https://doi.org/10.1007/s00165-021-00537-4
The Author(s) © 2021
Formal Aspects of Computing

Formal Aspects
of Computing

Integration of formal proof into unified assurance cases with Isabelle/SACM

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

Department of Computer Science, University of York, United Kingdom

Abstract. Assurance cases are often required to certify critical systems. The use of formal methods in assurance can improve automation, increase confidence, and overcome errant reasoning. However, assurance cases can never be fully formalised, as the use of formal methods is contingent on models that are validated by informal processes. Consequently, assurance techniques should support both formal and informal artifacts, with explicated inferential links between them. In this paper, we contribute a formal machine-checked interactive language, called Isabelle/SACM, supporting the computer-assisted construction of assurance cases compliant with the OMG Structured Assurance Case Meta-Model. The use of Isabelle/SACM guarantees well-formedness, consistency, and traceability of assurance cases, and allows a tight integration of formal and informal evidence of various provenance. In particular, Isabelle brings a diverse range of automated verification techniques that can provide evidence. To validate our approach, we present a substantial case study based on the Tokeneer secure entry system benchmark. We embed its functional specification into Isabelle, verify its security requirements, and form a modular security case in Isabelle/SACM that combines the heterogeneous artifacts. We thus show that Isabelle is a suitable platform for critical systems assurance.

Keywords: Assurance cases; Integrated formal methods; Proof assistants; Safety cases; Common criteria

1. Introduction

Assurance cases (ACs) are structured arguments, supported by evidence, intended to demonstrate that a system meets its requirements, such as safety or security, when applied in a particular operational context [WKD^{+}19, Kel98]. They are recommended by several international standards, such as ISO 26262 for automotive applications. An AC consists of a hierarchical decomposition of claims, through appropriate argumentation strategies, into further claims, and eventually supporting evidence. Several AC notations exist, including the Goal Structuring Notation (GSN) [Kel98], Claims, Arguments, and Evidence (CAE) [BB98], and the Structured Assurance Case Metamodel (SACM) [Obj20, WKD^{+}19]. These notations support the management of complex ACs via their hierarchical decomposition and modular representation [GC17].

Correspondence to: Simon Foster, simon.foster@york.ac.uk.

Published online: 08 June 2021

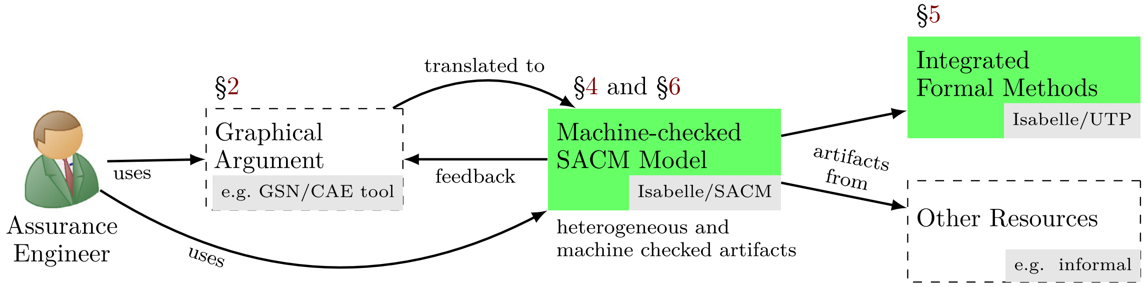


Fig. 1. Overview of our approach to integrative model-based assurance cases

AC creation can be supported by model-based design, which utilises a variety of models for argumentation, system architecture and behaviour, and a variety of other assurance artifacts [HHK+15, DMW+18, WKD+19]. The use of models can improve the systems assurance process by allowing the consequences of design decisions to be explored early on, and the justifications recorded in the AC argument. In particular, the SACM standard supports structured model-based assurance cases [WKD+19] with argumentation and evidence traceability, and several advanced features like patterns [DP13] and modularity [DP15]. However, structured ACs can suffer from ambiguity, logical fallacies, and inadequate evidence, which can compromise confidence in assurance claims [GKHP06, DP15, DMW+18]. Moreover, ACs can be difficult to maintain and update in the presence of evolving requirements [GFN19]. There is therefore a need for tools that can automate AC creation and evolution, whilst maintaining the integrity of the arguments and evidence.

A proposed approach to improve confidence in ACs is their formalisation in a machine-checked logic to enable verification of consistency and well-formedness [Rus13, Rus14]. Moreover, as confirmed by the avionics standard DO-178C and its supplement DO-333, the evidence gathering process can also benefit from the rigour of Formal Methods (FMs) [J+11]. It is clear that there are significant benefits to the use of formality in systems assurance [DMW+18]. At the same time, we observe that (1) ACs are intended primarily to convince humans (particularly non-experts) of their claims; and (2) formal models must be validated using informal processes [HK14] (particularly by domain experts). Consequently, ACs usually combine informal and formal content, and so tools must support this combination. Moreover, there is a need to integrate several FMs [Pai97], potentially with differing computational paradigms and levels of abstraction [HH98]. Therefore we maintain traceability across these heterogeneous artifacts using FM integration [GFW19], which goes beyond shallow hyperlinking of engineering data.

Vision Our vision, illustrated in Fig. 1, is a unified framework for machine-checked ACs with heterogeneous artifacts and integrated FMs. We envisage an assurance backend for a variety of graphical assurance tools [DP18, WKD+19] that utilise SACM [Obj20] as a unified interchange format, and an array of FM tools provided by the Isabelle proof assistant [NPW02, WW07] and our unified verification platform, Isabelle/UTP [FBC+20, FZN+19]. Our framework aims to improve existing processes by harnessing formal verification to produce mathematically grounded ACs with guarantees of consistency and adequacy of the evidence, whilst supporting human readable justifications accessible by non-experts. In the context of safety regulation in domains such as intelligent transportation, critical infrastructure, human-robot collaboration, or medical devices, our framework can aid AC evaluation through machine-checking with automated verification [BW19b, FNO+20].

Contributions A first step towards our vision is made by the contributions of this paper, which are: (1) Isabelle/SACM, an implementation of SACM in Isabelle; (2) an interactive DSL for machine-checked ACs based on Isabelle/SACM; (3) a novel mechanisation of the Tokeneer benchmark [BCJ+06] in Isabelle/UTP, which serves as a case study for our approach; (4) formal verification of the Tokeneer security requirements[1]; and (5) a modular assurance case capturing the lifecycle artifacts and claims that Tokeneer meets its security requirements. Our Tokeneer assurance case demonstrates how one can integrate formal artifacts, resulting from the work with Isabelle/UTP (4), and informal artifacts, such as the Tokeneer documentation.

Isabelle [NPW02] provides a sophisticated executable document model for presenting a graph of hyperlinked formal artifacts, such as definitions, theorems, and proofs [Wen19]. It provides automatic and

[1] Supporting materials, including Isabelle theories, can be found on our website.

incremental consistency checking, where updates to artifacts trigger rechecking. Such capabilities can support efficient maintenance and evolution of model-based ACs [WKD$^+$19]. Moreover, the document model allows both formal and informal content [Wen18], and provides access to an array of automated proof tools [WW07, Wen18]. Additionally, Brucker et al. [BACW18, BW19a, BW19b] have recently created Isabelle/DOF, a framework with a textual language for the embedding of ontologies into Isabelle. DOF allows the embedding of structured informal content into an Isabelle document, such as documentation and justification, which we harness to support SACM. These facilities make Isabelle an attractive platform both for assurance cases and integration of formal methods, for which our work serves as a proof-of-concept.

Isabelle/UTP [FBC$^+$20] employs Unifying Theories of Programming [HH98] (UTP) to provide formal verification facilities for a variety of languages, with paradigms as diverse as concurrency [FCC$^+$19], real-time [FCWZ18], and hybrid computation [FTCW16, MSF20]. Moreover, verification techniques such as Hoare logic, weakest precondition calculus, and refinement calculus are all available through a variety of proof tactics. This makes Isabelle/UTP an obvious choice for modelling and verification of Tokeneer, and more generally as a platform for integrated FMs based on unifying formal semantics.

We believe our novel mechanisation of the Tokeneer functional specification in Isabelle/UTP is one of the most complete to date, with respect to the original benchmark [C$^+$08a]. The model includes 60 state variables, 38 top-level operations for user entry, admin, and enrolment procedures, 30 invariants, and several hundred discharged invariant proof obligations. Where possible, we are faithful to the benchmark, using the same names and structure for the system model. With our mechanisation we are able to formally verify three security requirements that could only be argued semi-formally in the original [C$^+$08b]. Our work therefore demonstrates how automated proof tools have advanced over the past fifteen years. We also highlight a few invariants missing from the original formal specification, without which we could not verify Tokeneer.

This paper is an extension of a previous conference paper [FNGK19]. We develop a more elaborate modular assurance case for Tokeneer (§ 3 and § 6), further develop our IAL (§ 4), and formalise the Admin operations in the Tokeneer formal model and verify two additional security properties (§ 5). We also provide further implementation details and examples throughout, and in particular describe our strategy for converting the Tokeneer Z schemas into Isabelle/UTP.

Our paper is organised as follows. In § 2, we outline preliminaries: SACM, Isabelle, DOF, and Isabelle/UTP. In § 3 we describe the Tokeneer system. In § 4, we begin our contributions by describing Isabelle/SACM, which consists of the embedding of SACM into DOF (§ 4.1), and IAL (§ 4.2). In § 5, we model and verify Tokeneer in Isabelle/UTP. In § 6, we describe the mechanisation of the Tokeneer AC in the ACME graphical assurance case tool and Isabelle/SACM. In § 7, we indicate relationships to previous research. After reflecting on our approach in § 8, we conclude in § 9.

2. Preliminaries

In this section, we provide background material on ACs, the SACM standard, the Isabelle components, and Isabelle/UTP. These are all required to follow our investigations in § 4, § 5, and § 6.

2.1. Assurance cases and SACM

Assurance cases are often presented using a graphical notation like GSN [Kel98], as shown in Fig. 2. In GSN, claims (called "goals") are rectangles, which are linked with "supported-by" arrows, strategies are parallelograms, and the circles are evidence ("solutions"). The other shapes denote various types of context, which are linked to by the "in-context-of" arrows. An argument in GSN proceeds from the most abstract claim down, through argumentation strategies and further subclaims, until the claims can be directly supported by evidence. GSN also has a modular extension, where arguments can be encapsulated in modules. Particular elements in modules can be marked as public, meaning that they are visible in the module interface, and can be cited from other modules using elements such as "away goals" and "away context". Elements that are not marked as public are hidden, which allows arguments to be abstracted and thus favours compositionality.

SACM is an OMG standard meta-model for ACs [WKD$^+$19]. It unifies, extends, and refines several predecessor notations, including GSN [Kel98] and CAE [BB98] (Claims, Arguments, and Evidence), and is intended as a definitive reference model. SACM models three crucial concepts: arguments, artifacts, and terminology. An argument is a set of claims, evidence citations, and inferential links between them.

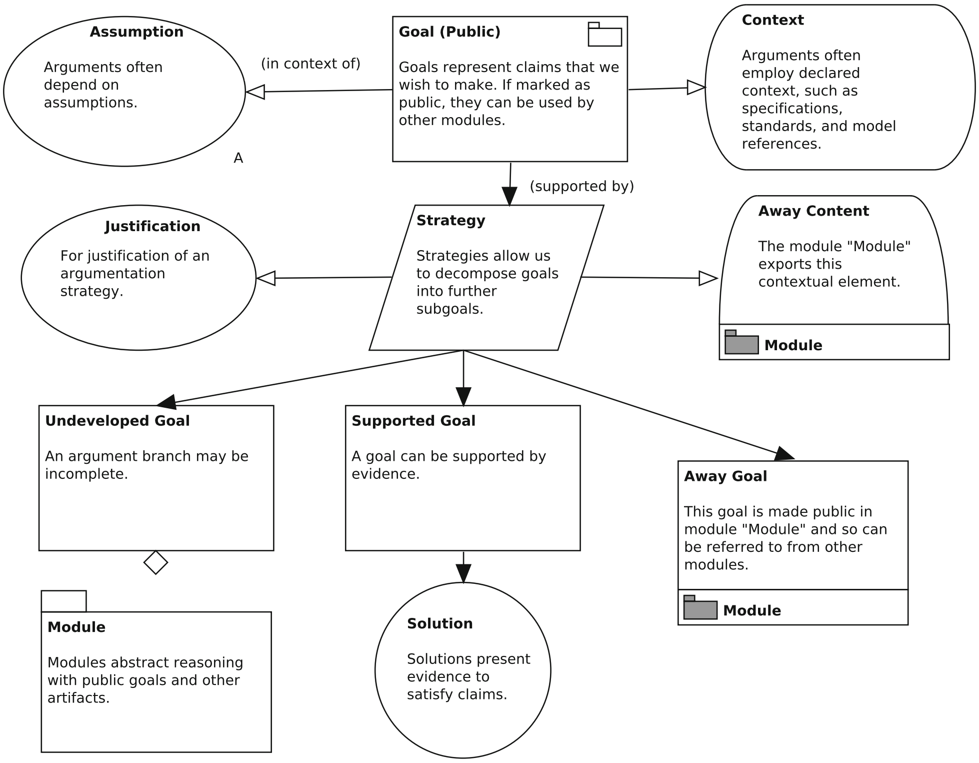

Assumption
Arguments often depend on assumptions.
A
(in context of)
Goal (Public)
Goals represent claims that we wish to make. If marked as public, they can be used by other modules.
Context
Arguments often employ declared context, such as specifications, standards, and model references.
(supported by)
Justification
For justification of an argumentation strategy.
Strategy
Strategies allow us to decompose goals into further subgoals.
Away Content
The module "Module" exports this contextual element.
Module
Undeveloped Goal
An argument branch may be incomplete.
Supported Goal
A goal can be supported by evidence.
Away Goal
This goal is made public in module "Module" and so can be referred to from other modules.
Module
Module
Modules abstract reasoning with public goals and other artifacts.
Solution
Solutions present evidence to satisfy claims.

Fig. 2. The goal structuring notation (GSN) with modular extensions

Artifacts represent evidence, such as system models, techniques, results, activities, participants, and traceability links. Terminology fixes formal terms for use in claims. Normally, claims are in natural language, but in SACM they can also contain structured expressions, which allows integration of formal languages. Arguments, artifacts, and terminology can all be grouped into a number of packages, which generalise GSN modules.

The argumentation meta-model of SACM is shown in Fig. 3. 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 Fig. 2.

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 explicit reasoning from premises to conclusion(s); or AssertedArtifactContext which documents a dependency between the claims of two artifacts.

Both Claims and AssertedRelationships inherit from Assertion, because 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 counter evidence is presented.

For development of graphical assurance cases, we use an Eclipse-based tool called *Assurance Case Management Environment* (ACME), from which we captured Fig. 2. ACME supports the creation and management of assurance cases using notations such as GSN (and in the future CAE), the abstract syntax of which is an extension of SACM, as explained in detail in [WKD$^+$19]. ACME integrates a number of model management tools and frameworks, including Eclipse Epsilon [KPP06], Eclipse Hawk [BK13], and Xtext [Bet16], towards the management of fully model-based assurance cases. Using SACM's full potential and with the help of model management frameworks, ACME currently supports 1) fine-grained traceability from an assurance case to its referenced engineering models (defined in mainstream modelling technologies such as, e.g. UML) to the level of model elements; 2) traceability to formal notations in Isabelle; 3) automated means to validate/verify traced engineering artifacts; 4) use and execution of constrained natural language for model validation; 5) automated change impact analysis for assurance cases and their engineering artifacts.

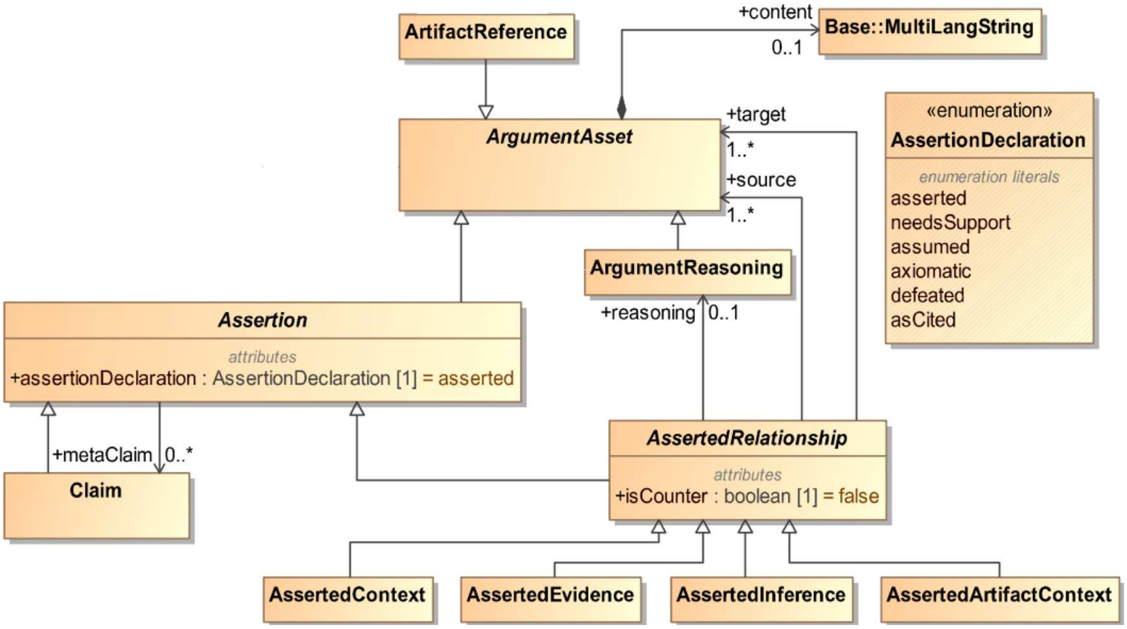


Fig. 3. Fragment of the SACM argumentation meta-model [Obj20]

2.2. Isabelle, Isar, and DOF

Isabelle/HOL is an interactive theorem prover for higher order logic (HOL) [NPW02], based on the generic framework Isar [WW07]. The former provides a functional specification language, and an array of automated proof tools [BBN11]. The latter has an interactive, extensible, and executable document model [Wen18], which describes Isabelle theories. Plugins, such as Isabelle/HOL, DOF, Isabelle/UTP, and Isabelle/SACM have document models that contain conservative extensions to Isar.

