic, by application of weakest precondition and relational calculi.

We calculate the weakest precondition, and conjoin this with *TIS-Inv*, which always holds, and the predicate *tis:currentLatch = locked* to capture behaviours when the latch was initially locked. We show that this composite precondition implies that either a valid user token and fingerprint were present, or else a valid authorisation certificate. We have therefore now verified a formalisation of SFR1. In the next section we place this in the context of an assurance argument.

6 Mechanising the Tokeener Assurance Case

In the following, we mechanize an assurance argument with the claim that TIS satisfies **SFR1**. The assurance case fragment is shown in [Figure 7](#), which is inspired by the formalisation pattern [9]. The latter shows how results from a formal method can be employed in an assurance case. This is contingent on the validation of both the formal model and the formal requirement. Consequently, the formalisation pattern breaks down a requirement into 3 claims stating that (1) the formal model is validated, (2) the formal requirement correctly characterises the informal requirement, and (3) the formal requirement is satisfied by the formal model. The former two claims will usually have an informal process argument,

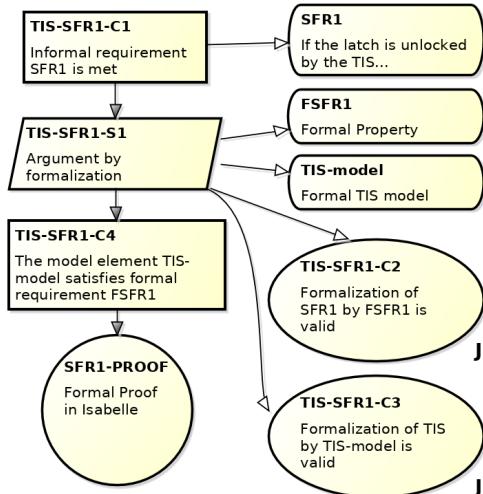


Figure 7: TIS Claim Formalization

```

EXPRESSION SFR1 CONTENT <If the latch is unlocked by the TIS, then the
TIS must be in possession of either a User Token...>
CLAIM TIS_SFR1_C1 CONTENT <Informal Requirement @{Expression SFR1} is met>
CLAIM TIS_SFR1_C2 CONTENT <Formalization of the informal requirement
@{Expression SFR1} by the formal requirement @{thm FSFR1_def} is valid>
CLAIM TIS_SFR1_C3 CONTENT <Formalization of system element @{Resource TIS}
by model @{const TIS_model} is valid>
CLAIM TIS_SFR1_C4 CONTENT <The model element @{const TIS_model}
satisfies formal requirement @{thm FSFR1_def}>
CLAIM TIS_SFR1_C5
    CONTENT <Formal requirement @{thm FSFR1_def} is established by proof>
ASSERTED_INFERENCE TIS_SFR1_S1
SOURCE <{@{docitem ''TIS_SFR1_C2''}, {@docitem ''TIS_SFR1_C3''},
{@docitem ''TIS_SFR1_C4''}}> TARGET <{@{docitem ''TIS_SFR1_C1''}}>
ARGUMENT_REASONING <Argument by formalization>
ASSERTED_INFERENCE TIS_SFR1_S2
SOURCE <{@{docitem ''TIS_SFR1_C5''}}> TARGET <{@{docitem ''TIS_SFR1_C4''}}>
ASSERTED_CONTEXT TIS_SFR1_AC1
SOURCE <{@{docitem ''ISABELLE_2018_REF''}}> TARGET <{@{docitem ''TIS_SFR1_S2''}}>
ASSERTED_EVIDENCE TSI_SFR1_AE1
SOURCE <{@{docitem ''TIS_SFR1_SPEC_THY_REF''}}> TARGET <{@{docitem ''TIS_SFR1_C5''}}>

```

Figure 8: TIS argument: Claims and their relations in Isabelle/SACM

The argument in Figure 7 justifies the link between the informal claim “TIS satisfies **SFR1**”, which is in natural language, and the formal theorem **FSRF1** from §5, which is expressed in HOL. The top-level claim, **TIS-SFR1-C1**, states that **SFR1** is satisfied, which it references as a contextual element. This claim is decomposed by the use of the formalization strategy, **TIS-SFR1-S1**, which has the formal property (**FSRF1**) and TIS model from §5 as context. The satisfaction of the formal claim is expressed by **TIS-SFR1-C4**, and evidenced by **SFR1-PROOF**, which is the formal proof. The validation claims are encoded as justifications **TIS-SFR1-C3** and **TIS-SFR1-C4**, which are not elaborated.

