Isabelle/SACM: Computer-Assisted Assurance Cases with Integrated Formal Methods

Yakoub Nemouchi, Simon Foster[(✉)], Mario Gleirscher, and Tim Kelly

University of York, York, UK
{yakoub.nemouchi,simon.foster,mario.gleirscher,tim.kelly}@york.ac.uk

Abstract. Assurance cases (ACs) are often required to certify critical systems. The use of integrated formal methods (FMs) in assurance can improve automation, increase confidence, and overcome errant reasoning. However, ACs can rarely be fully formalised, as the use of FMs 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 for the computer-assisted construction of ACs called Isabelle/SACM. The framework guarantees well-formedness, consistency, and traceability of ACs, and allows a tight integration of formal and informal evidence of various provenance. To validate Isabelle/SACM, we present a novel formalisation of the Tokeneer benchmark, verify its security requirements, and form a mechanised AC that combines the resulting formal and informal artifacts.

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 [24,30]. They are recommended by several international standards, such as ISO26262 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 languages exist, including the Goal Structuring Notation (GSN) [24], Claims, Arguments, and Evidence (CAE) [2], and the Structured Assurance Case Metamodel (SACM)[1] [30], a standard that unifies several notations.

AC creation can be supported by model-based design, which utilises architectural and behavioural models over which requirements can be formulated [30]. However, ACs can suffer from logical fallacies and inadequate evidence [20]. A proposed solution is formalisation in a machine-checked logic to enable verification of consistency and well-foundedness [28]. As confirmed by avionics standard

---

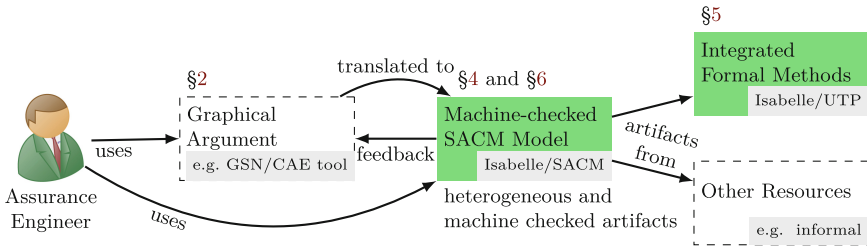[1] *OMG Structured Assurance Case Metamodel*: http://www.omg.org/spec/SACM/.

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

DO-178C, the evidence gathering process can also benefit from the rigour of formal methods (FMs). However, it is also the case that (1) ACs are intended primarily for human consumption, and (2) that formal models must be validated informally [21]. Consequently, ACs usually combine informal and formal content, and so tools must support this. Moreover, there is a need to integrate several FMs [26], potentially with differing computational paradigms and levels of abstraction [22], and so it is necessary to manage the resulting heterogeneity [19].

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 [9,30] that utilise SACM as a unified interchange format, and an array of FM tools provided by our Isabelle-based verification platform, Isabelle/UTP [16,17]. Our framework aims to improve existing assurance processes by harnessing formal verification to produce mathematically grounded ACs with guarantees of consistency and adequacy of the evidence. In the context of safety regulation, it can help with AC evaluation through machine-checking and automated verification.

Contributions. A first step in this direction is made by the contributions of this paper, which are: (1) Isabelle/SACM, an implementation of SACM in Isabelle [25], (2) a front-end for Isabelle/SACM called interactive assurance language (IAL), which is an interactive DSL for the definition of machine-checked SACM models, (3) a novel formalisation of Tokeneer [1] in Isabelle/UTP, (4) the verification of the Tokeneer security requirements[2], and (5) the definition of an AC with the claims that Tokeneer meets its security requirements. Our Tokeneer AC demonstrates how to integrate formal artifacts, resulting from Isabelle/UTP (4), and informal artifacts, such as the Tokeneer documentation.

Isabelle provides a sophisticated executable document model for presenting a graph of hyperlinked formal artifacts, like definitions, theorems, and proofs. It provides automatic and incremental consistency checking, where updates to artifacts trigger rechecking. Such capabilities can support efficient maintenance

---

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

and evolution of model-based ACs [30]. Moreover, the document model allows both formal and informal content [32], and provides access to an array of automated proof tools [31,32]. Additionally, Brucker et al. [4] extend Isabelle with DOF, a framework with a textual language for embedding of meta-models into the Isabelle document model, which we harness to embed SACM.

Isabelle/UTP [16,17] harnesses Unifying Theories of Programming [22] (UTP) to provide formal verification facilities for a variety of languages, with paradigms as diverse as concurrency [13], real-time [14], and hybrid computation [15]. 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 semantics.

The paper is organised as follows. In Sect. 2 we outline preliminaries: SACM, Isabelle, and DOF. In Sect. 3 we describe the Tokeneer system. In Sect. 4 we begin our contributions by describing Isabelle/SACM, which consists of the embedding of SACM into DOF (Sect. 4.1), and IAL (Sect. 4.2). In Sect. 5 we model and verify Tokeneer in Isabelle/UTP. In Sect. 6 we describe the mechanisation of the Tokeneer AC in Isabelle/SACM. In Sect. 7 we highlight related work, and in Sect. 8 we conclude.