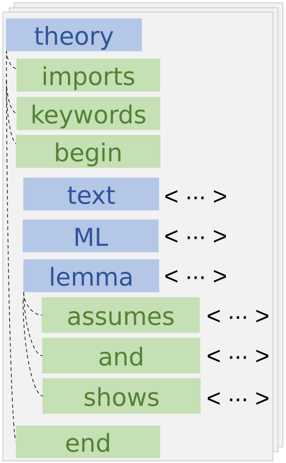


Fig. 4: The Isabelle/Isar Document Model

Figure 4 illustrates the document model. The first section for *context definition* describes *imports* of existing theories, and *keywords* which extend the concrete syntax. The second section is the body enclosed between *begin*-*end*, which is a sequence of commands. The concrete syntax of commands consists of (1) a pre-declared keyword (in blue), such as the command `ML`, (2) a "semantics area" enclosed between `‹...›`, and (3) optional subkeywords (in green). Commands generate document elements. For example, the command `lemma` creates a new theorem within the underlying theory context. When a document is edited by removal, addition, or alteration of elements, it is immediately executed and checked by Isabelle, with feedback provided to the frontend. This includes consistency checks for the context and well-formedness checks for the commands. Isabelle is therefore ideal for ACs, which have to be maintainable, well-formed, and consistent. In § 4.2 we extend this document model with commands that define our assurance language.

Moreover, informal artifacts in Isabelle theories can be combined with formal artifacts using the command `text ‹...›`. It is a processor for markup strings containing a mixture of informal artifacts and hyperlinks to formal artifacts through *antiquotations* of the form `@{aqname ...}`. For example, `text ‹The reflexivity theorem @{thm HOL.refl}›` mixes natural language with a hyperlink to the theorem `HOL.refl` through the antiquotation `@{thm HOL.refl}`. This is important since antiquotations are also checked by Isabelle as follows: (1) whether the referenced artifact exists within the underlying theory context; (2) whether the type of the referenced artifact matches the antiquotation's type.

A major foundation for our work is Isabelle/DOF [BACW18, BW19a, BW19b], an ontology framework for Isabelle. DOF permits the description of ontologies using Isabelle Ontology Specification Language (IOSL), a language to model *document classes*, which extends the document model with new structures. We use the command `doc_class` from IOSL to add new document classes for each of the SACM classes. Instances of DOF classes are not embedded into the HOL logic as datatypes, but sit at the meta-logical level in the document model. This means they can refer to other objects like theorems and definitions, can themselves be referenced using antiquotations, and carry an enriched version of the corresponding Isabelle markup string. One of DOF's targets is formal development of certification documents, making use of Isabelle's proof facilities [BW19b]. In this work, we advance this goal with our SACM-based assurance case framework.

2.3. Isabelle/UTP

Isabelle/UTP [FBC+20] is a tool for developing formal semantics and verification tools based on Hoare and He's Unifying Theories of Programming [HH98]. Isabelle/UTP contains a number of theories for reasoning about programs built using different computational paradigms, such as concurrent and real-time programming. In this paper, we use the core relational programming model to verify the functional specification of Tokeneer.

Variable mutation in Isabelle/UTP is modelled algebraically using lenses [FGM+07]. A variable of type V in a state space S is denoted by a lens $x : V \Longrightarrow S$, with two functions $\textbf{\textsf{get}} : S \to V$ and $\textbf{\textsf{put}} : S \to V \to S$, that respectively query and update the value of the variable in a given state $s : S$. This allows us to treat variables as semantic objects, rather than syntactic objects. We can check whether two lenses, x and y, refer to disjoint regions of the state space using independence $x \bowtie y$. We can also check whether an expression e depends on a particular variable x using unrestriction, which is written $x \sharp e$, and is a semantic encoding of variable freshness. For example, if $x \bowtie y$ (they are different variables), then $x \sharp (y + 1)$, since the valuation of $y+1$ does not depend on the value of x. Using these predicates, and the UTP relational program model [HH98], we can express laws about assignments, such as commutativity:

$(x := e \mathbin{⨾} y := f) = (y := f \mathbin{⨾} x := e)$ provided that $x \bowtie y$, $y \sharp e$, $x \sharp f$

that is, two assignments commute provided that the variables are independent and the assigned expressions do not use the adjacent variables. As we have shown [FBC+20], lenses allow us to semantically characterise variable sets as well (for example $a = \{x, y, z\}$) and thus framing properties. As a dual to unrestriction, we also have the used-by predicate, $a \natural e$, which states that e uses only those variables mentioned in a.

Rather than using the Z notation [Spi89], we use a variant of Dijkstra's Guarded Command Language (GCL) [Dij75] encoded in Isabelle/UTP to specify the model's behaviour. Our GCL has the following syntax:

$$P ::= \textbf{\textsf{skip}} \mid \textbf{\textsf{abort}} \mid P \mathbin{⨾} P \mid \mathcal{E} \longrightarrow P \mid P \sqcap P \mid \mathcal{V} := \mathcal{E} \mid \mathcal{V}{:}[P]$$

Here, P is a program, $\mathcal{E}$ is an expression, and $\mathcal{V}$ is a variable. The language provides sequential composition, guarded commands, non-deterministic choice[2], and assignment. We adopt a frame operator $a{:}[P]$, which states that P changes only variables in the namespace a [FZW16, FZN+19]. The namespace is modelled by a lens $a : S_1 \Longrightarrow S_2$, which shows how to embed the inner state-space S_1 into the outer state-space S_2. This enables modular reasoning about the Tokeneer Identification Station (TIS) internal and real-world states, which is a further novelty of our work. We give both a weakest precondition ($\textbf{\textsf{wp}}$) and weakest liberal precondition ($\textbf{\textsf{wlp}}$) semantics to our GCL. Technically, each operator is denoted as a relational predicate in UTP, and the following laws are theorems of these definitions [HH98, CW06].

Theorem 2.1 (UTP Weakest Preconditions).

$$\begin{array}{rlrll}
\textbf{\textsf{skip}}\ \textbf{\textsf{wp}}\ b = b & & (P \sqcap Q)\ \textbf{\textsf{wp}}\ b = P\ \textbf{\textsf{wp}}\ b \vee Q\ \textbf{\textsf{wp}}\ b & & \\
\textbf{\textsf{abort}}\ \textbf{\textsf{wp}}\ b = \mathit{false} & & x := e\ \textbf{\textsf{wp}}\ b = b[e/x] & & \\
(P \mathbin{⨾} Q)\ \textbf{\textsf{wp}}\ b = P\ \textbf{\textsf{wp}}\,(Q\ \textbf{\textsf{wp}}\ b) & & a{:}[P]\ \textbf{\textsf{wp}}\ b = (b \wedge (P\ \textbf{\textsf{wp}}\ \mathit{true})_{\uparrow a}) & & a \sharp b \\
(e \longrightarrow P)\ \textbf{\textsf{wp}}\ b = (e \wedge P\ \textbf{\textsf{wp}}\ b) & & a{:}[P]\ \textbf{\textsf{wp}}\ b = (P\ \textbf{\textsf{wp}}\ b_{\downarrow a})_{\uparrow a} & & a \natural b
\end{array}$$

With these equations, we can calculate the weakest precondition of any program composed of these operators.

[2] Technically, this is identical to the $\vee$ in Isabelle/UTP, the same as in Z.

The $\textsf{\textbf{wlp}}$ semantics is almost the same for each operator, except for the following equations:

$$\textsf{\textbf{abort}}\ \textsf{\textbf{wlp}}\ b = \mathit{true} \qquad (e \longrightarrow P)\ \textsf{\textbf{wlp}}\ b = (e \Rightarrow P\ \textsf{\textbf{wp}}\ b) \qquad (P \sqcap Q)\ \textsf{\textbf{wlp}}\ b = P\ \textsf{\textbf{wp}}\ b \wedge Q\ \textsf{\textbf{wp}}\ b$$

Most of the $\textsf{\textbf{wp}}$ and $\textsf{\textbf{wlp}}$ laws are standard [Dij75], the exception being the laws for the frame operator. These make use of state space coercions [FB20], $P_{\uparrow a}$ and $P_{\downarrow a}$, which respectively grow and shrink the state space of P using a. This, for example, means that the types of variables and quantifiers in P are type coerced. The first frame law has the proviso $a \sharp b$, meaning that the postcondition b does not depend on any variables in the frame a. Consequently, the weakest precondition is essentially b, but we also need to conjoin the domain of P. Since P operates on the inner state space, we need to grow its state space using coercions. The second frame law, conversely, has that b depends only on the variables in a. Consequently, the weakest precondition is derived directly from P, but with suitable state space coercions applied.

We can use $\textsf{\textbf{wlp}}$ calculus to verify Hoare triples, using the following well-known theorem [FBC+20]:

Theorem 2.2 $\{p\}\, Q\, \{r\} \Leftrightarrow (p \Rightarrow Q\ \textsf{\textbf{wlp}}\ r)$

In Isabelle/UTP we have developed a tactic, `hoare_wlp_auto`, that utilises this theorem, calculates the precondition using Theorem 2.1, and uses the UTP tactic `rel_auto` for relational calculus [FBC+20] to try and discharge the resulting verification condition. We also have a similar tactic, `hoare_auto` [FBC+20] that employs the classic Hoare logic deduction rules, rather than a $\textsf{\textbf{wlp}}$ calculation.

In addition to Hoare logic verification, some properties require that we can reason about the variables that a given operation can modify. In Isabelle/UTP, we can answer such framing questions using lenses [FBC+20]. We define a novel modification predicate, $P\ \textsf{\textbf{nmods}}\ a$, which states that the program P does not modify any of the variables captured by a. This can be formally specified by requiring that the relational characterisation of P is a fixed point of the function $M(X) \triangleq (a' = a \wedge X)$, so that every observation of P satisfies $a' = a$. We prove the following modification laws for this predicate.

Theorem 2.3 (Modification Predicate).

$$\frac{—}{\textsf{\textbf{skip}}\ \textsf{\textbf{nmods}}\ x} \qquad \frac{—}{\textsf{\textbf{abort}}\ \textsf{\textbf{nmods}}\ x} \qquad \frac{P\ \textsf{\textbf{nmods}}\ x \qquad Q\ \textsf{\textbf{nmods}}\ x}{(P \mathbin{⨾} Q)\ \textsf{\textbf{nmods}}\ x}$$

$$\frac{P\ \textsf{\textbf{nmods}}\ x \qquad Q\ \textsf{\textbf{nmods}}\ x}{(P \sqcap Q)\ \textsf{\textbf{nmods}}\ x} \qquad \frac{x \bowtie y}{(y := v)\ \textsf{\textbf{nmods}}\ x} \qquad \frac{P\ \textsf{\textbf{nmods}}\ x}{(b \longrightarrow P)\ \textsf{\textbf{nmods}}\ x}$$

$$\frac{x \notin a}{a{:}[P]\ \textsf{\textbf{nmods}}\ x} \qquad \frac{P\ \textsf{\textbf{nmods}}\ x}{a{:}[P]\ \textsf{\textbf{nmods}}\ a{:}x}$$

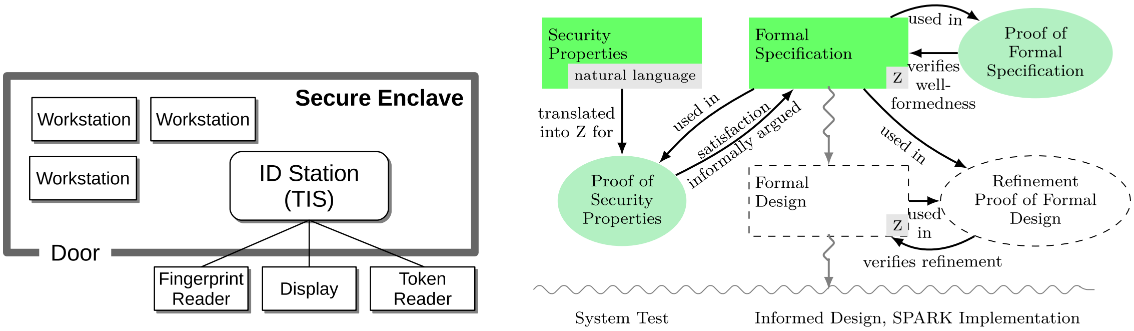
As expected, neither $\textsf{\textbf{skip}}$ nor $\textsf{\textbf{abort}}$ modify any variable x. Sequential composition, $P \mathbin{⨾} Q$, does not modify x provided that neither P nor Q does, and similarly for internal choice. Assignment to y does not modify x provided that x is independent of y ($x \bowtie y$), which effectively means that y is not part of x. A guarded command $b \longrightarrow P$ does not modify x provided that P also does not. For the frame operator, $a{:}[P]$, we identify two cases. If a variable x is not in a, then clearly it is not modified. Conversely, if x is within the a namespace, then it is necessary to check whether $P\ \textsf{\textbf{nmods}}\ x$. We use this predicate for formulating one of the Tokeneer security properties.

3. Case study: Tokeneer

To demonstrate our approach, we use the Tokeneer Identification Station (TIS)[3] illustrated in Fig. 5, a system that guards entry to a secure enclave. The pioneering work on the TIS assurance was carried out by Praxis High Integrity Systems and SPRE Inc. [BCJ+06]. Barnes et al. performed security analysis, definition of a security target, formal functional specification using Z, refinement to a formal design, implementation in SPARK, and verification of the security properties against the Z specification (Fig. 5b).

After independent assessment, Common Criteria (CC) Evaluation Assurance Level (EAL) 5 was achieved. Therefore, Tokeneer can be seen as a successful example of using FMs to assure a system against CC. Though now more than fifteen years old, it remains an important benchmark for FMs and other assurance techniques.

[3] Project website: https://www.adacore.com/tokeneer.



(a) Overview of the physical infrastructure.

(b) Fragment of the original assurance procedure.

Fig. 5. The Tokeneer ID station architecture and assurance procedure (adapted from [C+08c])

As indicated in Fig. 5a, the physical infrastructure consists of a door, fingerprint reader, display, and card (token) reader. The main function is to check the credentials on a presented token, read a fingerprint if necessary, and then either unlatch the door, or deny entry. Entry is permitted when the token holds at least three data items: (1) a user identity (ID) certificate, (2) a privilege certificate, with 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 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 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.

The security of the TIS is assured by demonstrating six Security Functional Requirements (SFRs) [C+08b]:

SFR1 If the latch is unlocked, then TIS must possess either a User token or an Admin token. The User token must either have a valid authorisation certificate, or valid ID, Privilege, and I&A Certificates, together with a template that allowed 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 the role "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.

SFR5 The presence of an audit record of one type will always be preceded by certain other audit records.

SFR6 The configuration data will be changed, or information written to the floppy, only if there is an Admin person logged on to the TIS.

Our objective is to (i) construct a machine-checked assurance case that argues that the TIS fulfils the security properties SFR1, SFR3, and SFR6, and (ii) integrate evidential artifacts from the mechanised model of the TIS behaviour in Isabelle/UTP into this assurance case. For these SFRs, our approach re-enacts the green parts in Fig. 5b. Particularly, we focus on verifying the functional formal specification against the security properties and on checking well-formedness of the functional specification. We omit SFR4 and SFR5 because they are not formalised by the Tokeneer project, and SFR2 for technical reasons which we explain in § 8.