Figure 8 contains a mechanised version of the same argument in Isabelle/SACM. The structure is slightly different from the GSN diagram since justifications are particular kind of claims in SACM. The five claims are specified using the **CLAIM** command, with a name and content associated. The text in these claims integrate hyper-linked semi-formal and formal content; for example **TIS-SFR1-C1** uses the antiquotation **Expression** to insert a formal link to the defined expression for **SFR1**. Similarly, **TIS-SFR1-C3** contains a reference to the resource artifact **TIS**, which is a reference to the Tokeneer specification, and also an Isabelle constant **TIS-model** which contains the formal TIS model.

The inferences between these claims are specified by several instances of the **ASSERTED_INFERENCE** command, each of which links several premise claims to one or more conclusions. **TIS-SFR1-S1** shows that satisfaction of the informal requirement depends on the formal requirement, and the two validation claims.

Figure 7 does not elaborate further on the evidential artifacts required, for the verification, as GSN does not support this. This is functionality which SACM supports through the artifact meta-model, which allows us to record activities,

```

RESOURCE TIS PROPERTIES <{}>
LOCATION <@{url "https://www.adacore.com/tokeneer/">
CONTENT <Website with the specification documents for the TIS. >
ARTIFACT TIS_FSFR1_PR00F_THY VERSION <> DATE <> PROPERTIES <{}>
CONTENT <Proof of @{thm FSFR1_def}>
ACTIVITY TIS_FSFR1_PR00F_THM StartTime <02/04/2019>
EndTime <02/04/2019> PROPERTIES <{}>
CONTENT <Discharging the proof obligation @{thm FSFR1_def} by tactics
such as @{method auto} to the theorem @{thm SFRs_Correct}.>
RESOURCE Isabelle2018 PROPERTIES <{}>
LOCATION <@{url "https://isabelle.in.tum.de/">
CONTENT <Website of the Isabelle Interactive Theorem Prover>
PARTICIPANT Simon_Foster PROPERTIES <{}> CONTENT <Proof done by Simon Foster>
ARTIFACT_ASSET_RELATION TIS_SFR1_PR00F_ACTIVITY_REL
SOURCE<{@{docitem ''TIS_FSFR1_PR00F_THM''}, {@{docitem ''Isabelle2018''}},
{@{docitem ''Simon_Foster''}}>
TARGET<{@{docitem ''TIS_FSFR1_PR00F_THY''}}>
ARTIFACT_REFERENCE TIS_SFR1_SPEC_THY_REF
REFERENCED_ARTIFACTS <{@{docitem ''TIS_FSFR1_PR00F_THY''}}>

```

Figure 9: TIS argument: Artifacts and their relations in Isabelle/SACM

participants, resources, and other assurance artifacts. Figure 9 supplements the argument in Figure 8 with the various evidential artifacts, and the relationships between them. For example, the evidence supporting the claim TIS_SFR1_C4 is a reference to the artifact TIS_FSFR1_SPEC_THY which is the Isabelle theory containing the proof of the theorem stated by TIS_SFR1_C4. TIS_FSFR1_SPEC_THY refers via the artifact relationship TIS_SFR1_PR00F_ACTIVITY_REL to the context of the proof which is the artifact Isabelle_IT and to the author of the proof which is Simon_Foster. This illustrates how Isabelle/SACM allows us to combine informal artifacts and activities with the formal results they produce.