2 Preliminaries

SACM. ACs are often presented using a notation like GSN [24] (Fig. 2). Here, claims 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. SACM is an OMG standard metamodel for ACs [30]. It

Fig. 2. Goal Structuring Notation

unifies, extends, and refines several predecessor notations, including GSN [24] and CAE [2] (Claims, Arguments, and Evidence), and is intended as a definitive reference model.
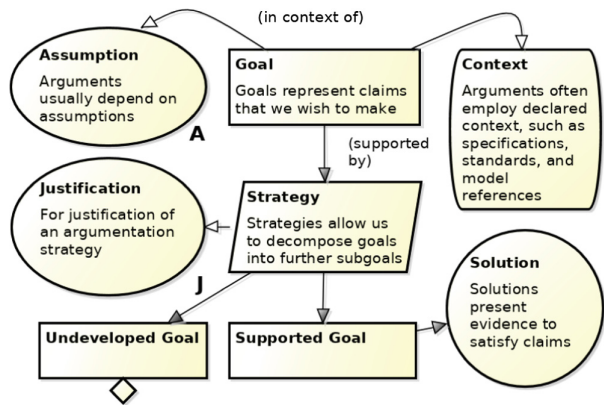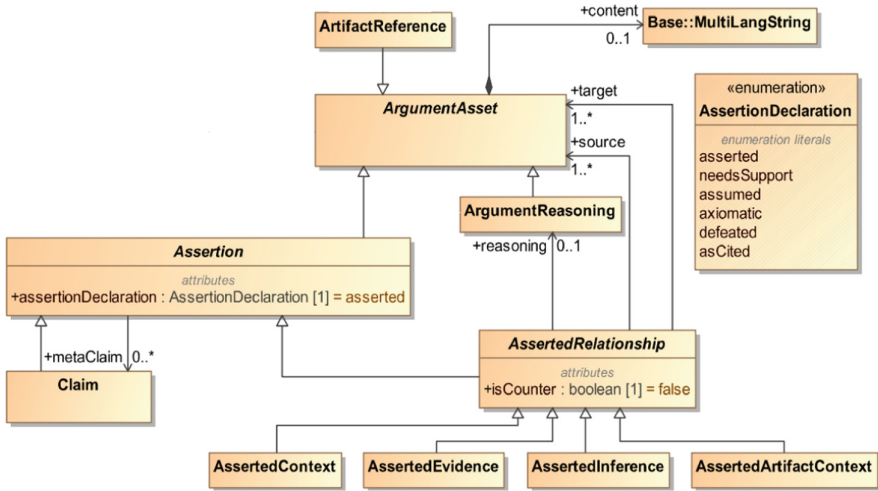
Fig. 3. SACM argumentation meta-model (See footnote 1)

SACM models three crucial concepts: arguments, artifacts, and terminology. An argument is a set of claims, evidence citations, and inferential links between them. 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 languages, but in SACM they can also contain structured expressions, which allows integration of formal languages.

The argumentation meta-model 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 AssertedArtifactSupport which documents an inferential dependency between the claims of two artifacts.

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

Isabelle. Isabelle/HOL is an interactive theorem prover for higher order logic (HOL) [25], based on the generic framework Isar [31]. The former provides a functional specification language, and an array of automated proof tools [3]. The latter has an interactive, extensible, and executable document model [32], which describes Isabelle theories. Plugins, such as Isabelle/HOL, DOF, Isabelle/UTP, and Isabelle/SACM have document models that contain conservative extensions to Isar.

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 predeclared keyword (in blue), such as the command `ML`, (2) a "semantics area" enclosed between `<...>`, and (3) optional subkeywords (in green). Commands generate formal document artifacts. 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 formal artifacts, 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 Sect. 4.2 we extend this document model with commands that define our assurance language, IAL.

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 formal artifact `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.

DOF. A foundation for our work is DOF and its Isabelle Ontology Specification Language (IOSL) [4]: a textual 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 sit at the meta-logical level, so they can be referenced using antiquotations, and carry an enriched version of Isabelle's markup string.
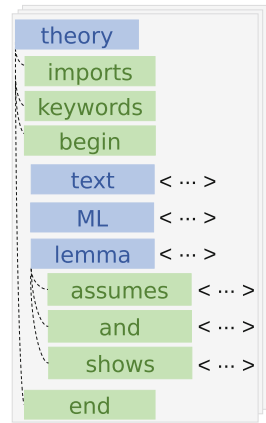
Fig. 4. Document model (Color figure online)

3   Case Study: Tokeneer

To demonstrate our approach, we use the 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. [1]. Barnes et al. performed security analysis, specification using Z, implementa-

Fig. 5. Tokeneer system overview

tion in SPARK, and verification of the security properties. After independent assessment, Common Criteria (CC) Evaluation Assurance Level (EAL) 5 was achieved. Tokeneer is therefore 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 formal methods and assurance techniques.

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 security of the TIS is assured by demonstrating six Security Functional Requirements (SFRs) [7], of which the first three are shown below:

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 role of "guard".