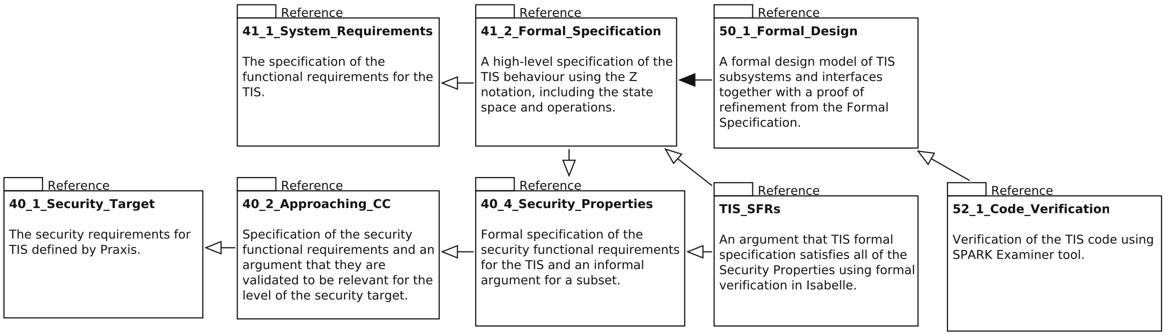

Reference
41_1_System_Requirements
The specification of the functional requirements for the TIS.
Reference
41_2_Formal_Specification
A high-level specification of the TIS behaviour using the Z notation, including the state space and operations.
Reference
50_1_Formal_Design
A formal design model of TIS subsystems and interfaces together with a proof of refinement from the Formal Specification.
Reference
40_1_Security_Target
The security requirements for TIS defined by Praxis.
Reference
40_2_Approaching_CC
Specification of the security functional requirements and an argument that they are validated to be relevant for the level of the security target.
Reference
40_4_Security_Properties
Formal specification of the security functional requirements for the TIS and an informal argument for a subset.
Reference
TIS_SFRs
An argument that TIS formal specification satisfies all of the Security Properties using formal verification in Isabelle.
Reference
52_1_Code_Verification
Verification of the TIS code using SPARK Examiner tool.

Fig. 6. A modular assurance case for the Tokeneer system development and assurance

We envisage the modular assurance case [Kel98, DP15] for Tokeneer illustrated in Fig. 6. Here, we have modelled the main documents produced during the development process as assurance case modules using modular GSN. The numbers correspond to the document codes given in the Tokeneer archive[4]. Each of the package symbols represents a collection of claims, arguments, and other lifecycle artifacts, for example 40_4_Security_Properties provides formalisation of some of the six SFRs. Certain artifacts are marked public, meaning they can be used by other modules, and some are private. The arrows between the modules indicate dependencies, for example the formal specification is developed both in the context of the system requirements and the security properties. In this paper, we focus on formalisation of 41_2_Functional_Specification, and the argument that the SFRs are satisfied in TIS_SFRs. The assurance arguments and artifacts will be embedded into Isabelle/SACM, which we develop in the next section.

4. Isabelle/SACM

In this section we encode SACM as a DOF ontology (§ 4.1), and use it to provide an interactive machine-checked AC language (§ 4.2). Our embedding implements ACs as meta-logical entities in Isabelle, that is, elements of the document model, rather than as formal elements embedded in the HOL logic, as this would prevent the expression of informal reasoning and explanation. Therefore, antiquotations to formal artifacts can be freely mixed with natural language and other informal artifacts.

4.1. Modelling: embedding SACM in Isabelle

We embed the SACM meta-model in Isabelle using IOSL, and we focus on modelling `ArgumentAsset`[5] and its child classes from Fig. 3, as these are the most relevant classes for the TIS assurance argument that we develop in § 6. The class `ArgumentAsset` has the following textual model:

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

Here, `doc_class` defines a new class, and automatically generates an antiquotation type, `@{ArgumentAsset <...>}`, which can be used to refer to entities of this type. `ArgumentationElement` is a class which the class `ArgumentAsset` inherits from, but is not discussed further. `content_assoc` models the content association in Fig. 3. To model `MultiLangString` in Isabelle/SACM, we use DOF's markup string. Thus, the usage of antiquotations is allowed for artifacts with the type `MultiLangString`.

`ArgumentAsset` has three subclasses: (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. Since DOF extends the Isabelle/HOL document model, we can use the latter's types, such as sets and enumerations (algebraic datatypes), in modelling SACM classes, as shown below:

[4] Tokeneer materials: https://www.adacore.com/tokeneer/download.

[5] We model all parts of SACM in DOF, but omit details for sake of brevity.

```
datatype assertionDeclarations_t =
  Asserted|Axiomatic|Defeated|Assumed|NeedsSupport|AsCited

doc_class Assertion = ArgumentAsset +
  assertionDeclaration::assertionDeclarations_t

doc_class Claim = Assertion +
  metaClaim::"Assertion set" <= "{}"

doc_class ArgumentReasoning  = ArgumentAsset +
  structure_assoc::"ArgumentPackage option"

doc_class ArtifactReference = ArgumentAsset +
  referencedArtifactElement_assoc::"ArtifactElement set"
```

Here, **datatype** defines an algebraic datatype, `assertionDeclarations_t` is an enumeration, `set` is the set type, and `option` is an optional type. Attribute `assertionDeclaration` has type `assertionDeclarations_t`, which specifies the status of instances of type `Assertion`. Examples of `Assertions` in SACM are claims, justifications, and both kinds of arrows in Fig. 2. A `Claim` is an assertion, extended with the `metaClaim` association. The attribute `structure_assoc`, in class `ArgumentReasoning`, is an association to the class `ArgumentPackage`, which is not discussed here. Finally, the attribute `referencedArtifactElement_assoc`, from class `ArtifactReference`, is an association to `ArtifactElements` from the `ArtifactPackage`, allowing instances of type `ArgumentAsset` to be supported by evidential artifacts.

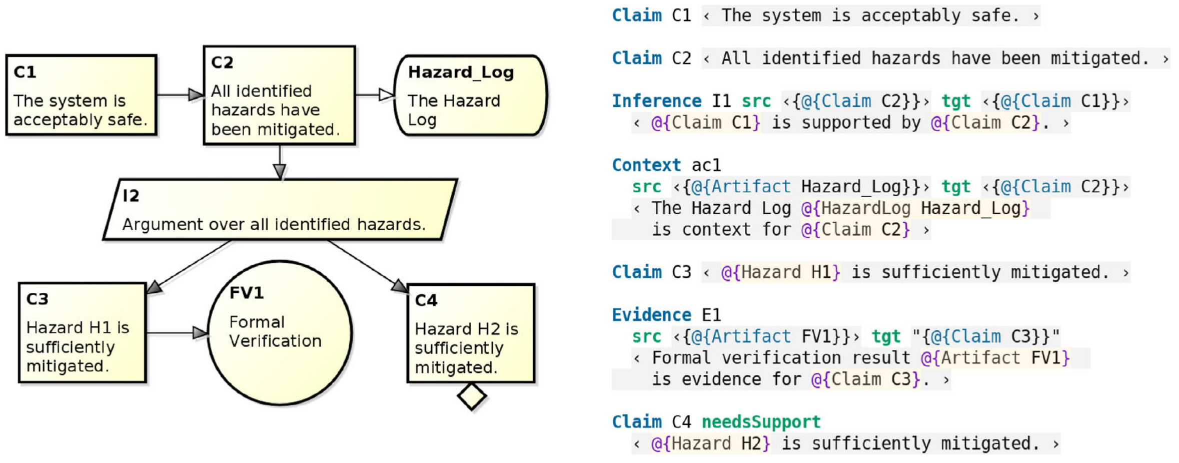
The class `Claim` in Fig. 3 inherits from the class `Assertion` the attributes `gid`, `content_assoc`, and `assertionDeclaration` of type `assertionDeclarations_t`. The other child class of `Assertion` is `AssertedRelationship`, as shown below.

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

doc_class AssertedInference = AssertedRelationship +
  isCounter::bool <= False
  source::"Assertion set"
  target::"Assertion set"

doc_class AssertedEvidence = AssertedRelationship +
  isCounter::bool <= False
  source::"ArtifactAsset set"
  target::"Assertion set"
```

`AssertedRelationship` models the relationships between instances of type `ArgumentAsset`, such as the "supported-by" and "in-context-of" arrows of Fig. 2. `isCounter` specifies whether the target of the relation is supported or refuted by the source, and `reasoning_assoc` is an association to `ArgumentReasoning`, which models GSN strategies in SACM. The attributes `source` and `target`, both of type `ArgumentAsset`, specify the source and target for the relation. Rather than placing them directly in `AssertedRelationship` we put them in the concrete subclasses, as this means they can be specialised to enforce OCL constraints in the reference meta-model [Obj20]. The various kinds of relationship classes, such as `AssertedInference` and `AssertedEvidence`, are then created as subclasses. An `AssertedInference` can only connect assertions, and an `AssertedEvidence` can only connect an evidential artifact to an assertion. These constraints are enforced by DOF when model instances are created.

4.2. Interactive assurance language

Interactive Assurance Language (IAL) is our assurance language with a concrete syntax consisting of various Isabelle commands that extend the document model shown in Fig. 4. Each command performs a number of checks: (1) standard Isabelle checks (§ 2); (2) OCL-style constraints imposed on the attributes by SACM (provided by DOF); (3) well-formedness checks against the meta-model, e.g. instances comply to the type restrictions imposed by the SACM datatypes.


C1
The system is acceptably safe.
C2
All identified hazards have been mitigated.
Hazard_Log
The Hazard Log
I2
Argument over all identified hazards.
C3
Hazard H1 is sufficiently mitigated.
FV1
Formal Verification
C4
Hazard H2 is sufficiently mitigated.

```
Claim C1 ‹ The system is acceptably safe. ›

Claim C2 ‹ All identified hazards have been mitigated. ›

Inference I1 src ‹{@{Claim C2}}› tgt ‹{@{Claim C1}}›
  ‹ @{Claim C1} is supported by @{Claim C2}. ›

Context ac1
  src ‹{@{Artifact Hazard_Log}}› tgt ‹{@{Claim C2}}›
  ‹ The Hazard Log @{HazardLog Hazard_Log}
    is context for @{Claim C2} ›

Claim C3 ‹ @{Hazard H1} is sufficiently mitigated. ›

Evidence E1
  src ‹{@{Artifact FV1}}› tgt "{@{Claim C3}}"
  ‹ Formal verification result @{Artifact FV1}
    is evidence for @{Claim C3}. ›

Claim C4 needsSupport
  ‹ @{Hazard H2} is sufficiently mitigated. ›
```

Fig. 7. A GSN diagram and its translation to the interactive assurance language (IAL)

IAL instantiates the `doc_class`es from § 4.1 to create SACM model elements in Isabelle, for example, the command `Claim` creates a model element of the class `Claim`. Attributes and associations of a class have a concrete syntax represented by an Isabelle (green) subcommand. The grammar of the IAL commands for creating argumentation elements is shown below.

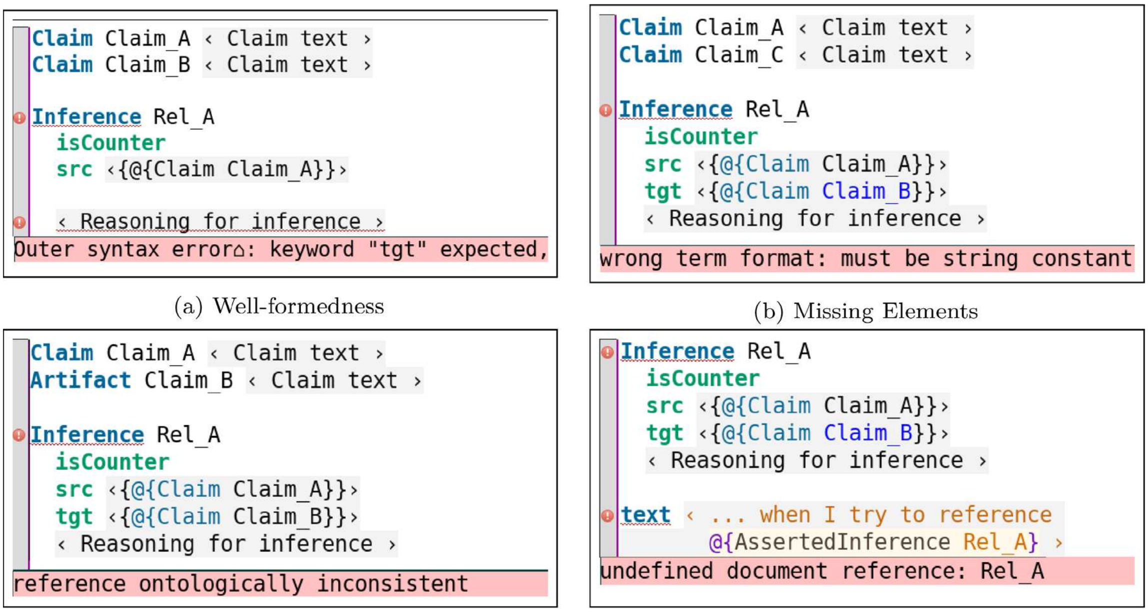
```
<AssertDecl>    := asserted | axiomatic | assumed | defeated | needsSupport
<ClaimComm>     := Claim <gid> isAbstract? isCitation? (metaClaims <gid>*)? <AssertDecl>? <Description>
<InferenceComm> := Inference <gid> <AssertDecl> (src <gid>*) (tgt <gid>*) <Description>
<ContextComm>   := Context <gid> <AssertDecl> (src <gid>*) (tgt <gid>*) <Description>
<EvidenceComm>  := Evidence <gid> <AssertDecl> (src <gid>*) (tgt <gid>*) <Description>
```

`Claim` creates a model element of type `Claim` with an identifier (`gid`), and description contained in a `MultiLangString`. The antiquotation `@{Claim <<gid>>}` can be used to reference the created model element. The subcommands `isAbstract`, `isCitation`, `metaClaims`, and `<AssertDecl>` are optional, with default values being `False`, `False`, `{}` and `asserted`, respectively. The `metaClaims` keyword allows us to link a claim to assertions about this claim, such as the level of confidence in it. `Inference` creates an inference between several model elements of type `ArgumentAsset`. It has subcommands `src` and `tgt` that are both lists of antiquotations pointing to `ArgumentAsset`s. The use of antiquotations to reference the instances ensures that Isabelle will do the checks explained in § 2. `Context` similarly asserts that an instance should be treated as context for another, and `Evidence` associates evidence with a claim. Model elements created by IAL are *semi-formal*, since they can contain both informal content and references to machine checked formal content.

With these commands, IAL can represent a GSN diagram, as shown in Fig. 7. In this translation, each shape and arrow element is mapped to an IAL command. The arrows do not necessarily have labels in GSN, so we allocate these for referencing purposes. Claims `C1`–`C4` are encoded using the `Claim` command. Claim `C1` is supported by `C2` via the inference `I1`, which represents the "supported-by" arrow in the GSN at the left, and uses antiquotations to refer to the two claims. An artifact called `Hazard_Log` is introduced as context for `C2` using the `Context` command, which corresponds to the "in-context-of" arrow to the right of goal `C2` in the GSN diagram. A further evidence artifact `FV1` is used to support claim `C3`, using the `Evidence` command to create `E1`, which corresponds to a "supported-by" arrow. The claim `C4` is left undeveloped, marked using a diamond, and so the corresponding `Claim` is marked by the `needsSupport` keyword.

Figure 8 shows the interactive nature of IAL, and some of the error types. In Fig. 8a, the `Inference` command expects a source followed by target element, but the latter is missing, and so IAL raises the error message at the bottom. The exclamation marks on the left denote from where the error originates. In the jEdit interface, these errors are raised interactively whilst the user is typing. Moreover, this kind of check ensures that the model elements produced conform to the SACM reference meta-model.

In Fig. 8b, the target is specified (`Claim_B`), but it refers to a claim that does not exist (hence the blue colour), and so IAL again raises an error message. In Fig. 8c, an element called `Claim_B` exists, but it is of the wrong type. `Claim_B` is an artifact, which violates the OCL constraints of the SACM standard [Obj20], and so DOF raises an ontological error.



```
Claim Claim_A ‹ Claim text ›
Claim Claim_B ‹ Claim text ›

Inference Rel_A
  isCounter
  src ‹{@{Claim Claim_A}}›

  ‹ Reasoning for inference ›
Outer syntax error⌂: keyword "tgt" expected,
```

(a) Well-formedness

```
Claim Claim_A ‹ Claim text ›
Claim Claim_C ‹ Claim text ›

Inference Rel_A
  isCounter
  src ‹{@{Claim Claim_A}}›
  tgt ‹{@{Claim Claim_B}}›
  ‹ Reasoning for inference ›
wrong term format: must be string constant
```

(b) Missing Elements

```
Claim Claim_A ‹ Claim text ›
Artifact Claim_B ‹ Claim text ›

Inference Rel_A
  isCounter
  src ‹{@{Claim Claim_A}}›
  tgt ‹{@{Claim Claim_B}}›
  ‹ Reasoning for inference ›
reference ontologically inconsistent
```

(c) Element Typing

```
Inference Rel_A
  isCounter
  src ‹{@{Claim Claim_A}}›
  tgt ‹{@{Claim Claim_B}}›
  ‹ Reasoning for inference ›

text ‹ ... when I try to reference
    @{AssertedInference Rel_A} ›
undefined document reference: Rel_A
```

(d) Cascading Errors

Fig. 8. The interactive nature of error handling in IAL

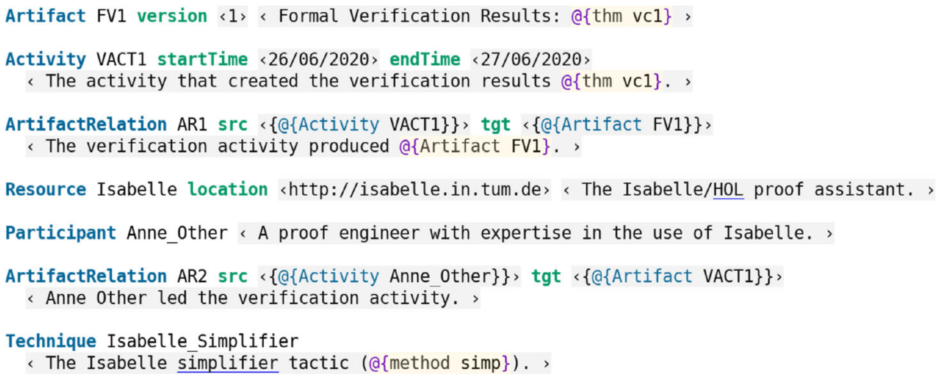
Finally, Fig. 8d shows the cascading effect of errors: `Claim_B` does not exist, the element `Rel_A` fails to process, and consequently an attempt to reference it also fails. This kind of cascading can also be used to detect proof failures following an update to a model and failed verification.

In addition to argumentation commands, we have also implemented several commands for creating different kinds of SACM artifacts, for assurance case context and evidence, as show below.