7 Related Work

Woodcock et al. [31] highlight defects of the Tokeneer SPARK implementation, indicate undischarged verification conditions, and perform robustness tests generated by the Alloy SAT solver [22] model. Using De Bono’s lateral thinking, these test cases go beyond the anticipated operational envelope and stimulate anomalous behaviours of the Tokeneer implementation. In shortening the feedback cycle for verification and test engineers, interactive theorem proving can help using Woodcock’s approach more intensively.

Rivera et al. [25] present an Event-B model of the TIS (the enrolment operations of the admin are presented), verify this model, generate Java code from it using the Rodin tool, and test this code by JUnit tests manually derived from the specification. The tests validate the model in addition to the Event-B invariants derived from the same specification. The tests aim to detect errors in Event-B

model caused by misunderstandings of the specification. Using Rodin, the authors verify the security properties (Section 3) using Hoare triples.

We believe that our work is the first to put formal verification effort into the wider context of structured assurance argumentation, in our case, a machine-checked security case using Isabelle/SACM.

Several works bring formality to assurance cases [7–9, 11, 27]. AdvoCATE is a tool for the construction of GSN-based safety cases [8, 9]. It uses a formal foundation called argument structures, which prescribe well-formedness checks for the graph structure, and allow instantiation of assurance case patterns. Our work likewise ensures well-formedness, and additionally allows the embedding of formal content. Denney’s formalisation pattern [9] is an inspiration for our work.

Rushby shows how assurance arguments can be embedded into formal logic to overcome logical fallacies [27]. Our framework similarly allows reasoning using formal logic, but additionally allows us to combine formal and informal artifacts. We were also inspired by the work on evidential toolbus [7], which allows the combination of evidence from several formal and semi-formal analysis tools. Isabelle similarly allows integration of a variety of formal analysis tools [29].

8 Conclusion

We have presented Isabelle/SACM, a framework for mechanised assurance cases. We showed how SACM is embedded into Isabelle as an ontology, and provided an interactive assurance language that generates valid instances. We applied it to the production of part of the Tokeneer security case, including verification of one of the security functional requirements, and embedded these results into a mechanised assurance argument. Of a particular note, Isabelle/SACM enforce the usage of the formal ontological links which represent the relationships between the assurance arguments and their claims, a feature we inherit from Isabelle/DOF. Isabelle/SACM also combines features from Isabelle/HOL, Isabelle/DOF and SACM which results in a framework that allows the integration of formal methods and argument-based safety assurance cases.

In future work, we will consider the integration of assurance case pattern execution [9] into our framework, which facilitate their production. We will also complete the mechanisation of the TIS security case, including verification of the other five SFRs. In parallel with this, we are developing our verification framework, Isabelle/UTP [14] to support a variety of notations used in software engineering. We recently demonstrated formal verification facilities for a statechart-like notation [12], and are also working towards tools to support hybrid dynamical languages [13] like Modelica and Simulink. Our overarching goal is a comprehensive assurance framework supported by a variety of integrated formal methods in order to approach complex certification tasks for cyber-physical systems and autonomous robots.