SFR2. If the latch is unlocked automatically by TIS, then the current time must be close to being within the allowed entry period defined for the User requesting access.

SFR3. An alarm will be raised whenever the door/latch is insecure.

Our objective is to construct a machine-checked assurance case that argues that the TIS fulfils these security properties, and integrate evidential artifacts from our mechanised model of the TIS behaviour in Isabelle/UTP.
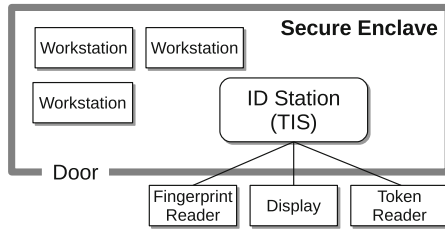
---

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

4   Isabelle/SACM

Here, we encode SACM as a DOF ontology (Sect. 4.1), and use it to provide an
interactive machine-checked AC language (Sect. 4.2). Our embedding implements
ACs as meta-logical entities in Isabelle, 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 mod-
elling `ArgumentAsset`[4] and its child classes from Fig. 3, as these are the most
relevant classes for the TIS assurance argument that we develop in Sect. 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 antiquota-
tion type, `@{ArgumentAsset <...>}`, which can be used to refer to entities of this
type. `ArgumentationElement` is a class which `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 expli-
cate 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:

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

Here, `datatype` defines a HOL enumeration type, `assertionDeclarations_t` is the
defined enumeration type, `set` is the set type, and `option` is the optional type.
Attribute `assertionDeclaration` is of type `assertionDeclarations_t`, which spec-
ifies the status of instances of type `Assertion`. Examples of `Assertions` in SACM
are claims, justifications, and both kinds of arrows from Fig. 2. The attribute

---

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

structure_assoc is an association to the class `ArgumentPackage`, which is not discussed here. Finally, the attribute `referencedArtifactElement_assoc` is an association to `ArtifactElements` from the `ArtifactPackage`, allowing instances of type `ArgumentAsset` to be supported by evidential artifacts.

The class `Claim` from Fig. 3 inherits from the class `Assertion`. This means that `Claim` inherits the attributes `gid`, `content_assoc`, and `assertionDeclaration` of type `assertion Declarations_t`. The other child class of `Assertion` is:

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

This 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 child classes of `AssertedRelationship` also have the attributes `source` and `target`, both of type `ArgumentAsset`.

4.2   Interactive Assurance Language (IAL)

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

IAL instantiates `doc_class`es from Sect. 4.1 to create SACM models in Isabelle, for example, the command `CLAIM` creates an instance of the class `Claim`. Attributes and associations of a class have a concrete syntax represented by an Isabelle (green) subcommand. For example, the association `content_assoc::` `MultiLangString` is represented by `CONTENT <...>`; where `<...>` is DOF's markup string. A selection of IAL commands is given below.

```
CLAIM isABS isCITE ASSERTED <gid> CONTENT <MultiLangString>
ASSERTED_INFERENCE <gid> SOURCE <gid>* TARGET <gid>*
ASSERTED_CONTEXT <gid> SOURCE <gid>* TARGET <gid>*
ASSERTED_EVIDENCE <gid> SOURCE <gid>* TARGET <gid>*
```

`CLAIM` creates an instance of type `Claim` with an identifier (`gid`), and content described by a `MultiLangString`. The antiquotation `@{Claim <<gid>>}` can be used to reference the created instance. The subcommands `isABS`, `isCITE` and `ASSERTED` are optional. `ASSERTED_INFERENCE` creates an inference between several instances of type `ArgumentAsset`. It has subcommands `SOURCE` and `TARGET` that are both lists of antiquotations pointing to `ArgumentAssets`. The use of antiquotations to reference the instances ensures that Isabelle will do the checks explained in Sect. 2. `ASSERTED_CONTEXT` similarly asserts that an instance should be treated as context for another, and `ASSERTED_EVIDENCE` associates evidence with a claim. All

instances created by IAL are *semi-formal*, since they can contain both informal content and references to formal content that are machine checked.

Figure 6 shows the interactive nature of IAL. It represents an inferential link between the semi-formal artifacts `Claim_A` and `Claim_C`. The semi-formal artifact `Rel_A`, which is the inferential link between `Claim_A` and `Claim_C`, is created via the command `ASSERTED_INFERENCE`. However, `Claim_C` does not exist, and so the error message at the top is issued. The command `text` is then used to reference `Rel_A` using the antiquotation `@{AssertedInference Rel_A}`. This also leads to an error, shown at the bottom, since `Rel_A` was not introduced to the context of the document model, due to the error at the top.
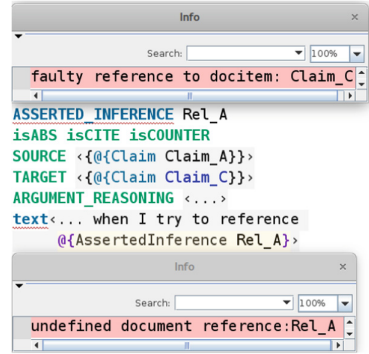

Fig. 6. Interactive DSL

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

Here, we present a novel mechanisation of Tokeneer in Isabelle/UTP [16,17] to provide evidence for the AC. In [7], the SFRs are argued semi-formally, but here we provide a formal proof. We focus on the verification of SFR1[5], the most challenging of the six SFRs, and describe the necessary model elements.