```
<ArtifactComm>    := Artifact <gid> (version <string>)? (date <string>)? <Description>
<RequirementComm> := Requirement <gid> (version <string>)? (date <string>)? <Description>
<ResourceComm>    := Resource <gid> (location <URI>) <Description>
<ActivityComm>    := Activity <gid> (startTime <string>)? (endTime <string>)? <Description>
<EventComm>       := Event <gid> (occurence <string>)? <Description>
<ParticipantComm> := Participant <gid> <Description>
<TechniqueComm>   := Technique <gid> <Description>
<ArtifactRelComm> := ArtifactRelation <gid> src <gid>* tgt <gid>* <Description>
```

With the exception of `Requirement`, these artifact classes are adopted from the SACM standard [WKD+19]. They allow us to model the various artifacts created during the development and assurance lifecycle, and the relationships between them, for the purposes of traceability. The `Artifact` command represents a unit of data produced during the lifecycle, such as a specification or verification results. It can be annotated with a version, and the creation date. The `Requirement` command can be used to represent requirements, which contain the same information as artifacts, but are marked as requirements that should be satisfied. The `Resource` command can be used to model a link to an external resource, such as a standard or code base, which is uniquely represented by a URI. The `Activity` command models an activity or process, with a start time and end time, and `Event` similarly represents a timed and dated event. A `Participant` models an actor that takes part in the lifecycle, such as a developer, and `Technique` models a technique, such as a modelling language or formal method, that is applied in the creation of artifacts. Finally, `ArtifactRelation` allows us to relate two artifacts.

An example using the artifact commands is shown in Fig. 9, which further elaborates the verification result in Fig. 7. The formal verification result `FV1` is an artifact, with version 1, that points to the Isabelle theorem `vc1`. The result was created during a verification activity, `VACT1`, as shown using the artifact relation `AR1`. Isabelle was used to perform the proof, which is modelled using a `Resource` that links to the Isabelle website. The verification activity was led by a proof engineer, Anne Other, who is modelled as a `Participant`, and linked to the verification activity by a further artifact relation. The specific technique used for the proof was the Isabelle simplifier, which is modelled as a `Technique`, and contains a link to the proof method *simp*.



```
Artifact FV1 version ‹1› ‹ Formal Verification Results: @{thm vc1} ›

Activity VACT1 startTime ‹26/06/2020› endTime ‹27/06/2020›
  ‹ The activity that created the verification results @{thm vc1}. ›

ArtifactRelation AR1 src ‹{@{Activity VACT1}}› tgt ‹{@{Artifact FV1}}›
  ‹ The verification activity produced @{Artifact FV1}. ›

Resource Isabelle location ‹http://isabelle.in.tum.de› ‹ The Isabelle/HOL proof assistant. ›

Participant Anne_Other ‹ A proof engineer with expertise in the use of Isabelle. ›

ArtifactRelation AR2 src ‹{@{Activity Anne_Other}}› tgt ‹{@{Artifact VACT1}}›
  ‹ Anne Other led the verification activity. ›

Technique Isabelle_Simplifier
  ‹ The Isabelle simplifier tactic (@{method simp}). ›
```

Fig. 9. Specification of artifacts and traceability in IAL

We have now developed Isabelle/SACM and our IAL. In the next section, we consider the modelling verification of the Tokeneer system.

5. Modelling and verification of Tokeneer

In this section we present a novel mechanisation of Tokeneer in Isabelle/UTP [FZW16, FZN$^+$19] to provide evidence for the AC. This model encodes the formal functional specification (41_2) in the modular assurance case in Fig. 6. In [C$^+$08b], the satisfaction of the SFRs are argued semi-formally using the functional specification, but here we provide a formal proof. We focus on the verification of three of the requirements: SFR1 (the most challenging of the six), SFR3, and SFR6, and describe the necessary model elements. All formal artifacts in this section are accompanied by links to their mechanisations in our repository.

5.1. Modelling and mechanisation

The TIS functional specification [C$^+$08a] describes 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 monitored and controlled variables shared with the real world. 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 complex state transition system, with around 50 operations for enrolling the station, performing various administrative operations, such as archiving log files and updating the configuration file, and the user entry operations. The main user entry operations are illustrated in Fig. 10 (cf. [C$^+$08a, page 43]), where each transition corresponds to an operation. 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 using hierarchical state space types, with invariants adapted from the Z specification [C$^+$08a]. We define the operations using GCL [Dij75] rather than the Z schemas directly, to enable syntax-directed reasoning. The syntax of Z [Spi89], though maximally flexible, does not easily lend itself to such reasoning, since every operation schema contains a set of conjoined predicates which must be considered in turn. In contrast, as illustrated in § 2.3, it is straightforward to calculate the weakest precondition of a GCL program. Within these constraints, we have endeavoured to remain faithful to the function specification by representing each of the state and operation schemas and using the same naming and overall structure. The use of GCL means that the model is also much closer to a program, and consequently refinement to code should be straightforward. Moreover, since GCL has a denotational semantics in UTP's relational calculus [HH98], it is possible to prove equivalences between program notation and Z-style schema notation.

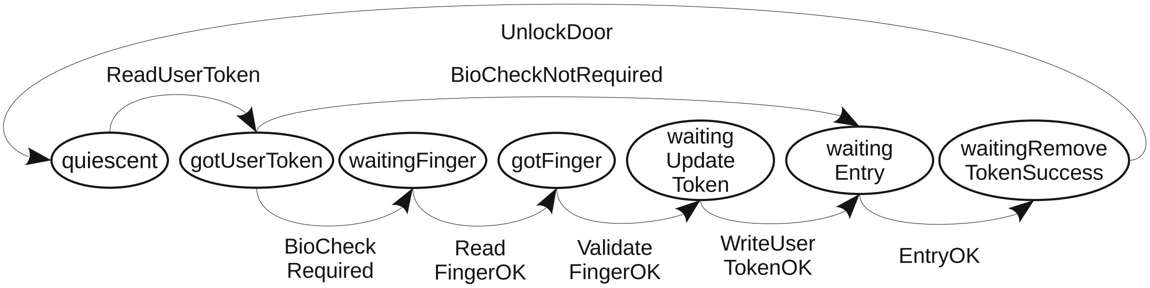


Fig. 10. The main states and transitions of the TIS unlocking procedure

5.2. State space

We first describe the state space of the TIS state machine:

Definition 5.1 (TIS types and state space).

$$
\begin{aligned}
LATCH &\triangleq \mathsf{unlocked} \mid \mathsf{locked} \\
DOOR &\triangleq \mathsf{open} \mid \mathsf{closed} \\
TOKENTRY &\triangleq \mathsf{noT} \mid \mathsf{badT} \mid \mathsf{goodT}\ Token \\
PRESENCE &\triangleq \mathsf{present} \mid \mathsf{absent} \\
PRIVELEGE &\triangleq \mathsf{userOnly} \mid \mathsf{guard} \mid \mathsf{securityOfficer} \mid \mathsf{auditManager} \\
ADMINOP &\triangleq \mathsf{archiveLog} \mid \mathsf{updateConfigData} \mid \mathsf{overrideLock} \mid \mathsf{shutdownOp} \\
FLOPPY &\triangleq \mathsf{noFloppy} \mid \mathsf{emptyFloppy} \mid \mathsf{badFloppy} \mid \mathsf{configFile}(configFileOf : Config) \mid \cdots \\
Config &\triangleq \begin{bmatrix} alarmSilentDuration : TIME, latchUnlockDuration : TIME, \\ tokenRemovalDutation : TIME, enclaveClearance : CLEARANCE, \cdots \end{bmatrix} \\
IDStation &\triangleq \begin{bmatrix} currentUserToken : TOKENTRY, currentTime : TIME, \\ userTokenPresence : PRESENCE, status : STATUS, \\ enclaveStatus : ENCLAVESTATUS, currentDisplay : DISPLAYMESSAGE, \\ issuerKey : USER \nrightarrow KEYPART, rolePresent : PRIVELEGE\ option, \\ availableOps : ADMINOP\ set, currentAdminOp : ADMINOP\ option, \\ ifloppy : FLOPPY, config : Config, \cdots \end{bmatrix} \\
Controlled &\triangleq \begin{bmatrix} latch : LATCH, alarm : ALARM, display : DISPLAYMESSAGE, screen : Screen \end{bmatrix} \\
Monitored &\triangleq \begin{bmatrix} now : TIME, finger : FINGERPRINTTRY, \\ userToken : TOKENTRY, floppy : FLOPPY, keyboard : KEYBOARD \end{bmatrix} \\
RealWorld &\triangleq [mon : Monitored, ctrl : Controlled] \\
SystemState &\triangleq [rw : RealWorld, tis : IDStation]
\end{aligned}
$$

A collection of algebraic data types characterise the state of system elements, including the door, latch, and token. The type $\alpha\ option$ represents an optional value that can either take the value *None*, meaning that it is undefined, or $Some\, x$ for $x : \alpha$, meaning that it is defined. The function $the : \alpha\ option \to \alpha$ allows us to extract the value from a defined value. We define state types for the TIS state, 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 (*rw*) and the TIS (*tis*).

Variable *currentUserToken* represents the last token presented to the TIS, and *userTokenPresence* indicates whether a token is currently present. The variable *status* is used to record the state the TIS is in, and can take the values indicated in the state bubbles of Fig. 10. Variable *issuerKey* is a partial function representing the public key chain, which is needed to authorise user entry. Variables *rolePresent*, *availableOps*, and *currentAdminOp* are used to represent the presence of an Admin, the available operations for this Admin, and the current operation being executed.

In addition to the state types, we also encode a number of predicates that represent the invariants of seven Z state schemas. These effectively encode low-level well-formedness constraints for the types; the higher level invariants are considered in § 5.4. The predicate representing the invariants associated with the Admin variables is shown below.

Definition 5.2 (Administrator Invariants).

$$
\mathit{Admin} \triangleq \left(\begin{array}{l}
(\mathit{rolePresent} \neq \mathit{None} \Rightarrow \mathit{the}(\mathit{rolePresent}) \in \{\mathsf{guard}, \mathsf{auditManager}, \mathsf{securityOfficer}\}) \\
\wedge\ (\mathit{rolePresent} = \mathit{None} \Rightarrow \mathit{availableOps} = \{\}) \\
\wedge\ (\mathit{rolePresent} = \mathit{Some}(\mathsf{guard}) \Rightarrow \mathit{availableOps} = \{\mathsf{overrideLock}\}) \\
\wedge\ (\mathit{rolePresent} = \mathit{Some}(\mathsf{auditManager}) \Rightarrow \mathit{availableOps} = \{\mathsf{archiveLog}\}) \\
\wedge\ (\mathit{rolePresent} = \mathit{Some}(\mathsf{securityOfficer}) \Rightarrow \mathit{availableOps} = \{\mathsf{updateConfigData}, \mathsf{shutdownOp}\}) \\
\wedge\ (\mathit{currentAdminOp} \neq \mathit{None} \Rightarrow \mathit{the}(\mathit{currentAdminOp}) \in \mathit{availableOps} \wedge \mathit{rolePresent} \neq \mathit{None})
\end{array}\right)
$$

This predicate closely corresponds to the *Admin* schema in the functional specification [C⁺08a, page 22]. It states, firstly, that if a role is present, it must be one of the three Admin roles. Conversely, if no roles are present then no Admin operations are available to be executed. The next three implications assign possible operations to the given Admin roles. The final predicate states that if an Admin operation is being executed, then it must be one of the available operations and there must be a role present. We collect the seven well-formedness predicates in *TIS-wf*, as defined below.

Definition 5.3 (Well-formedness Properties).

$$
\mathit{TIS\text{-}wf} \triangleq (\mathit{DoorLatchAlarm} \wedge \mathit{Floppy} \wedge \mathit{KeyStore} \wedge \mathit{Admin} \wedge \mathit{Config} \wedge \mathit{AdminToken} \wedge \mathit{UserToken})
$$

The verification of the TIS SFRs depends on these state predicates being invariant for all of the operations.

5.3. Operations

We now specify the operations over *IDStation*, a selection of these are shown below[6]:

Definition 5.4 (User Entry Operations).

$$
\begin{array}{rl}
\mathit{ReadUserToken} \triangleq & \left(\begin{array}{l}
\mathit{enclaveStatus} \in \left\{\begin{array}{l}\mathsf{enclaveQuiescent}, \\ \mathsf{waitingRemoveAdminTokenFail}\end{array}\right\} \\
\wedge\ \mathit{status} = \mathsf{quiescent} \wedge \mathit{userTokenPresence} = \mathsf{present}
\end{array}\right) \\
& \longrightarrow \mathit{currentDisplay} := \mathsf{wait} \fatsemi \mathit{status} := \mathsf{gotUserToken} \\[1ex]
\mathit{BioCheckRequired} \triangleq & \left(\begin{array}{l}
\mathit{status} = \mathsf{gotUserToken} \wedge \mathit{userTokenPresence} = \mathsf{present} \\
\wedge\ \mathit{UserTokenOK} \wedge (\neg \mathit{UserTokenWithOKAuthCert})
\end{array}\right) \\
& \longrightarrow \mathit{status} := \mathsf{waitingFinger} \fatsemi \mathit{currentDisplay} := \mathsf{insertFinger} \\[1ex]
\mathit{ReadFingerOK} \triangleq & \left(\begin{array}{l}
\mathit{status} = \mathsf{waitingFinger} \wedge \mathit{fingerPresence} = \mathsf{present} \\
\wedge\ \mathit{userTokenPresence} = \mathsf{present}
\end{array}\right) \\
& \longrightarrow \mathit{status} := \mathsf{gotFinger} \fatsemi \mathit{currentDisplay} := \mathsf{wait} \\[1ex]
\mathit{UnlockDoor} \triangleq & \left(\begin{array}{l}
\mathit{latchTimeout} := \mathit{currentTime} + \mathit{latchUnlockDuration} \fatsemi \\
\mathit{alarmTimeout} := \mathit{currentTime} + \mathit{latchUnlockDuration} + \mathit{alarmSilentDuration} \fatsemi \\
\mathit{currentLatch} := \mathsf{unlocked} \fatsemi \mathit{doorAlarm} := \mathsf{silent}
\end{array}\right)
\end{array}
$$

[6] The TIS operations have been mechanised using the same names as in [C⁺08a].

$$UnlockDoorOK \triangleq \begin{array}{l} \left(\begin{array}{l} status = \mathsf{waitingRemoveTokenSuccess} \\ \land\ userTokenPresence = \mathsf{absent} \end{array}\right) \\ \longrightarrow \begin{array}{l} UnlockDoor \fatsemi status := \mathsf{quiescent} \fatsemi \\ currentDisplay := \mathsf{doorUnlocked} \end{array} \end{array}$$

Each operation is guarded by execution conditions and consists of several assignments. *BioCheckRequired* requires that the current state is $\mathsf{gotUserToken}$, the user token is $\mathsf{present}$, 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. We give the definition of *UserTokenOK* below.

$$UserTokenOK \triangleq \left(\exists\, t \bullet \begin{array}{l} currentUserToken = \mathsf{goodT}(t) \land t \in CurrentToken\ \land \\ (\exists\, c \in IDCert \bullet idCert(t) = c \land CertOKc)\ \land \\ (\exists\, c \in PrivCert \bullet privCert(t) = c \land CertOKc)\ \land \\ (\exists\, c \in IandACert \bullet iandACert(t) = c \land CertOKc) \end{array}\right)$$

It requires that *currentUserToken* contains a token, which is current (*CurrentToken*), and has valid ID, privilege, and I&A certificates. The definitions of the omitted predicates can be found elsewhere [C+08a].

Assuming these preconditions hold, operation *BioCheckRequired* updates the state to $\mathsf{waitingFinger}$ and the display with an instruction to provide a fingerprint. *ReadFingerOK* requires that the state is $\mathsf{waitingFinger}$, and checks whether both a finger and user token are present. If they are, then the state switches to $\mathsf{gotFinger}$, and the display is updated to wait. *UnlockDoorOK* requires that the state is $\mathsf{waitingRemoveTokenSuccess}$, and the token has been removed. It unlocks the door, using the auxiliary operation *UnlockDoor*, returns the status to $\mathsf{quiescent}$, and updates the display. *UnlockDoor* both unlocks the latch, and also updates two timeout variables, *latchTimeout* and *alarmTimeout*. The former is used to close the latch after a certain period, and the latter to sound an alarm if the door is left open.

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.

$$UEC(Op) \triangleq tis{:}[Op] \fatsemi rw{:}[mon{:}now \leq mon{:}now' \land ctrl' = ctrl]$$

In Z, this functionality is provided by the schema *UserEntryContext* [C+08a], from which we derive the name *UEC*. It promotes *Op* to act on *tis*, and composes this with a relational predicate that constrains the real-world variables (*rw*). The behaviour of all monitored variables other than *now* is arbitrary, and all controlled variables are unchanged. This separation enables modular reasoning, since we can promote invariants of the TIS to any real world context using Theorem 2.1 and the following Hoare logic theorem.

Theorem 5.1 (TIS Promotion). If $\{I\}\, P\, \{I\}$ then $\{I_{\uparrow tis}\}\, UEC(P)\, \{I_{\uparrow tis}\}$

This shows that if I is an invariant of P, an operation of the TIS, then I in the extended state space is an invariant of promoted operation. For comparison with the original Z operation schemas, we give the *ReadUserToken* schema below:

$$\begin{array}{|l}
\multicolumn{1}{l}{ReadUserToken} \\ \hline
UserEntryContext \\[1ex]
\Xi\, UserToken \\
\Xi\, DoorLatchAlarm \\
\Xi\, Stats \\[1ex]
AddElementsToLog \\ \hline
enclaveStatus \in \{enclaveQuiescent, waitingRemoveAdminTokenFail\} \\[1ex]
status = quiescent \\
userTokenPresence = present \\[1ex]
currentDisplay' = wait \\
status' = gotUserToken \\ \hline
\end{array}$$

We have employed a pattern for conversion from Z to GCL. Every conditional predicate, for example $\mathit{status} = \mathit{quiescent}$, becomes a guard in Definition 5.4. Every primed variable equation, such as $\mathit{status}' = \mathit{gotUserToken}$, becomes an assignment, and all of the resulting commands are sequentially composed. Nevertheless, we preserve the non-determinism of the original model, and these assignments are equivalent to primed variable equations since in UTP we denote an assignment as follows [HH98, CW06]:

$$(x := e) \triangleq x' = e \wedge y' = y \wedge \cdots \wedge z' = z$$

Consequently, the operations could be expressed as relational expressions. Using UEC, we promote each operation, for example $\mathit{TISReadToken} \triangleq \mathit{UEC}(\mathit{ReadToken})$, to effectively include $\mathit{UserEntryContext}$.

In Z, invariants of the state are imposed through the inclusion of state schemas, such as $\mathit{UserToken}$. Here, we do not impose these but we prove that each operation preserves each invariant in § 5.4. This sometimes requires that we add extra assignments to satisfy the invariant, as we illustrate below.

We next define some of the key Admin operations, which are necessary to prove the security properties.

Definition 5.5 (Admin Operations).

$$\mathit{OverrideDoorLockOK} \triangleq \begin{array}{l} \left(\begin{array}{l} \mathit{enclaveStatus} = \mathsf{waitingStartAdminOp} \\ \wedge\ \mathit{adminTokenPresence} = \mathsf{present} \\ \wedge\ \mathit{currentAdminOp} = \mathit{Some}(\mathsf{overrideLock}) \end{array}\right) \\ \longrightarrow \begin{array}{l} \mathit{screenMsg} := \mathsf{requestAdminOp} \fatsemi \mathit{currentDisplay} := \mathsf{doorUnlocked} \fatsemi \\ \mathit{enclaveStatus} = \mathsf{enclaveQuiescent} \fatsemi \mathit{UnlockDoor} \fatsemi \\ \mathit{currentAdminOp} := \mathit{None} \end{array} \end{array}$$

$$\mathit{FinishUpdateConfigOK} \triangleq \begin{array}{l} \left(\begin{array}{l} \mathit{enclaveStatus} = \mathsf{waitingFinishAdminOp} \\ \wedge\ \mathit{adminTokenPresence} = \mathsf{present} \\ \wedge\ \mathit{currentAdminOp} = \mathit{Some}(\mathsf{updateConfigData}) \\ \wedge\ \mathit{floppyPresence} = \mathsf{present} \wedge \mathit{currentFloppy} \in \mathit{range}(\mathit{configFile}) \\ \wedge\ \mathit{ValidConfig}(\mathit{configFileOf}(\mathit{currentFloppy})) \end{array}\right) \\ \longrightarrow \begin{array}{l} \mathit{config} := \mathit{configFileOf}(\mathit{currentFloppy}) \fatsemi \\ \mathit{screenMsg} := \mathsf{requestAdminOp} \fatsemi \\ \mathit{screenConfig} := \mathit{displayConfigData}(\mathit{config}) \fatsemi \\ \mathit{enclaveStatus} = \mathsf{enclaveQuiescent} \fatsemi \mathit{currentAdminOp} := \mathit{None} \end{array} \end{array}$$

$$\mathit{AdminLogout} \triangleq \begin{array}{l} \mathit{rolePresent} \neq \mathit{None} \\ \longrightarrow\ \mathit{rolePresent} := \mathit{None} \fatsemi \mathit{currentAdminOp} := \mathit{None} \fatsemi \mathit{availableOps} := \{\} \end{array}$$

$$\mathit{ShutdownOK} \triangleq \begin{array}{l} \left(\begin{array}{l} \mathit{enclaveStatus} = \mathsf{waitingStartAdminOp} \\ \wedge\ \mathit{currentAdminOp} = \mathit{Some}(\mathsf{shutdownOp}) \\ \wedge\ \mathit{currentDoor} = \mathsf{closed} \end{array}\right) \\ \longrightarrow \begin{array}{l} \mathit{LockDoor} \fatsemi \mathit{AdminLogout} \fatsemi \mathit{screenMsg} = \mathsf{clear} \fatsemi \\ \mathit{enclaveStatus} := \mathsf{shutdown} \fatsemi \mathit{currentDisplay} := \mathsf{blank} \end{array} \end{array}$$

$\mathit{OverrideDoorLockOK}$ allows the door to be unlocked when an Admin has already logged in who can execute the $\mathsf{overrideLock}$ command, that is, an Admin with the role guard. If the enclave is awaiting an Admin command, an Admin token is present, and the Admin gives the $\mathsf{overrideLock}$ command, then the door is unlocked and the enclave returns to awaiting another Admin command. $\mathit{FinishUpdateConfigOK}$ is the second part of a two stage process for updating the configuration file. The first step checks whether a configuration file floppy has been inserted. In this second step, if the command $\mathsf{updateConfigData}$ has been selected and a valid floppy has been inserted, then the config is updated, displayed on the screen, and the enclave again returns to the main menu. Finally, $\mathit{ShutdownOK}$ is used to shutdown the TIS. If an Admin is logged in, selects the $\mathsf{shutdownOp}$ command, and the door is closed, then the operation locks the door, logs the Admin out, blanks the screen and display, and sets the status to $\mathsf{shutdown}$. The auxiliary operation $\mathit{AdminLogout}$ has one more assignment than the corresponding Z schema [C+08a, page 41], $\mathit{availableOps} := \{\}$, to ensure that the Admin invariants in Definition 5.2 are satisfied. In Z, this is implicit because the invariants are enforced at each stage.

The overall behaviour of the TIS operations is given below.

Definition 5.6 (Top-Level TIS Operations).

$$\mathit{TISUserEntryOp} \triangleq \begin{pmatrix} \mathit{TISReadUserToken} \sqcap \mathit{TISValidateUserToken} \\ \sqcap\ \mathit{TISReadFinger} \sqcap \mathit{TISValidateFinger} \\ \sqcap\ \mathit{TISWriteUserToken} \sqcap \mathit{TISValidateEntry} \\ \sqcap\ \mathit{TISUnlockDoor} \sqcap \mathit{TISCompleteFailedAccess} \end{pmatrix}$$

$$\mathit{TISAdminOp} \triangleq \begin{pmatrix} \mathit{TISOverrideDoorLockOp} \sqcap \mathit{TISShutdownOp} \\ \sqcap\ \mathit{TISUpdateConfigDataOp} \sqcap \mathit{TISArchiveLogOp} \end{pmatrix}$$

$$\mathit{TISOp} \triangleq \begin{pmatrix} \mathit{TISEnrolOp} \sqcap \mathit{TISUserEntryOp} \sqcap \mathit{TISAdminLogon} \sqcap \mathit{TISStartAdminOp} \\ \sqcap\ \mathit{TISAdminOp} \sqcap \mathit{TISAdminLogout} \sqcap \mathit{TISIdle} \end{pmatrix}$$

We omit several operations, though these have all been mechanised. 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 achieved by composition with the following relational update operation.

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

This allows time to advance, allows other monitored variables to change, and copies the digital state of the latch and display to the corresponding controlled variables. The system transitions are described by $\mathit{TISOp} \fatsemi \mathit{TISUpdate}$.

5.4. Formal verification

In this section, we verify three SFRs of the formal model using Isabelle/UTP. We first formalise the TIS state invariants that are necessary to prove the SFRs[7]:

Definition 5.7 (TIS State Invariants Selection).

$$\mathit{Inv}_1 \triangleq \begin{pmatrix} \mathit{status} \in \left\{ \begin{matrix} \mathsf{gotFinger}, \mathsf{waitingFinger}, \mathsf{waitingUpdateToken} \\ \mathsf{waitingEntry}, \mathsf{waitingUpdateTokenSuccess} \end{matrix} \right\} \\ \Rightarrow (\mathit{UserTokenWithOKAuthCert} \vee \mathit{UserTokenOK}) \end{pmatrix}$$

$$\mathit{Inv}_2 \triangleq \begin{pmatrix} \mathit{status} \in \{\mathsf{waitingEntry}, \mathsf{waitingRemoveTokenSuccess}\} \\ \Rightarrow (\mathit{UserTokenWithOKAuthCert} \vee \mathit{FingerOK}) \end{pmatrix}$$

$$\mathit{Inv}_3 \triangleq (\mathit{rolePresent} \neq \mathit{None} \Rightarrow \mathit{AdminTokenOK})$$

$$\mathit{Inv}_4 \triangleq \begin{pmatrix} \mathit{currentAdminOp} \in \{\mathit{Some}(\mathsf{shutdownOp}), \mathit{Some}(\mathsf{overrideLock})\} \\ \Rightarrow \mathit{ownName} \neq \mathit{None} \end{pmatrix}$$

$$\mathit{Inv}_5 \triangleq \begin{pmatrix} \mathit{adminTokenPresent} = \mathit{present} \wedge \mathit{rolePresent} \neq \mathit{None} \\ \Rightarrow \mathit{rolePresent} = \mathit{Some}(\mathit{role}(\mathit{authCert}(\mathit{ofGoodT}(\mathit{currentAdminToken})))) \end{pmatrix}$$

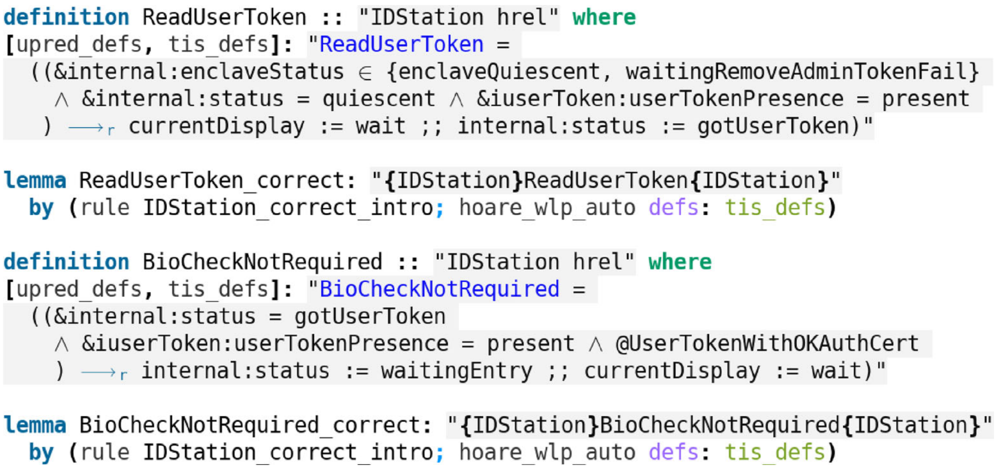
$$\mathit{Inv}_6 \triangleq \mathit{enclaveStatus} \in \{\mathsf{waitingStartAdminOp}, \mathsf{waitingFinishAdminOp}\} \Leftrightarrow \mathit{currentAdminOp} \neq \mathit{None}$$

$$\mathit{Inv}_7 \triangleq \mathit{enclaveStatus} = \mathsf{gotAdminToken} \Rightarrow \mathit{rolePresent} = \mathit{None}$$

$$\mathit{TIS\text{-}inv} \triangleq \mathit{TIS\text{-}wf} \wedge \mathit{Inv}_1 \wedge \mathit{Inv}_2 \wedge \mathit{Inv}_3 \wedge \mathit{Inv}_4 \wedge \mathit{Inv}_5 \wedge \mathit{Inv}_6 \wedge \mathit{Inv}_7 \cdots \mathit{Inv}_{10}$$

Inv_1 states that whenever the TIS is in a state beyond $\mathsf{gotUserToken}$, then either a valid authorisation certificate is present, or else the user token is valid. It corresponds to the first invariant in the *IDStation* schema [C+08a, page 26]. However, we need to add an extra state, $\mathsf{updateTokenSuccess}$ and strengthen the consequent. The consequent originally only required that there is a token with a valid authorisation certificate, which may not be the case if a fingerprint has not yet been taken. Inv_2 states that whenever in state $\mathsf{waitingEntry}$ or $\mathsf{waitingRemoveTokenSuccess}$, then either an authorisation certificate or a valid fingerprint is present.

[7] We adopt a different order for the invariants than our mechanisation, for the sake of presentation.