References

1. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the tokeneer enclave protection software. In: Proceedings of IEEE International Symposium on Secure Software Engineering (2006), https://www.adacore.com/uploads/technical-papers/issse2006tokeneer_altran.pdf
2. Brucker, A.D., Ait-Sadoune, I., Crisafulli, P., Wolff, B.: Using the isabelle ontology framework - linking the formal with the informal. In: CICM. Lecture Notes in Computer Science, vol. 11006, pp. 23–38. Springer (2018)
3. Brucker, A.D., Wolff, B.: On theorem prover-based testing. Formal Asp. Comput. 25(5), 683–721 (2013), <https://doi.org/10.1007/s00165-012-0222-y>
4. Common Criteria Consortium: Common criteria for information technology security evaluation – part 1: Introduction and general model. Tech. Rep. CCMB-2017-04-001, Common Criteria Consortium (2017), <https://www.commoncriteriaportal.org/index.cfm>
5. Cooper, D., et al.: Tokeneer ID Station: Formal Specification. Tech. rep., Praxis High Integrity Systems (August 2008), <https://www.adacore.com/tokeneer>
6. Cooper, D., et al.: Tokeneer ID Station: Security Properties. Tech. rep., Praxis High Integrity Systems (August 2008), <https://www.adacore.com/tokeneer>
7. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the evidential tool bus. In: VMCAI. LNCS, vol. 7737. Springer (2013)
8. Denney, E., Pai, G.: A formal basis for safety case patterns. In: SAFECOMP. LNCS, vol. 8153. Springer (2013)
9. Denney, E., Pai, G.: Tool support for assurance case development. Automated Software Engineering 25, 435–499 (2018)
10. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
11. Diskin, Z., Maibaum, T., Wassyng, A., Wynn-Williams, S., Lawford, M.: Assurance via model transformations and their hierarchical refinement. In: MODELS. IEEE (2018)
12. Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A., Woodcock, J.: Automating verification of state machines with reactive designs and Isabelle/UTP. In: Proc. 15th. Intl. Conf. on Formal Aspects of Component Software. LNCS, vol. 11222. Springer (October 2018)
13. Foster, S., Thiele, B., Cavalcanti, A., Woodcock, J.: Towards a UTP semantics for Modelica. In: UTP. pp. 44–64. LNCS 10134, Springer (2016)
14. Foster, S., Zeyda, F., Nemouchi, Y., Ribeiro, P., Wolff, B.: Isabelle/UTP: Mechanised Theory Engineering for Unifying Theories of Programming. Archive of Formal Proofs (2019), <https://www.isa-afp.org/entries/UTP.html>
15. Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: ICTAC. LNCS 9965, Springer (2016)
16. Gleirscher, M., Foster, S., Woodcock, J.: New opportunities for integrated formal methods. Unpublished working paper, Department of Computer Science, University of York (2018), <https://arxiv.org/abs/1812.10103>
17. Greenwell, W., Knight, J., Holloway, C.M., Pease, J.: A taxonomy of fallacies in system safety arguments. In: Proc. 24th Intl. System Safety Conference (July 2006)
18. Habli, I., Ibarra, I., Rivett, R., Kelly, T.: Model-based assurance for justifying automotive functional safety. In: Proc. SAE World Congress (April 2010)
19. Habli, I., Kelly, T.: Balancing the formal and informal in safety case arguments. In: VeriSure: Verification and Assurance Workshop, colocated with Computer-Aided Verification (CAV) (July 2014)

20. Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T.: Weaving and assurance case from design: A model-based approach. In: Proc. 16th Intl. Symp. on High Assurance Systems Engineering. IEEE (2015)
21. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
22. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11(2), 256–290 (July 2000)
23. Kelly, T.: Arguing Safety – A Systematic Approach to Safety Case Management. Ph.D. thesis, University of York (1998)
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
25. Rivera, V., Bhattacharya, S., Cataño, N.: Undertaking the tokeneer challenge in Event-B. In: Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering - FormaliSE '16. ACM Press (2016)
26. Rushby, J.: Logic and epistemology in safety cases. In: SAFECOMP. vol. LNCS 8153. Springer (2013)
27. Rushby, J.: Mechanized support for assurance case argumentation. In: New Frontiers in Artificial Intelligence. LNCS, vol. 8417. Springer (2014)
28. Selviandro, N., Kelly, T., Hawkins, R.: The visual inheritance structure to support the design of visual notations. In: MODELS Workshop. CEUR Workshop Proceedings, vol. 2245 (October 2018)
29. Wenzel, M., Wolff, B.: Building formal method tools in the isabelle/isar framework. In: TPHOLs. LNCS, vol. 4732. Springer (2007)
30. Wenzel, M.M., München, T.U.: Isabelle/isar - a versatile environment for human-readable formal proof documents (2002)
31. Woodcock, J., Aydal, E.G., Chapman, R.: The tokeneer experiments. In: Reflections on the Work of C.A.R. Hoare, pp. 405–430. Springer London (2010)