5.1   Modelling and Mechanisation

The TIS specification [6] describes an elaborate state space and a collection of relational operations. The state is bipartite, consisting of (1) the digital state 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 state transition system, illustrated in Fig. 7 (cf. [6, page 43]), where each transition corresponds to an operation. Several operations are omitted due to space constraints. Following enrolment, the TIS becomes quiescent (awaiting interaction). ReadUserToken triggers if the token is presented, and reads its contents. Assuming a valid token, the TIS determines whether a fingerprint is necessary, and then triggers either BioCheckRequired or BioCheckNotRequired. If required, the TIS then reads a fingerprint (ReadFingerOK), validates it (ValidateFingerOK), and finally writes an authorisation certificate to the token (WriteUserTokenOK). If the access credentials are available (waitingEntry), then a final check is performed (EntryOK), and once the user removes their token (waitingRemoveTokenSuccess), the door is unlocked (UnlockDoor).

---

[5] The administrator (role "guard") part is verified but omitted for space reasons.
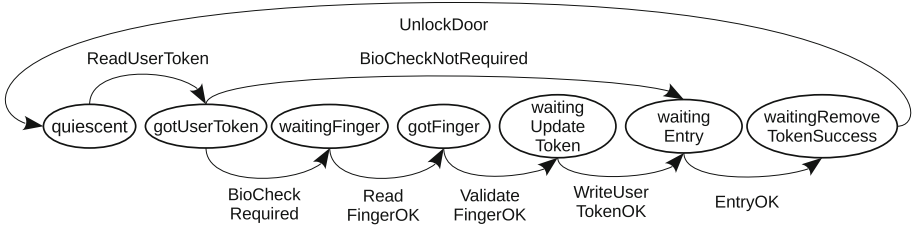
Fig. 7. TIS main states

We mechanise the TIS using hierarchical state space types, with invariants adapted from the Z specification [6]. We define the operations using guarded command language [10] (GCL) rather than the Z schemas directly, to enable syntax-directed reasoning. GCL has a denotational semantics in UTP's relational calculus [22], so that it is possible to prove equivalence with the corresponding Z operations. We use a GCL variant that follows the following syntax:

$$\mathcal{P} ::= \textbf{skip} \mid \textbf{abort} \mid \mathcal{P} \, \raisebox{0.2ex}{\scriptsize\textbf{;}} \, \mathcal{P} \mid \mathcal{E} \longrightarrow \mathcal{P} \mid \mathcal{P} \sqcap \mathcal{P} \mid \mathcal{V} := \mathcal{E} \mid \mathcal{V} {:} [\mathcal{P}]$$

Here, $\mathcal{P}$ is a program, $\mathcal{E}$ is an expression, and $\mathcal{V}$ is a variable. The language provides sequential composition, guarded commands, non-deterministic choice, and assignment. We adopt a frame operator $a{:}[P]$, which states that P changes only variables in the namespace a [16,17]. This enables modular reasoning about the TIS internal and real-world states, which is a further novelty of our work. *State Types.* We first describe the state space of the TIS state machine:

$$IDStation \triangleq \begin{bmatrix} currentUserToken : TOKENTRY, currentTime : TIME, \\ userTokenPresence : PRESENCE, status : STATUS, \\ issuerKey : USER \nrightarrow KEYPART, \cdots \end{bmatrix}$$

$$Controlled \triangleq \begin{bmatrix} latch : LATCH, alarm : ALARM, \cdots \end{bmatrix}$$

$$Monitored \triangleq \begin{bmatrix} now : TIME, finger : FINGERPRINTTRY, \\ userToken : TOKENTRY, \cdots \end{bmatrix}$$

$$RealWorld \triangleq [mon : Monitored, ctrl : Controlled]$$

$$SystemState \triangleq [rw : RealWorld, tis : IDStation]$$

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 presented. The variable *status* is used to record the state the TIS is in, and can take the values

indicated in the state bubbles of Fig. 7. Variable *issuerKey* is a partial function representing the public key chain, which is needed to authorise user entry. *Operations.* We now specify a selection of the operations over *IDStation*[6]:

$$BioCheckRequired \triangleq \begin{pmatrix} status = \mathsf{gotUserToken} \wedge userTokenPresence = \mathsf{present} \\ \wedge\ UserTokenOK \wedge (\neg UserTokenWithOKAuthCert) \end{pmatrix}$$
$$\longrightarrow status := \mathsf{waitingFinger}\ \mathbin{;}\ currentDisplay := \mathsf{insertFinger}$$

$$ReadFingerOK \triangleq \begin{pmatrix} status = \mathsf{waitingFinger} \wedge fingerPresence = \mathsf{present} \\ \wedge\ userTokenPresence = \mathsf{present} \end{pmatrix}$$
$$\longrightarrow status := \mathsf{gotFinger}\ \mathbin{;}\ currentDisplay := \mathsf{wait}$$

$$UnlockDoorOK \triangleq \begin{pmatrix} status = \mathsf{waitingRemoveTokenSuccess} \\ \wedge\ userTokenPresence = \mathsf{absent} \end{pmatrix}$$
$$\longrightarrow \quad UnlockDoor\ \mathbin{;}\ status := \mathsf{quiescent}\ \mathbin{;}$$
$$currentDisplay := \mathsf{doorUnlocked}$$