```
definition ReadUserToken :: "IDStation hrel" where
[upred_defs, tis_defs]: "ReadUserToken =
  ((&internal:enclaveStatus ∈ {enclaveQuiescent, waitingRemoveAdminTokenFail}
    ∧ &internal:status = quiescent ∧ &iuserToken:userTokenPresence = present
   ) ⟶ᵣ currentDisplay := wait ;; internal:status := gotUserToken)"

lemma ReadUserToken_correct: "{IDStation}ReadUserToken{IDStation}"
  by (rule IDStation_correct_intro; hoare_wlp_auto defs: tis_defs)

definition BioCheckNotRequired :: "IDStation hrel" where
[upred_defs, tis_defs]: "BioCheckNotRequired =
  ((&internal:status = gotUserToken
    ∧ &iuserToken:userTokenPresence = present ∧ @UserTokenWithOKAuthCert
    ) ⟶ᵣ internal:status := waitingEntry ;; currentDisplay := wait)"

lemma BioCheckNotRequired_correct: "{IDStation}BioCheckNotRequired{IDStation}"
  by (rule IDStation_correct_intro; hoare_wlp_auto defs: tis_defs)
```

Fig. 11. Verification of the Tokeneer invariants in Isabelle/UTP

Inv_2 is not present at all in [C+08a], but we found it necessary to satisfy SFR1, specifically to ensure that a valid fingerprint is present. That certain invariants are missing, or too weak, is acknowledged in the TIS Security Properties document [C+08b, page 11], but this does not invalidate the functional specification; it just makes it difficult to formally verify the SFRs.

Invariant Inv_3 states that whenever an Admin role is present, then a valid Admin token is also present ($AdminTokenOK$), so that it is not necessary to explicitly check this in each Admin operation. Similar to Inv_1, it is equivalent to the second $IDStation$ schema invariant [C+08a, page 26], but we again needed to strengthen the consequent. Inv_4 states that if an Admin operation *shutdownOp* or *overrideLock* is selected, then the TIS must have an assigned name (also present in the key store), and hence it must already be enrolled. Inv_5 states that if an Admin token and Admin role are both present, then the role must match with the one contained on the Admin token. This invariant does not seem to be present at all in [C+08a], but we believe that it is certainly necessary to prove SFR1. Inv_6 states that if an Admin operation is starting or finishing, then an Admin operation currently must be selected. Finally, Inv_7 states if the Admin token has just been inserted, then a role is not currently present. We omit the additional three invariants that deal with the screen and enrolment [C+08a] as they are not relevant for SFR verification.

As before, and differently to [C+08a], which imposes the invariants by construction, we prove that each operation preserves the invariants using Hoare logic, similar to [RBC16]:

Theorem 5.2 (TIS Operation Invariants).

- $\{TIS\text{-}inv\}\ TISUserEntryOp\ \{TIS\text{-}inv\}$
- $\{TIS\text{-}inv\}\ TISAdminOp\ \{TIS\text{-}inv\}$

This theorem shows that the user entry and Admin operations never violate the well-formedness properties and ten state invariants. We can therefore assume that they hold to satisfy any requirements. The proof involves discharging verification conditions for a total of 32 operations in Isabelle/UTP, a process that is automated using our proof tactics `hoare_auto` [FBC+20] and `hoare_wlp_auto`. We illustrate this in Fig. 11 for two of the defined operations. We follow the mathematical notation for GCL as much as possible. Each proof first applies an introduction rule, `IDStation_correct_intro`, that splits the goal into the well-formedness and behavioural invariants. Then, `hoare_wlp_auto` is applied to each resulting goal. This high-level automation means that proofs can be adapted for small changes to the operations with minimal intervention.

We use this fact to assure SFR1, which is formalised by the formula FSFR1, that characterises the conditions under which the latch will become unlocked having been previously locked. We can determine these states by application of the weakest precondition calculus [Dij75], which mirrors the (informal) Z schema domain calculations in [C+08b, page 5]. Specifically, we characterise the weakest precondition under which execution of $TISOp$ followed by $TISUpdate$ leads to a state satisfying $rw{:}ctrl{:}latch = \mathsf{unlocked}$. We formalise this in the definition below.

Definition 5.8 (Formalisation of SFR1).

$$FSFR1 \triangleq \left(\begin{array}{l} \left(\begin{array}{l} (TISOp \fatsemi TISUpdate)\ \boldsymbol{wp}\ (rw{:}ctrl{:}latch = \mathsf{unlocked}) \\ \wedge\ TIS\text{-}inv \wedge tis{:}currentLatch = \mathsf{locked} \end{array} \right) \\ \quad \Rightarrow \left(\begin{array}{l} (UserTokenOK \wedge FingerOK) \vee UserTokenWithOKAuthCert \\ \vee\ AdminTokenGuardOK \end{array} \right) \end{array} \right)$$

$$AdminTokenGuardOK \triangleq \left(\begin{array}{l} \exists\, t \in TokenWithValidAuth \bullet currentAdminToken = \mathsf{goodT}(t) \\ \wedge\ (\exists\, c \in AuthCert \bullet authCert(t) = Some(c) \wedge role(c) = guard) \end{array} \right)$$

We first state the unlocking precondition for $TISOp$ using $\boldsymbol{wp}$ calculus. Then, we conjoin the $\boldsymbol{wp}$ formula with $tis{:}currentLatch = \mathsf{locked}$ to capture behaviours when the latch was initially locked, and the TIS invariants. This formula needs to imply that either a user token is present, meeting certain conditions, or that an Admin token is present with the role guard, which is formulated using $AdminTokenGuardOK$. The only operation that unlocks the door for users is $UnlockDoorOK$, and for Admins it is $OverrideDoorLockOK$. As a result, we can calculate the following unlocking preconditions.

Theorem 5.3 (Unlocking Preconditions).

$$\begin{array}{l} ((TISUserOp \fatsemi TISUpdate)\ \boldsymbol{wp}\ (rw{:}ctrl{:}latch = \mathsf{unlocked}))[\mathsf{locked}/tis{:}currentLatch] \\ \qquad = (status = \mathsf{waitingRemoveTokenSuccess} \wedge userTokenPresence = \mathsf{absent}) \end{array}$$

$$\begin{array}{l} ((TISAdminOp \fatsemi TISUpdate)\ \boldsymbol{wp}\ (rw{:}ctrl{:}latch = \mathsf{unlocked}))[\mathsf{locked}/tis{:}currentLatch] \\ \qquad = \left(\begin{array}{l} enclaveStatus = \mathsf{waitingStartAdminOp} \wedge adminTokenPresence = \mathsf{present} \\ \wedge\ currentAdminOp = Some(\mathsf{overrideLock}) \wedge rolePresent \neq None \wedge currentAdminOp \neq None \end{array} \right) \end{array}$$

The first equation shows that the precondition for a user unlock is that access is permitted and the token has been removed. The second equation shows that the precondition for an Admin unlock is that the TIS is waiting for an Admin command, an Admin token is present, and the selected command is $\mathsf{overrideLock}$. From these equations we can calculate the unlocking precondition of $TISOp$ itself, which is the disjunction of the two preconditions above. We can then conjoin this with $TIS\text{-}Inv$, since we know that it holds in any state. We show that this composite precondition implies that either a valid user token and fingerprint were present (using Inv_2) or a valid authorisation certificate, or else an Admin is present (using Inv_5), and we can use the well-formedness invariant $Admin$ to show that this Admin must have the $\boldsymbol{\mathsf{guard}}$ role. Consequently, $FSFR1$ can indeed be verified.

Theorem 5.4 ($FSFR1$ is provable).

Proof. By application of weakest precondition and relational calculus.
Proof of $SFR2$ can likely be achieved in a similar way to $SFR1$, but more complex additional invariants are required that depend on time, which we have not been able to formalise for this case study (see § 8).

Next, we consider $SFR3$, which requires that an alarm is raised if the door is left open. This property can be proved more straightforwardly, since it is essentially a property of the well-formedness invariant in $DoorLatchAlarm$ [C+08a, page 23].

Definition 5.9 (Formalisation of SFR3).

$$FSFR3 \triangleq \left(\begin{array}{l} IDStation \wedge currentLatch = lock \wedge currentDoor = open \wedge currentTime \geq alarmTimeout \\ \qquad \Rightarrow doorAlarm = alarming \end{array} \right)$$

This states that if the invariants hold, the latch is locked, the door is open, and the time has advanced beyond the alarm timeout, then the door alarm is sounding. By Theorem 5.2, $DoorLatchAlarm$ always holds and therefore FSFR3 can be verified.

Theorem 5.5 ($FSFR3$ is provable).

Finally, we consider SFR6, which requires that the configuration and floppy can change only when an Admin is logged on. This property can be formulated using the $\boldsymbol{\mathsf{nmods}}$ predicate, and the various laws we proved in Theorem 2.3. Using these laws we can automatically verify that a program does not modify certain variables, and so formalise $SFR6$ as follows.

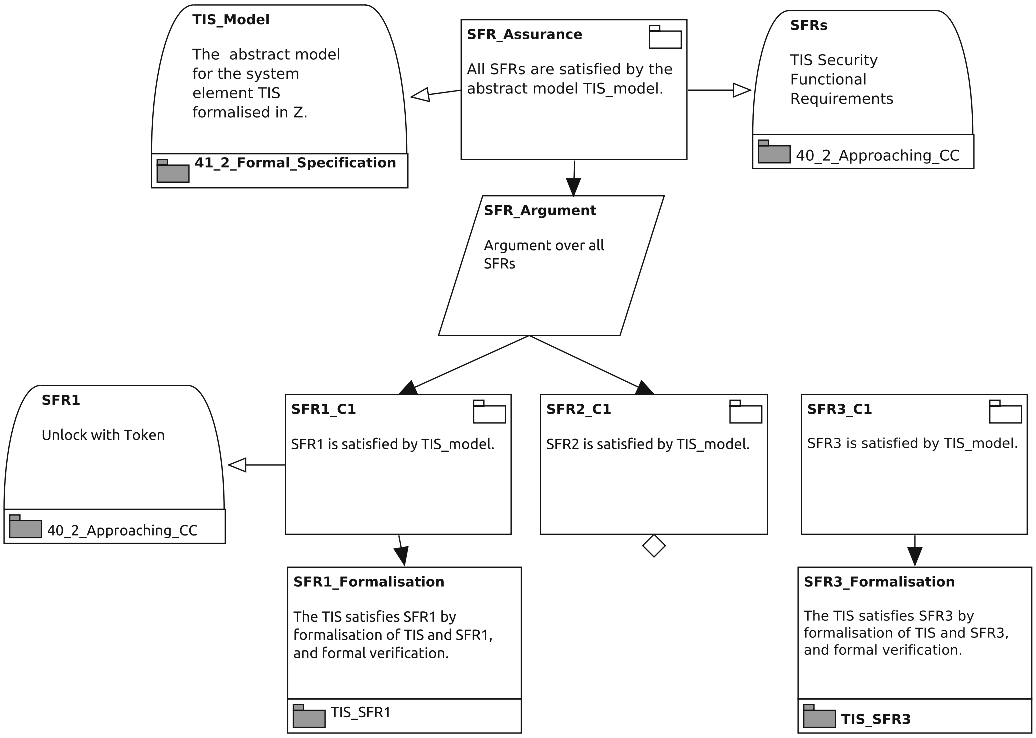

TIS_Model
The abstract model for the system element TIS formalised in Z.
41_2_Formal_Specification
SFR_Assurance
All SFRs are satisfied by the abstract model TIS_model.
SFRs
TIS Security Functional Requirements
40_2_Approaching_CC
SFR_Argument
Argument over all SFRs
SFR1
Unlock with Token
40_2_Approaching_CC
SFR1_C1
SFR1 is satisfied by TIS_model.
SFR2_C1
SFR2 is satisfied by TIS_model.
SFR3_C1
SFR3 is satisfied by TIS_model.
SFR1_Formalisation
The TIS satisfies SFR1 by formalisation of TIS and SFR1, and formal verification.
TIS_SFR1
SFR3_Formalisation
The TIS satisfies SFR3 by formalisation of TIS and SFR3, and formal verification.
TIS_SFR3

Fig. 12. Partial argument for satisfaction of the security functional requirements (TIS_SFRs)

Definition 5.10 $\mathit{FSFR6} \triangleq (\mathit{adminTokenPresence} = \mathsf{absent} \longrightarrow \mathit{TISOp})\ \textbf{\textsf{nmods}}\ \{\mathit{config}, \mathit{floppy}\}$

If we assume that there is not an Admin token, then this means that *TISOp* cannot modify either *config* or *floppy*. For the verification, we can distribute the absence precondition throughout the operations using the law $b \longrightarrow (P \sqcap Q) = (b \longrightarrow P) \sqcap (b \longrightarrow Q)$. One Admin operation can modify *config*, namely *FinishUpdateConfigOK* in Definition 5.5. If we prefix this operation with $\mathit{adminTokenPresence} = \mathsf{absent}$, we obtain the program ***abort*** which does not modify *config*, since this violates the second guard. We can also prove that for every other operation P, P ***nmods*** *config* deductively using Theorem 2.3. Consequently, we can prove *FSFR6*.

Theorem 5.6 (*FSFR6* is provable).

We have now formalised and verified three of the SFRs. In the next section we place these in the context of an assurance case.

6. Mechanising the Tokeener assurance case

In this section, we use ACME and Isabelle/SACM to model the Tokeneer development process originally followed by Praxis, and illustrated in Fig. 5b. A GSN diagram of the modular structure of the AC is shown in Fig. 6. The modules have names that correspond to the modelled artifact and a brief description. Each module contains a mixture of lifecycle and certification artifacts, such as requirements and models, and GSN arguments. The former were developed originally by Praxis and evaluated to comply with CC EAL 5 in the context of the certification process of TIS.

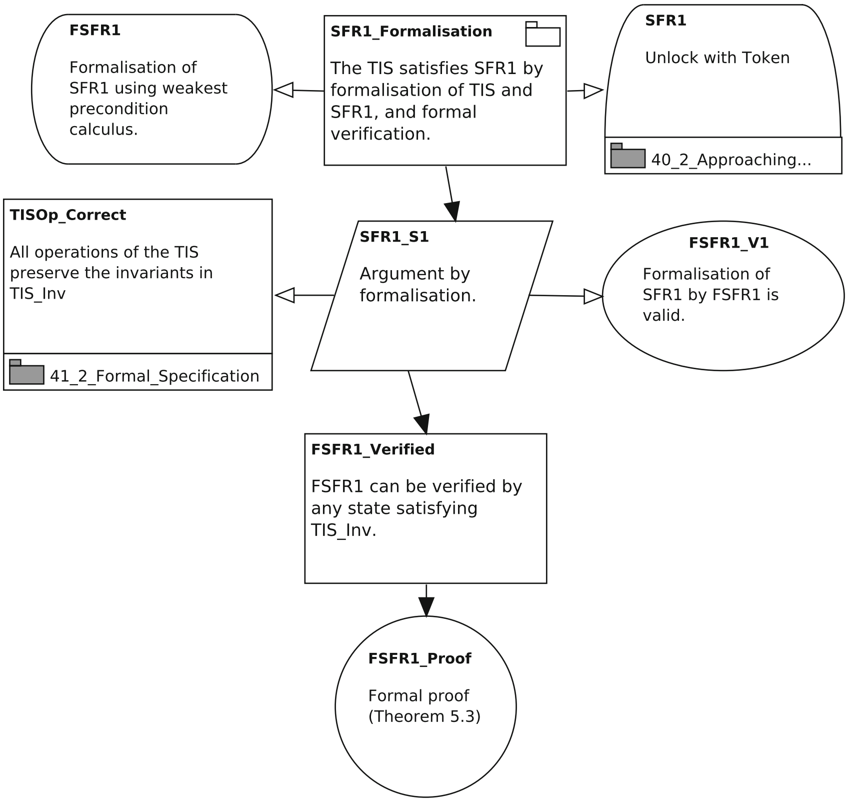

FSFR1
Formalisation of SFR1 using weakest precondition calculus.
SFR1_Formalisation
The TIS satisfies SFR1 by formalisation of TIS and SFR1, and formal verification.
SFR1
Unlock with Token
40_2_Approaching...
TISOp_Correct
All operations of the TIS preserve the invariants in TIS_Inv
41_2_Formal_Specification
SFR1_S1
Argument by formalisation.
FSFR1_V1
Formalisation of SFR1 by FSFR1 is valid.
FSFR1_Verified
FSFR1 can be verified by any state satisfying TIS_Inv.
FSFR1_Proof
Formal proof (Theorem 5.3)

Fig. 13. An argument for satisfaction of SFR1 by formalisation

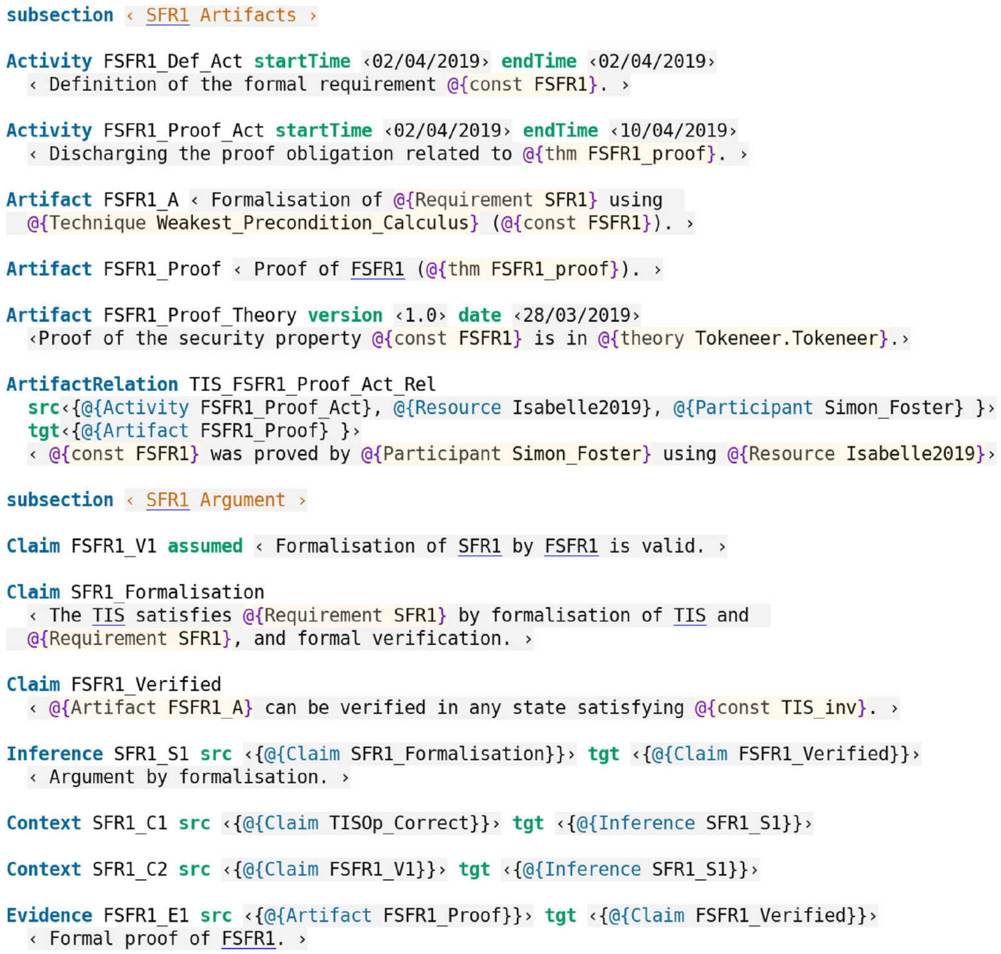
While the certification artifacts were the contribution of Praxis, the GSN AC argument modelling those artifacts is our contribution. Therefore, we complement the certification process of TIS with a GSN model translated to Isabelle/SACM, which aids the evaluation process of the certification artifacts by offering a machine-checked argument structure with full traceability. Our work also provides guidelines on the use of modular GSN to document the artifacts.

We focus on the module TIS_SFRs, illustrated in Fig. 12. It encapsulates an argument for a public claim that all SFRs are satisfied, which are defined in the module 40_2, by the TIS model, which is defined in 41_2. We reference these artifacts with the use of *away context* elements. For now, we focus on the SFRs that we have formally verified, namely SFR1, SFR3, and SFR6. For this, we use the Theorems 5.2, 5.4, 5.5, and 5.6 from § 5 as evidential artifacts. Satisfaction of SFR1 is modelled by the claim SFR1_C1, which uses SFR1 as context. The claim is satisfied by formalisation, which is performed in another module called TIS_SFR1. Satisfaction of the other SFRs can be represented using the same pattern.

The argument for SFR1 is shown in Fig. 13. It uses the "formalisation pattern" [DP18], which shows how results from a formal method can be used to provide evidence for claims to satisfy a requirement {R} for a system element {S}. The strategy used to decompose the claim "Requirement {R} is met by {S}" is contingent on the validation of both the formal specification of {R} and the formal model of {S}. Consequently, the pattern breaks down the satisfaction of {R} into three claims stating that (1) the formal model of {S} is validated, (2) the formalisation of {R} correctly characterises {R}, and (3) the formal model of {S} satisfies the formalisation of {R}. The former two claims are usually satisfied by manual review, since they are not amenable to formalisation, as the DO-178C formal methods supplement also makes clear:

> *"Formal methods cannot show that derived requirements and the reason for their existence are correctly defined; this should be achieved by review."* [J+11]

In Fig. 13, we adapt Denney's pattern [DP18] as follows. We begin with the claim SFR1_Formalisation, which references both SFR1, with its natural language description from module 40_1, and FSFR1, which is defined in this module and is defined in Definition 5.8 from § 5.