Each operation is guarded by execution conditions and consist of several assignments. *BioCheckRequired* requires that the current state is **gotUserToken**, the user token is **present**, and sufficient for entry (*UserTokenOK*), but there is no authorisation certificate (¬*UserTokenWithOKAuthCert*). The latter two predicates essentially require that (1) the three certificates can be verified against the public key store, and (2) additionally there is a valid authorisation certificate present. Their definitions can be found elsewhere [6]. *BioCheckRequired* updates the state to **waitingFinger** and the display with an instruction to provide a fingerprint. *UnlockDoorOK* requires that the current state is **waitingRemoveTokenSuccess**, and the token has been removed. It unlocks the door, using the elided operation *UnlockDoor*, returns the status to **quiescent**, and updates the display.

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

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

In Z, this functionality is provided by the schema *UserEntryContext* [6], 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*); this separation enables modular reasoning. The behaviour of all monitored variables other than *now* is arbitrary, and all controlled variables are unchanged. Then, we promote each operation, for example *TISReadTokenOK* $\triangleq$ *UEC*(*ReadTokenOK*). The overall behaviour of the entry operations is given below:

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

---

[6] Most TIS operations have been mechanised, using the same names as in [6].

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

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

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 $TISUserEntryOp \mathbin{;} TISUpdate$.

5.2   Formal Verification of SFR1

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

$$Inv_1 \triangleq \begin{array}{l} status \in \left\{ \begin{array}{l} \textit{gotFinger}, \textit{waitingFinger}, \textit{waitingUpdateToken} \\ \textit{waitingEntry}, \textit{waitingUpdateTokenSuccess} \end{array} \right\} \\ \Rightarrow (UserTokenWithOKAuthCert \lor UserTokenOK) \end{array}$$

$$Inv_2 \triangleq \begin{array}{l} status \in \{\textit{waitingEntry}, \textit{waitingRemoveTokenSuccess}\} \\ \Rightarrow (UserTokenWithOKAuthCert \lor FingerOK) \end{array}$$

$$TIS\text{-}inv \triangleq Inv_1 \land Inv_2 \land \cdots$$

Inv_1 states that whenever the TIS is in a state beyond *gotUserToken*, then either a valid authorisation certificate is present, or else the user token is valid; it corresponds to the first invariant in the *IDStation* schema [6, page 26]. Inv_2 states that whenever in state *waitingEntry* or *waitingRemoveTokenSuccess*, then either an authorisation certificate or a valid finger print is present. Inv_2 is actually not present in [6], but we found it necessary to satisfy SFR1[7]. We elide the additional eight invariants that deal with administrators, the alarm, and audit data [6].

Unlike [6], which imposes the invariants by construction, we prove that each operation preserves the invariants using Hoare logic, similar to [27]:

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

This theorem shows that the state machine never violates the 10 state invariants, and we can assume that they hold, to satisfy any requirements. This involves discharging verification conditions for a total of 22 operations in Isabelle/UTP, a process that is automated using our proof tactic `hoare_auto`.

We use this 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 [10], which mirrors the (informal) Z schema

---

[7] There seems to be no invariant that ensures the presence of a valid fingerprint in [6]. We also believe that a necessary invariant regarding admin roles is missing.

domain calculations in [7, page 5]. Specifically, we characterise the weakest precondition under which execution of *TISUserEntryOp* followed by *TISUpdate* leads to a state satisfying $rw{:}ctrl{:}latch = $ **unlocked**. We formalise this in the theorem below.

Theorem 5.2 (FSFR1)

$$\left(\begin{array}{l} \textit{TIS-inv} \wedge \textit{tis}{:}currentLatch = \textbf{locked} \\ \wedge \, (\,TISUserEntryOp \,\fatsemi\, TISUpdate\,) \, \textbf{wp} \, (\,rw{:}ctrl{:}latch = \textbf{unlocked}) \end{array} \right)$$
$$\Rightarrow ((\,UserTokenOK \wedge FingerOK\,) \vee UserTokenWithOKAuthCert)$$

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

We conjoin the **wp** formula with $tis{:}currentLatch = $ **locked** to capture behaviours when the latch was initially locked. The only operation that unlocks the door for users is *UnlockDoorOK*, as confirmed by the calculated unlocking precondition:

$$status = \textbf{waitingRemoveTokenSuccess} \wedge userTokenPresence = \textbf{absent}$$

that is, access is permitted and the token has been removed. We conjoin this with *TIS-Inv*, since we know 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 else a valid authorisation certificate. We have now verified a formalisation of SFR1. In the next section we place this in the context of an AC.
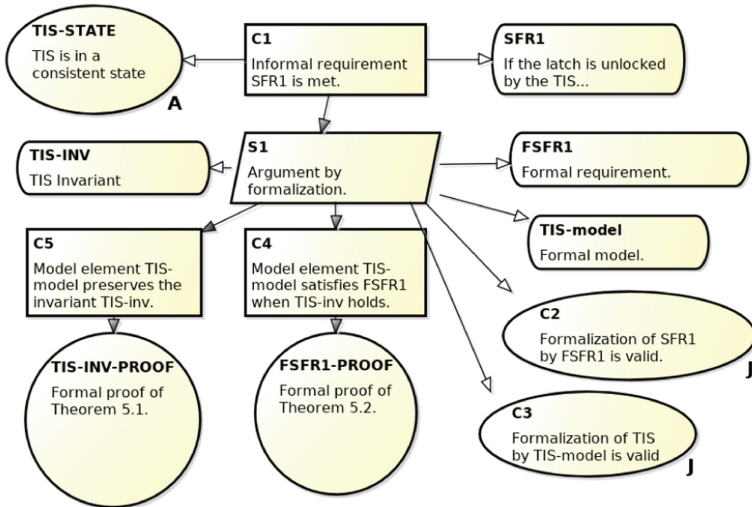

Fig. 8. TIS claim formalization

6   Mechanising the Tokeener Assurance Case

Here, we use Isabelle/SACM to model an AC with the claim that TIS satisfies **SFR1**, using Theorems 5.1 and 5.2 from Sect. 5 as evidential artifacts. The GSN diagram for the AC is shown in Fig. 8, which is inspired by the "formalisation pattern" [9]. Figure 8 is translated to IAL and the result is show in Figs. 9 and 10, which illustrate (1) a machine checked AC; (2) integration of informal, formal, and semi-formal artifacts; and (3) use of Isabelle/UTP verification techniques.

The Formalisation Pattern. [9] shows how results from a formal method can be used to provide evidence to an AC that claims to satisfy a given requirement {R}. The strategy used to decompose the claim "Informal requirement {R} is met by {S}" is contingent on the validation of both the formal model of {R} and the formal model of {S}. Consequently, the pattern breaks down the satisfaction of {R} into 3 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 usually have an informal process argument. In Fig. 8, we adapt this pattern as follows. Firstly, instead of using two validation claims, we use two justification elements, **C2** and **C3**. 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 **C4**. Secondly, we add the "requirement satisfaction" claim **C5** for the state invariant of TIS.

   In Fig. 8, we apply our adapted pattern to **C1**. This claim states that the informal requirement SFR1 is met, and references **SFR1**, with its natural language description, and the assumption **TIS-STATE**. The latter is important, as the AC's requirements are only satisfied when the invariant in Sect. 5 holds. **C1** is decomposed by the formalisation strategy, **S1**, which references the three formal artifacts **TIS-INV** (*TIS-Inv*), **FSFR1** and **TIS-model** (*TISUserEntryOp*) from Sect. 5. This decomposition is contingent on the validation arguments expressed by **C3** and **C2**. The latter could be an explanation of how FSFR1 formalises SFR1, such as the description of Theorem 5.2 in Sect. 5. **S1** subclaims are **C4** and **C5**. The former is supported by the evidence **FSFR1-PROOF** which refers to Theorem 5.2, and the latter by **TIS-INV-PROOF** which refers to Theorem 5.1.

Claims. Figure 9 shows the model of **C1**–**C5** from Fig. 8. In SACM, justifications, asusmptions, and claims are unified by the class `Assertion`. Thus, the claims and justifications from Fig. 8 are all represented by claims in Fig. 9. They are created using the command `CLAIM`, with a name and content associated. Since the checks done by IAL are successful, no errors are issued.

Formal, Semi-formal, and Informal. The `CONTENT` of the claims in Fig. 9 integrate hyperlinks, which are generated by antiquotations that reference semi-formal artifacts, i.e. instances created by IAL, formal artifacts, i.e. theorems and proof techniques created by Isabelle/HOL commands, and informal artifacts, i.e., natural language. For example, the `CONTENT` of `C4` combines natural language with the antiquotation `@{const <FSFR1>}` to insert a hyperlink to the formal artifact