```
subsection ‹ SFR1 Artifacts ›

Activity FSFR1_Def_Act startTime ‹02/04/2019› endTime ‹02/04/2019›
  ‹ Definition of the formal requirement @{const FSFR1}. ›

Activity FSFR1_Proof_Act startTime ‹02/04/2019› endTime ‹10/04/2019›
  ‹ Discharging the proof obligation related to @{thm FSFR1_proof}. ›

Artifact FSFR1_A ‹ Formalisation of @{Requirement SFR1} using
  @{Technique Weakest_Precondition_Calculus} (@{const FSFR1}). ›

Artifact FSFR1_Proof ‹ Proof of FSFR1 (@{thm FSFR1_proof}). ›

Artifact FSFR1_Proof_Theory version ‹1.0› date ‹28/03/2019›
  ‹Proof of the security property @{const FSFR1} is in @{theory Tokeneer.Tokeneer}.›

ArtifactRelation TIS_FSFR1_Proof_Act_Rel
  src‹{@{Activity FSFR1_Proof_Act}, @{Resource Isabelle2019}, @{Participant Simon_Foster} }›
  tgt‹{@{Artifact FSFR1_Proof} }›
  ‹ @{const FSFR1} was proved by @{Participant Simon_Foster} using @{Resource Isabelle2019}›

subsection ‹ SFR1 Argument ›

Claim FSFR1_V1 assumed ‹ Formalisation of SFR1 by FSFR1 is valid. ›

Claim SFR1_Formalisation
  ‹ The TIS satisfies @{Requirement SFR1} by formalisation of TIS and
  @{Requirement SFR1}, and formal verification. ›

Claim FSFR1_Verified
  ‹ @{Artifact FSFR1_A} can be verified in any state satisfying @{const TIS_inv}. ›

Inference SFR1_S1 src ‹{@{Claim SFR1_Formalisation}}› tgt ‹{@{Claim FSFR1_Verified}}›
  ‹ Argument by formalisation. ›

Context SFR1_C1 src ‹{@{Claim TISOp_Correct}}› tgt ‹{@{Inference SFR1_S1}}›

Context SFR1_C2 src ‹{@{Claim FSFR1_V1}}› tgt ‹{@{Inference SFR1_S1}}›

Evidence FSFR1_E1 src ‹{@{Artifact FSFR1_Proof}}› tgt ‹{@{Claim FSFR1_Verified}}›
  ‹ Formal proof of FSFR1. ›
```

Fig. 14. An argument and associated artifacts for FSFR1 in IAL

We then invoke an argumentation strategy, SFR1_S1, for formalisation. Instead of using a validation claim for the formalization of the requirements, we use a justification element, FSFR1_V1, which should be an explanation of how FSFR1 formalises SFR1. This is to preserve the well-formedness of the AC – the "requirement validation" claims have a type different from the "requirement satisfaction" claims. An example of a "requirement satisfaction" claim is SFR1_Formalisation.

FSFR1 also assumes that all the operations of the TIS preserve the system invariants, and so we record a link to this proof in module 41_2, which corresponds to Theorem 5.2. The subclaim of SFR1_S1 is FSFR1_Verified, which is supported by the evidence FSFR1-Proof, which refers to Theorem 5.4.

Figure 14 shows the IAL model of TIS_SFR1 that was manually translated and elaborated from Fig. 13. In our translation, each of the modules in Fig. 6 is assigned an Isabelle theory with the corresponding artifacts. We represent both the artifacts and argumentation elements necessary to assure satisfaction of SFR1. Each command has an optional descriptive text, enclosed in quotes ‹. . .› that can integrate hyperlinks to both formal artifacts, such as theorems and proof, and structured assurance artifacts, such as model elements generated by IAL. Since the checks performed by IAL are successful, no errors are issued in Fig. 14, which in particular indicates that every referenced artifact exists and is correctly typed.

SACM provides several additional concepts for representing lifecycle artifacts, and we utilise them here. We record two activities, FSFR1_Def_Act and FSFR1_Proof_Act, which represent the activities in the development workflow for defining the formal requirement and discharging the proof obligations. Both have a `startTime` and `endTime` associated. FSFR1 is represented by the artifact FSFR1_A, which links to the

IAL requirement SFR1, which contains the natural language description of the requirement SFR1, using the `Requirement` antiquotation, and the technique Weakest_Precondition_Calculus [Dij75]. FSFR1_A also contains a link to the corresponding formal Isabelle constant via the antiquotation `@{const FSFR1}`.

We record a link to the proof of FSFR1 in the artifact FSFR1_Proof, and the Isabelle theory where this resides in FSFR1_Proof_Theory. Finally, we create an artifact relation that gives the provenance for the proof of FSFR1. This proof was performed by the participant Simon_Foster, during the activity FSFR1_Proof_Act, using the theorem prover Isabelle2019. In this way we record precisely how and when a particular assurance artifact was created.

With the artifacts and their provenance defined, we move on to the argumentation. We first create the key claims using the `Claim` command, which variously reference the artifacts previously defined. Claim FSFR1_V1 is marked as `assumed`, since this is the validation claim that must be satisfied elsewhere by review. The strategy SFR1_S1 from Fig. 13, is modelled by `SFR1_S1` in Fig. 14. `SFR1_S1` is created using the command `Inference`, which uses antiquotations to refer to the premise claims `SFR1_Formalisation`, `TISOp_Correct`, and `FSFR1_V1`, that is, the source `src`, and the conclusion claim `FSFR1_Verified`, that is the target `tgt`.

We use the `Context` command to model the two contextual relations in Fig. 13. SFR1_C1 presents the external claim TISOp_Correct as context, which refers to the invariant proof (Theorem 5.2), and SFR1_C2 presents the assumed validation claim as context. Finally, we model the relationships from Fig. 13 that link FSFR1_Verified to FSFR1_Proof. This is done in Fig. 14 by `FSFR1_E1`, which is created using the command `Evidence`. It supports the claim `FSFR1_Verified` with the artifact `FSFR1_Proof`.

We have shown how Isabelle/SACM enables the integration of formal development with assurance argumentation, documenting how the evidence collected establishes the overall security claims. In the next two sections we survey related work and discuss the findings of our case study.

7. Related work

In this section, we discuss previous efforts in the verification of Tokeneer as well as other approaches to the formalisation of assurance cases and the integration of formal methods with assurance cases.

Comparison with Previous Work in the Verification of Tokeneer Woodcock et al. [WAC10] highlight defects of the Tokeneer SPARK implementation, indicate undischarged verification conditions, and perform robustness tests generated by the Alloy SAT solver [Jac00] from a corresponding Alloy model. Using De Bono's lateral thinking, these test cases go beyond the anticipated operational envelope and stimulate anomalous behaviours. In shortening the feedback cycle for verification and test engineers, theorem proving in the form of the proposed framework can help using this approach more intensively.

Abdelhalim et al. [ASST10] model part of the Tokeneer specification using UML activity diagrams translated to be checked for deadlock freedom by the CSP model checker FDR. Their formalisation is assumed to be implemented on top of asynchronous communication, modelled in CSP in terms of buffers for each channel between UML components. While our abstraction from such communication aspects yields a simpler proof of the SFRs in § 3, their deadlock checking at a lower level can be useful for checking the correctness of the communication in an implementation of the UML model such as the SPARK implementation mentioned in Fig. 5b. Their UML diagrams can lead to comparatively large specifications whereas our formalisation stays compact thanks to the abstraction and reuse mechanisms in Z schemas and Isabelle/UTP.

Rivera et al. [RBC16] present an Event-B model of the TIS, 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, and aim to detect errors in the Event-B model caused by misunderstandings of the specification. Using Rodin, the authors state that they verify the SFRs (§ 3) using Hoare triples. Our work uses a similar abstract machine specification, but with weakest precondition calculus as the main tool for verifying the SFRs. Beyond the replication of the Tokeneer case study, Rivera et al. [RBC16] deal with the relationship between the model and the code via testing, whereas we focus on the construction of certifiable assurance arguments from formal model-based specifications. Nevertheless, we believe Isabelle's code generation features could be applied in a similar way.

Previous Work on Formal Assurance and Formalised Assurance Cases In concordance with Woodcock et al.'s [WAC10] observations, several researchers have investigated ways of introducing formality into assurance cases [CHOS13, Rus14, DP18, DMW+18]. We highlight some of these approaches below.

AdvoCATE is a graphical tool for the construction of GSN-based safety cases [DP18]. It uses a formal foundation called argument structures, which prescribe well-formedness checks for the syntactic structure of (i.e. the graph underlying) an AC, and allow instantiation of assurance case patterns. Our work likewise ensures well-formedness, but also allows the embedding of content with formal semantics. Denney and Pai's formalisation pattern [DP18] is an inspiration for our work. Our framework is to be used as an assurance backend, which complements AdvoCATE with a deep integration of modelling and specification formalisms.

Rushby [Rus14] illustrates how assurance arguments can be formalized with modern verification systems such as Isabelle or PVS to overcome some of the logical fallacies associated with informal ACs. Similarly, our framework allows reasoning using formal logic, but additionally supports the combination of formal and informal artifacts. We drew inspiration from the work on the Evidential Tool Bus [CHOS13], which enables the combination of evidence from several formal and semi-formal analysis tools. In a very similar way, Isabelle supports the integration of a variety of formal analysis tools [WW07].

Diskin et al. [DMW+18] tackle the problem of hierarchical and modular assurance by using a formal model (in this case, a compositional data-flow model) of the system to be assured as the basis for generating evidence required for a particular assurance claim. Their framework is elaborate and practically relevant in the sense that it integrates well with the practice of model-based development. The paradigm of our approach is similar to theirs except that we use mechanised algebraic reasoning techniques, provide computer-assistance for the proposed reasoning steps, integrating informal assurance evidence.

Overall, 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. We have also recently applied our techniques to collision avoidance for autonomous ground robots [GFN19] and an autonomous underwater vehicle [FNO+20]; both of which are more recent benchmark examples.

8. Evaluation and discussion

Below, we summarise several observations and findings from our investigation.

Isabelle as an Assurance Platform The Isabelle architecture brings many benefits that we harnessed in our formal development. Though not a large model, the Tokeneer functional specification is by no means trivial. It consists of over 3000 lines of Isabelle code, many of which are calls to automated proof tactics that invoke compound proof procedures. Isabelle's Prover IDE manages the processing of formal artifacts asynchronously, with verification tasks being executed in the background, and where possible in parallel. Incremental changes to a proof document automatically trigger the rerunning of relevant proof tasks, with feedback provided to the user when proofs fail. We found this very useful for managing the development, where invariant checking could be rerun whenever it was necessary to modify a TIS operation specification, for example if we discovered a bug in a definition. These facilities could be inserted into a continuous integration platform for larger projects.

Moreover, Isabelle's support for informal structured mark-up in the document model allows readable justifications to have an integral part in a development. These features make Isabelle amenable to AC development and maintenance, though more work is required to make it accessible to non-experts. This could include the ability to drive development using a graphical tool like ACME [WKD+19], or possibly by providing a more word processor-like front end for non-experts. Nevertheless, we believe that both safety engineering and formal methods experts will always be required in such a development process. The critical question is whether the different actors can effectively communicate. We believe that Isabelle-based assurance cases can function as a central point of reference in the system lifecycle.

Proof Automation Our approach, as usual, reduces verification to finding invariants of the system under development. Consequently, knowledge of both invariant formulae and Hoare logic is required to verify the system. The majority of the invariant proofs are automated by the `hoare_auto` and `hoare_wlp_auto` tactics with minimal user interaction. For example, to prove the invariants in *IDStation* for the eight top-level user entry operations in Definition 5.6, we first split the goal into the well-formedness properties and state invariants, and then discharge the resulting goals using `hoare_auto`. In this case, there are no residual proof obligations that require manual intervention. For efficiency purposes, it is sometimes necessary that we prove each of the invariants separately, which requires some manual strengthening and weakening.

The main exception to this level of automation is the proof of the invariants for the enrolment operations (*TISEnrolOp*). Here, there is an operation called *ValidateEnrolmentDataOK*, which completes the enrolment procedure by updating the key store from a floppy disk using an operation called *UpdateKeyStore*. This operation establishes the validity of the key store, for example by ensuring that every key has been properly issued. The invariant proof is not fully automated, but produces six proof obligations that require reasoning about relations and set theory. Nevertheless, these can be solved relatively straightforwardly using a few deduction rules and automated theorem proving with *sledgehammer* [BBN11].

Proofs of SFR3 and SFR6 are both fully automatic. For SFR1, proof of the unlocking condition is fully automated using ***wp*** calculus and relational calculus. The completion of the proof requires, again, some weakening and strengthening, but this is mainly for efficiency reasons. It seems likely that manual interaction cannot fully be avoided, and therefore having a well-documented library of deduction rules for proof engineers is important. Nevertheless, the level of automation for verifying Tokeneer is very high, largely thanks to the highly developed library of theories and lemmas in Isabelle/HOL.

Tokeneer Assurance Case Despite its age, we see Tokeneer as a highly relevant benchmark specification, particularly since it is one of the grand challenges of the "Verified Software Initiative" [Woo06]. As we have argued elsewhere [GFW19], such benchmarks allow us to conduct objective analyses of assurance techniques to aid in their transfer to other domains. The issues highlighted in [WAC10] are systematic design problems that can be fixed by a change of the benchmark (e.g. by a two-way biometric identification on both sides of the enclave entrance). However, this is out of the scope of our work and does not harm Tokeneer in its function as a benchmark.

During the translation from Z into Isabelle/UTP's GCL and the formalisation of the SFRs, we identified some deficiencies in the way that the security requirements were originally proven. SFR4 and SFR5, were neither formalised nor proved. Moreover, as we have previously mentioned, the developers acknowledge that there are missing invariants necessary to support the proofs:

> *We have not [added the invariants], as we believe it will add little to the assurance of correctness, and is very time consuming. At higher levels of the CC assurance we would be required to carry out more formal proofs, in which case these modifications would be done.* [C$^+$08b, page 11]

One of the reasons we can now do this is because automation of formal proof has vastly improved since the development of Tokeneer. Consequently, we can reach these higher assurance levels with our mechanisation.

A further issue is that we could not prove SFR2 while staying faithful to its proposed formalisation in the benchmark artifacts [C$^+$08b, page 6]. This property states that, at the point of unlocking the door, the time must be "close to being within the permitted entry period". Like SFR1, it uses the operation $\mathit{TISOp} \fatsemi \mathit{TISUpdate}$ as the target for the verification. However, this operation does not allow *currentTime*, which internally records the time, to advance. This is because the variable *currentTime* is introduced by the schema *DoorLatchAlarm*, and all internal operations have $\Xi \mathit{DoorLatchAlarm}$, which implies that $\mathit{currentTime}' = \mathit{currentTime}$.

The advance of internal time occurs only when polling *currentTime* from the corresponding monitored variable *now*, using *TISPoll*, which can advance arbitrarily. Consequently, if *TISPoll* is not included, any invariant of time can be trivially satisfied, because time is constant. This seems to indicate that the security property has not been formalised correctly. A fix for the issue would require us to reason about *TISPoll*, and thus a more substantial proof. However, a reformulation of the requirement would require further interaction with the developers, and this is out scope. Nevertheless, this does illustrate the benefit of formal proof, since it highlights issues that cannot easily be spotted in an informal proof. If integrated into the system development lifecycle, it would allow feedback to other teams, and thus more robust outputs.

Formal Design and Refinement As shown in Fig. 5b on page 8, a complete assurance case of the TIS development would require the coverage of all three refinement steps described in [C$^+$08c], the functional or abstract *formal specification*, the more concrete *formal design*, and the *SPARK implementation*. The formal design is a data and operation refinement of the abstract types used in the formal specification, replacing sets and functions with data structures with operational semantics. Such refinement proofs would require formal reasoning about the memory models of the formal design and the SPARK implementation in Isabelle/UTP.

This reasoning can be based on separation logic like, for example, implemented in the Isabelle data refinement library [Lam17].

Wider Applicability Although Tokeneer is an example from the building security domain, we are confident that our approach is applicable to a wide variety of critical information and control systems. To justify our confidence, we can resort to seminal work by Lamport [Lam77] on the notion of a *safety property* as a property stating that "something will *not* happen". Here, "something" is to be understood as a substitute for "a bad thing". For example, in system safety, "bad thing" refers to critical events (e.g. valve failures) that can cause environmental damage or human injury. In building or infrastructural security, "bad thing" means critical events (e.g. misauthentication) that allow unauthorized/malicious access to secrets/controls, the latter potentially leading to damage or injury. Analogously, in IT security, "bad thing" signifies critical events (e.g. side-channel attacks) that allow such unauthorized or malicious access.