FSFR1. Also, `C1` refers to the semi-formal artifact `SFR1`, and `SFR1` copies the natural language requirement from the Tokeneer documentation.

```
EXPRESSION SFR1 CONTENT‹If the latch is unlocked by the TIS, then the TIS must be ...›

CLAIM C1 CONTENT‹Informal Requirement @{Expression SFR1} is met.›

CLAIM C2 ASSUMED
    CONTENT‹Formalization of the informal security requirement @{Expression SFR1}
           by the formal requirement @{const FSFR1} is valid.›

CLAIM C3 ASSUMED
    CONTENT‹Formalization of system element @{Resource TIS} by model @{const TIS_model} is valid.›

CLAIM C4 NEEDS_SUPPORT
    CONTENT‹The model element @{const TIS_model} satisfies the formal requirement @{const FSFR1}.›

CLAIM C5 NEEDS_SUPPORT
    CONTENT‹The model element @{const TIS_model} preserves the invariant @{const TIS_inv}.›

ASSERTED_INFERENCE S1
    SOURCE‹{@{Claim C2}, @{Claim C3},
           @{Claim C4}, @{Claim C5} }›
    TARGET‹{@{Claim C1} }› ARGUMENT_REASONING‹Argument by formalization›

ASSERTED_CONTEXT AC1
    SOURCE‹{@{ArtifactReference ISABELLE_2018_REF}, @{ArtifactReference TIS_INV_DEF_ACT_REF},
           @{ArtifactReference TIS_FSFR1_DEF_ACT_REF}, @{ArtifactReference TIS_MODEL_DEF_ACT_REF} }›
    TARGET‹{@{AssertedInference S1} }›

ASSERTED_EVIDENCE AE1
    SOURCE‹{@{ArtifactReference TIS_FSFR1_PROOF_REF} }› TARGET‹{@{Claim C4} }›
    ARGUMENT_REASONING‹The claim @{Claim C4} is established by @{thm FSFR1_proof}.›

ASSERTED_EVIDENCE AE2
    SOURCE‹{@{ArtifactReference TIS_INV_PROOF_REF} }› TARGET‹{@{Claim C5} }›
    ARGUMENT_REASONING‹The claim @{Claim C5} is established by @{thm TIS_inv_proof}.›
```

Fig. 9. TIS argument: claims and their relations in Isabelle/SACM

Relations Between Claims. The strategy **S1** from Fig. 8, connecting the elements **C1**–**C5**, is modelled by `S1` in Fig. 9. `S1` is created using the command `ASSERTED_INFERENCE`, which uses antiquotations to reference the premise claims `C4`, `C5`, `C2` and `C3`, i.e., the `SOURCE`, and the conclusion claim `C1`, i.e., the `TARGET`. `C4` and `C5` are left undeveloped, and hence marked as `NEEDS_SUPPORT`: the argument should be completed later. Moreover, `C2` and `C3` are marked as `ASSUMED`, meaning that this argument is contingent on their satisfaction elsewhere.

Context. We model the relations between the context elements **TIS-INV**, **FSFR1**, **TIS-model** and the strategy **S1** from Fig. 8. This is done in Fig. 9 using the command `ASSERTED_CONTEXT` which creates the relation `AC1`. It uses antiquotations to connect `S1` with: (1) `ISABELLE_2018_REF`, which is an "SACM reference" to the artifact `RESOURCE ISABELLE_2018`, which is created in Fig. 10 and models the verification tool; and (2) `TIS_FSFR1_DEF_ACT_REF`, `TIS_INV_DEF_ACT_REF`. and `TIS_MODEL_DEF_ACT_REF`, which are all artifact references to their corresponding

artifacts created in Fig. 10 using `ACTIVITY`. From the point of view of SACM, artifacts created using `ARTIFACT_REFERENCE` are references to the artifacts, and not the artifacts themselves. Similarly, relationships created using `ASSERTED_CONTEXT` and `ASSERTED_EVIDENCE` link `Assertions` to artifact references.