The inclined reader will immediately recognize that system safety and these two flavors of security are not only related in the form of their *requirements* but also in the nature of the consequences from violating these requirements. Returning to Tokeneer, those security properties that are expressed as invariants, are safety properties in Lamport's sense. Their verification/falsification boils down to the checking of whether all system runs/any system run fulfill/violates these invariants. In summary, the formal underpinning of our assurance approach can deal with any property that can be expressed as an invariant in the same way. It may certainly be an intricate issue to conceptually isolate invariants of appropriate strength, in other words, to express the "bad thing" as a meaningful safety property. This requires in-depth analysis, experience, and expertise.

9. Conclusions

We have presented Isabelle/SACM, a framework for integrating formal proof into a unified and standardised form of assurance cases and for their computer-assisted construction. We showed how SACM is embedded into Isabelle as an ontology, and provided an interactive assurance language that guides its user in generating valid instances of this ontology.

We applied this framework to part of the Tokeneer security case, including the verification of three of the security functional requirements, and embedded these results into a mechanised assurance argument. Isabelle/SACM enforces the usage of formal ontological links—a feature inherited from DOF—which establishes and enriches traceability between the assurance arguments, evidence of different provenance, and the assurance claims. Isabelle/SACM combines features from Isabelle/HOL, DOF, and SACM in a way that allows integration of formal methods and assurance cases [GFN19]. In summary, our work allows us to intertwine a formal development in Isabelle, potentially consisting of several heterogeneous notations and artifacts, with an assurance case that puts the formal results in context. We believe it also confirms the hypothesis that Isabelle is an ideal platform for systems assurance [BW19b], and as evidenced by projects like seL4 [KEH+09] and PikeOS [VHS+15], it can scale to verifications of a substantial size.

In future work, we will formalise the connection between ACME [WKD+19] and Isabelle/SACM, which will make the platform more accessible to safety practitioners. We are currently working on a prototype model-to-text transformation from SACM to Isabelle to facilitate this, and an integration with Eclipse to allow feedback from Isabelle to be propagated back to the diagram editors. In this way, the structure of an assurance case and its arguments can be created using a graphical editor, and formal artifacts from Isabelle can be presented as evidence. We are currently working with our industrial partners to explore how this technology can be inserted into their development processes, and the level of training required for employees. Moreover, we are also applying Isabelle/SACM to develop an assurance case for other case studies, including an autonomous underwater vehicle safety controller [FNO+20], which is being developed in partnership with D-RisQ[8] under the regime of the DO-178C standard. This work integrates a sophisticated graphical state machine and architectural language for modeling robotic systems, called RoboChart [MRL+19, FBC+18], and its associated Eclipsed-based tool, which will further aid accessibility.

From a safety engineering standpoint, we will also consider the integration of AC pattern execution [DP18], to facilitate AC production. Moreover, to support more advanced safety analysis, we are exploring the use of DOF to develop an ontology for safety concepts, such as hazards, risks, and control measures, following work by Banham [Ban20] and the Safety of Autonomous Systems Working Group [oASWG20], and also ontologies

[8] D-RisQ Software Systems. http://www.drisq.com/.

for formal methods. Indeed, we envisage the development of a variety of ontologies that provide the necessary terminology to formulate requirements, document developments, and otherwise aid in communication. We also plan to complete the mechanisation of the TIS security case, including the overarching argument for how the formal evidence can satisfy the requirements of CC [Com17]. This will involve mechanising the remaining operators, and tackling the last three security requirements, which will require derivation and verification of additional invariants. Moreover, formal provenance between the GCL operations and corresponding Z schemas could in the future be established with suitable equivalence proofs.

In parallel, we are developing our verification framework, Isabelle/UTP [FBC+20, FZW16, FZN+19] to support a variety of software engineering notations. We recently demonstrated formal verification facilities for a StateChart-like notation [FBC+18, FCC+19], and are also working towards tools to support hybrid dynamical languages [FTCW16, Fos19, MSF20] like Modelica and MATLAB Simulink. We have recently applied this to verification of an autonomous marine vehicle [FGC20]. Though these kinds of models seem quite different to Tokeneer, in the UTP they all have a relational semantics and so the development of proof facilities here feed into these works.

A recent survey of practitioners in critical software and systems engineering [GM20] indicates a willingness to apply formal methods, provided that the challenges of tool support and training can be addressed. Thus, our long-term overarching goal is a comprehensive assurance framework supported by a variety of integrated FMs, in order to support complex certification tasks for cyber-physical systems such as autonomous robots [GFW19, GFN19, FNO+20].

Acknowledgements

This work is funded by EPSRC projects CyPhyAssure (CyPhyAssure Project: https://www.cs.york.ac.uk/circus/CyPhyAssure/) (grant reference EP/S001190/1) and RoboCalc (grant reference EP/M025756/1), the German Science Foundation (DFG; grant 381212925), and the Assuring Autonomy International Programme (AAIP; grant CSI:Cobot).

References

[ASST10] Abdelhalim I, Sharp J, Schneider S, Treharne H (2010) Formal verification of Tokeneer behaviours modelled in fUML using CSP. In: Formal methods and software engineering, volume 6447 of LNCS. Springer, pp 371–387

[BACW18] Brucker AD, Aït-Sadoune I, Crisafulli P, Wolff B (2018) Using the Isabelle Ontology Framework—linking the formal with the informal. In: Proceedings of the 11th international conference on intelligent computer mathematics (CICM), volume 11006 of LNCS. Springer, pp 23–38

[Ban20] Banham D (2020) Formalising the language of risk. Saf Syst 28(1), February 2020

[BB98] Bishop PG, Bloomfield RE (1998) A methodology for safety case development. In: Redmill F, Anderson T (eds) Industrial perspectives of safety-critical systems: proceedings of 6th safety-critical systems symposium. Springer, pp 194–204

[BBN11] Blanchette JC, Bulwahn L, Nipkow T (2011) Automatic proof and disproof in Isabelle/HOL. In: Proceedings of 8th international symposium on frontiers of combining systems (FroCoS), volume 6989 of LNCS. Springer, pp 12–27

[BCJ+06] 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 (ISSSE)

[Bet16] Bettini L (2016) Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd

[BK13] Barmpis K, Kolovos D (2019) Hawk: towards a scalable model indexing architecture. In: Proceedings of the workshop on scalability in model driven engineering, pp 1–9

[BW19a] Brucker A, Wolff B (2019) Isabelle/DOF: Design and implementation. In: Proceedings of 17th international conference on software engineering and formal methods (SEFM), LNCS 11724. Springer, pp 279–292

[BW19b] Brucker A, Wolff B (2019) Using ontologies in formal developments targeting certification. In: Integrated formal methods (iFM), volume 11918 of LNCS. Springer, pp 65–82

[C+08a] Cooper D et al Tokeneer ID station: formal specification. Technical report, Praxis High Integrity Systems, August 2008. https://www.adacore.com/tokeneer

[C+08b] Cooper D et al Tokeneer ID station: security properties. Technical report, Praxis high integrity systems, August 2008. https://www.adacore.com/tokeneer

[C+08c] Cooper D et al Tokeneer ID station: summary report. Technical report S.P1229.81.1, Praxis High integrity systems, August 2008. https://www.adacore.com/tokeneer

[CHOS13] Cruanes S, Hamon G, Owre S, Shankar N (2013) Tool integration with the evidential tool bus. In: Proceedings of 14th interantional conference on verification, model checking, and abstract interpretation (VMCAI), volume 7737 of LNCS. Springer

[Com17] Common Criteria Consortium (2017). Common criteria for information technology security evaluation—part 1: Introduction and general model. Technical report CCMB-2017-04-001, Common Criteria Consortium, 2017

[CW06] Cavalcanti A, Woodcock J (2006) A tutorial introduction to CSP in unifying theories of programming. In: Refinement techniques in software engineering, volume 3167 of LNCS. Springer, pp 220–268

[Dij75] Dijkstra EW (1975) Guarded commands, nondeterminacy and formal derivation of programs. Commun ACM 18(8):453–457

[DMW+18] Diskin Z, Maibaum T, Wassyng A, Wynn-Williams S, Lawford M (2018) Assurance via model transformations and their hierarchical refinement. In: MODELS. IEEE

[DP13] Denney E, Pai G (2013) A formal basis for safety case patterns. In: Proceedings of 32nd international conference on computer safety, reliability, and security (SAFECOMP), volume 8153 of LNCS. Springer, pp 21–32

[DP15] Denney E, Pai G (2015) Towards a formal basis for modular safety cases. In: 34th inteenational confernece on computer safety, reliability, and security (SAFECOMP), volume 9337 of LNCS. Springer, pp 328–343

[DP18] Denney E, Pai G (2018) Tool support for assurance case development. Autom Softw Eng 25:435–499

[FB20] Foster S, Baxter J (2020) Automated algebraic reasoning for collections and local variables with lenses. In: Winter M (ed) Proceedings of 18th international conference on relational and algebraic methods in computer science (RAMiCS), volume 12062 of LNCS. Springer, April 2020

[FBC+18] Foster S, Baxter J, Cavalcanti A, Miyazawa A, Woodcock J (2018) Automating verification of state machines with reactive designs and Isabelle/UTP. In 15th Intl. Conf. on Formal Aspects of Component Software (FACS), volume 11222 of LNCS, pages 137–155. Springer, October 2018

[FBC+20] Foster S, Baxter J, Cavalcanti A, Woodcock J, Zeyda F (2020) Unifying semantic foundations for automated verification tools in Isabelle/UTP. Sci Comput Program 197, October 2020

[FCC+19] Foster S, Cavalcanti A, Canham S, Woodcock J, Zeyda F (2019) Unifying theories of reactive design contracts. Theor Comput Sci 802, September 2019

[FCWZ18] Foster S, Cavalcanti A, Woodcock J, Zeyda F (2018) Unifying theories of time with generalised reactive processes. Inf Process Lett 135:47–52

[FGC20] Foster S, Gleirscher M, Calinescu R (2020) Towards deductive verification of control algorithms for autonomous marine vehicles. In: 25th proceedings of international conference on engineering of complex computer systems (ICECCS). IEEE, October 2020

[FGM+07] Foster J, Greenwald M, Moore J, Pierce B, Schmitt A (2007) Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans Program Lang Syst 29(3), May 2007

[FNGK19] Foster S, Nemouchi Y, Gleirscher M, Kelly T (2019) Isabelle/SACM: Computer-assisted assurance cases with integrated formal methods. In: Proceedings of 15th international conference on integrated formal methods (iFM), volume 11918 of LNCS. Springer, pp 379–398, December 2019

[FNO+20] Foster S, Nemouchi Y, O'Halloran C, Tudor N, Stephenson K (2020) Formal model-based assurance cases in Isabelle/SACM: an autonomous underwater vehicle case study. In: Formal methods in software engineering (FormaliSE 2020): proceedings of the 8th international conference. ACM, 2020

[Fos19] Foster S (2019) Hybrid relations in Isabelle/UTP. In: UTP, volume 11885 of LNCS. Springer, pp 130–153

[FTCW16] Foster S, Thiele B, Cavalcanti A, Woodcock J (2016) Towards a UTP semantics for Modelica. In: UTP, LNCS 10134. Springer, pp 44–64

[FZN+19] Foster S, Zeyda F, Nemouchi Y, Ribeiro P, Wolff B (2019) Isabelle/UTP: mechanised theory engineering for unifying theories of programming. Archive of Formal Proofs, https://www.isa-afp.org/entries/UTP.html

[FZW16] Foster S, Zeyda F, Woodcock J (2016) Unifying heterogeneous state-spaces with lenses. In: Proceedings of 13th international colloquium on theoretical aspects of computing (ICTAC), LNCS 9965. Springer

[GC17] Gleirscher M, Carlan C (2017) Arguing from hazard analysis in safety cases: a modular argument pattern. In High assurance systems engineering (HASE), 18th international symposium, pp 53–60

[GFN19] Gleirscher M, Foster S, Nemouchi Y (2019) Evolution of formal model-based assurance cases for autonomous robots. In Proceedings 17th International Conference on Software Engineering and Formal Methods (SEFM), LNCS 11724. Springer

[GFW19] Gleirscher M, Foster S, Woodcock J (2019) New opportunities for integrated formal methods. ACM Comput. Surv 52(6). https://dl.acm.org/doi/10.1145/3357231

[GKHP06] Greenwell W, Knight J, Holloway CM, Pease J (2006) A taxonomy of fallacies in system safety arguments. In: Proceedings of 24th international system safety conference, pp 430–439, July 2006

[GM20] Gleirscher M, Marmsoler D (2020) Formal methods in dependable systems engineering: a survey of professionals from Europe and North America. Empir Softw Eng 25(6)

[HH98] Hoare CAR, He J (1998) Unifying theories of programming. Prentice-Hall

[HHK$^+$15] Hawkins R, Habli I, Kolovos D, Paige R, Kelly T (2015) Weaving and assurance case from design: A model-based approach. In Proceedings 16th international symposium on high assurance systems engineering. IEEE, pp 110–117

[HK14] Habli I, Kelly T (2014) Balancing the formal and informal in safety case arguments. In: VeriSure workshop, colocated with CAV, July 2014

[J$^+$11] Krodel J et al (2011) Formal methods supplement to DO-178C and DO-278A. RTCA, Inc.

[Jac00] Jackson D (2000) Alloy: a lightweight object modelling notation. ACM Trans Softw Eng Methodol, 11(2):256–290

[KEH$^+$09] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S (2009) seL4: formal verification of an OS kernel. In: Proceedings 22nd symposium on operating systems principles (SOSP), . ACM, pp 207–220

[Kel98] Kelly T (1998) Arguing safety—a systematic approach to safety case management. Ph.D. thesis, University of York

[KPP06] Kolovos D, Paige R, Polack F (2006) Eclipse development tools for Epsilon. In: Eclipse summit Europe, eclipse modeling symposium, vol 20062, p 200

[Lam77] Lamport L (1977) Proving the correctness of multiprocess programs. IEEE Trans Softw Eng 3(2):125–43

[Lam17] Lammich P (2017) Refinement to imperative HOL. J Autom Reason 62(4):481–503

[MRL$^+$19] Miyazawa A, Ribeiro P, Li W, Cavalcanti A, Timmis J, Woodcock J (2019) Robochart: modelling and verification of the functional behaviour of robotic applications. Softw Syst Modell 18, January 2019

[MSF20] Munive JHY, Struth G, Foster S (2020) Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL. In: 18th international conference on relational and algebraic methods in computer science (RAMiCS), volume 12062 of LNCS. Springer, pp 169–186, April 2020

[NPW02] Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL—a proof assistant for higher-order logic, volume 2283 of LNCS. Springer

[oASWG20] Safety of Autonomous Systems Working Group (2020) Safety assurance objectives for autonomous systems (version 2.0), volume SCS-153A. Safety Critical Systems Club, February 2020

[Obj20] Object Management Group (2020) Structured assurance case metamodel specification. https://www.omg.org/spec/SACM/, April 2020

[Pai97] Paige RF (1997) A meta-method for formal method integration. In: Formal methods Europe (FME), volume 1313 of LNCS. Springer, pp 473–494

[RBC16] Rivera V, Bhattacharya S, Cataño N (2016) Undertaking the Tokeneer challenge in Event-B. In: Proceedings of 4th international conference on formal methods in software engineering (FormaliSE). ACM Press

[Rus13] Rushby J (2013) Logic and epistemology in safety cases. In: Proceeidngs of 32nd international conference on computer safety, reliability, and security (SAFECOMP), volume 8153 of LNCS. Springer, pp 1–7

[Rus14] Rushby J (2014) Mechanized support for assurance case argumentation. In: New frontiers in artificial intelligence, volume 8417 of LNCS. Springer

[Spi89] Spivey M (1989) The Z-notation—a reference manual. Prentice Hall, Englewood Cliffs, N. J.

[VHS$^+$15] Verbeek F, Havle O, Schmaltz J, Tverdyshev S, Blasum H, Langenstein W, Stephan B, Wolff B, Nemouchi Y (2015) Formal API specification of the PikeOS separation kernel. In: Proceedings of 7th NASA formal methods symposium (NFM 2015), volume 9058 of LNCS. Springer, pp 375–389

[WAC10] Woodcock J, Aydal EA, Chapman R (2010) The Tokeneer experiments. In: Reflections on the work of C.A.R. Hoare. Springer, pp 405–430

[Wen18] Wenzel M (2018) Isabelle/jEdit as IDE for domain-specific formal languages and informal text documents. In: Proceedings of 4th workshop on formal integrated development environment (F-IDE), pp 71–84

[Wen19] Wenzel M (2019) Interaction with formal mathematical documents in Isabelle/PIDE. In Proceedings of 12th international confrence on intelligent computer mathematics (CICM), volume 11617 of LNCS. Springer, pp 1–15

[WKD$^+$19] Wei R, Kelly T, Dai X, Zhao S, Hawkins R (2019) Model based system assurance using the structured assurance case metamodel. Syst Softw 154

[Woo06] Woodcock J (2006) First steps in the verified software grand challenge. IEEE Comput 39(10)

[WW07] Wenzel M, Wolff B (2007) Building formal method tools in the Isabelle/Isar framework. In Proceedings of the 20th international conference on theorem proving in higher order logics (TPHOLs), volume 4732 of LNCS. Springer, pp 352–367

Received 23 July 2020

Accepted in revised form 30 January 2021 by Lizeth Tarifa, Wolfgang Ahrendt and Heike Warheim