Evidence. We model the relationships from Fig. 8 that link **C4**, **C5** to **FSFR1-PROOF** and **TIS-INV-PROOF** respectively. This is done in Fig. 9 by `AE1` and `AE2`, which are created using the command `ASSERTED_EVIDENCE`. They support claims `C4` and `C5` by the "SACM references" `TIS_FSFR1_PROOF_REF` and `TIS_INV_PROOF_REF`, respectively. One can see that `TIS_FSFR1_PROOF_REF` and `TIS_INV_PROOF_REF` point to `TIS_FSFR1_PROOF_ACT` and `TIS_INV_PROOF_ACT` using antiquotations. The latter are created in Fig. 10 using the command `ACTIVITY`, which records an activity with a `StartTime` and `EndTime`. They also have a `CONTENT` with antiquotations pointing to the formal artifacts `@{thm FSFR1_proof}` and `@{thm TIS_inv_proof}`, which are Theorems 5.2 and 5.1, respectively. Also, the antiquotations `@{method hoare_auto}` and `@{method rel_auto}` reference the formal artifacts `hoare_auto` and `rel_auto`, which are Isabelle/UTP proof tactics.

```
ACTIVITY TIS_MODEL_DEF_ACT StartTime ‹25/03/2019› EndTime‹28/03/2019› PROPERTIES‹{}›
    CONTENT‹Definition of the formal model @{const TIS_model}.›

ACTIVITY TIS_INV_DEF_ACT StartTime ‹29/03/2019› EndTime‹29/03/2019› PROPERTIES‹{}›
    CONTENT‹Definition of the state invariant @{const TIS_inv}.›

ACTIVITY TIS_INV_PROOF_ACT StartTime ‹29/03/2019› EndTime‹29/03/2019› PROPERTIES‹{}›
    CONTENT‹Proof of the state invariant preservation @{thm TIS_inv_proof} by @{method hoare_auto}.›

ACTIVITY TIS_FSFR1_DEF_ACT StartTime ‹02/04/2019› EndTime‹02/04/2019› PROPERTIES‹{}›
    CONTENT‹Definition of the formal requirement @{const FSFR1}.›

ACTIVITY TIS_FSFR1_PROOF_ACT StartTime ‹02/04/2019› EndTime‹10/04/2019› PROPERTIES‹{}›
    CONTENT‹Discharging the proof obligation related to @{thm FSFR1_proof} by @{method rel_auto}.›

RESOURCE TIS PROPERTIES ‹{}› LOCATION‹@{url "https://www.adacore.com/tokeneer/"}›
    CONTENT‹Website with the specification documents for the TIS.›

RESOURCE Isabelle2018 PROPERTIES ‹{}› LOCATION‹@{url "https://isabelle.in.tum.de/"}›
    CONTENT‹Website of the Isabelle Interactive Theorem Prover.›

ARTIFACT_REFERENCE ISABELLE_2018_REF REFERENCED_ARTIFACTS‹{@{Resource Isabelle2018} }›
ARTIFACT_REFERENCE TIS_MODEL_DEF_ACT_REF REFERENCED_ARTIFACTS‹{@{Activity TIS_MODEL_DEF_ACT} }›
ARTIFACT_REFERENCE TIS_INV_DEF_ACT_REF REFERENCED_ARTIFACTS‹{@{Activity TIS_INV_DEF_ACT} }›
ARTIFACT_REFERENCE TIS_FSFR1_DEF_ACT_REF REFERENCED_ARTIFACTS‹{@{Activity TIS_FSFR1_DEF_ACT} }›
ARTIFACT_REFERENCE TIS_FSFR1_PROOF_REF REFERENCED_ARTIFACTS‹{@{Activity TIS_FSFR1_PROOF_ACT} }›
ARTIFACT_REFERENCE TIS_INV_PROOF_REF REFERENCED_ARTIFACTS‹{@{Activity TIS_INV_PROOF_ACT} }›
```

Fig. 10. TIS argument: artifacts and their relations in Isabelle/SACM

7   Related Work

Woodcock et al. [34] highlight defects of the Tokeneer SPARK implementation, indicate undischarged verification conditions, and perform robustness tests generated by the Alloy SAT solver [23] 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 can help using this approach more intensively.

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" [33]. As we have argued elsewhere [19], such benchmarks allow us to conduct objective analyses of assurance techniques to aid their transfer to other domains. The issues highlighted in [34] 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 scope of our work and does not harm Tokeneer in its function as a benchmark.

Rivera et al. [27] 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 verify the security properties (Sect. 3) using Hoare triples. Our work uses a similar abstract machine specification, but with weakest precondition as the main tool for the requirements. Beyond the replication of the Tokeneer case study, [27] deals 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 similarly applied.

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 technique to collision avoidance for autonomous robots [18]; a modern benchmark.

Several works bring formality to assurance cases [8,9,11,29]. AdvoCATE is a powerful graphical tool for the construction of GSN-based safety cases [9]. It uses a formal foundation called argument structures, which prescribe well-formedness checks for the graph structure, and allow instantiation of assurance case patterns. Our work likewise ensures well-formedness, and additionally allows the embedding of formal content. Denney's formalisation pattern [9] is an inspiration for our work. Our framework is an assurance backend, which complements AdvoCATE with a deep integration of modelling and specification formalisms.

Rushby shows how assurance arguments can be embedded into formal logic to overcome logical fallacies [29]. Our framework similarly allows reasoning using formal logic, but additionally allows us to combine formal and informal artifacts. We were also inspired by the work on evidential toolbus [8], which allows the combination of evidence from several formal and semi-formal analysis tools. Isabelle similarly allows integration of a variety of formal analysis tools [31].

8   Conclusions

We have presented Isabelle/SACM, a framework for computer-assisted assurance cases with integrated formal methods. We showed how SACM is embedded

into Isabelle as an ontology, and provided an interactive assurance language that generates valid instances. We applied it to part of the Tokeneer security case, including verification of one of the security functional requirements, and embedded these results into a mechanised assurance argument. Isabelle/SACM enforces the usage of formal ontological links which represent the provenance between the assurance arguments and their claims, a feature inherited from DOF. Isabelle/SACM combines features from Isabelle/HOL, DOF, and SACM in a way that allows integration of formal methods and ACs [18].

In future work, we will connect Isabelle/SACM to a graphical AC tool, such as ACME [30], which will make the platform more accessible. We will consider the integration of AC pattern execution [9], to facilitate AC production. We will also complete the mechanisation of the TIS security case, including the overarching argument for how the formal evidence can satisfy the requirements of CC [5]. In parallel, we are developing our verification framework, Isabelle/UTP [16,17] to support a variety of software engineering notations. We recently demonstrated formal verification facilities for a statechart-like notation [12,13], and are also working towards tools to support hybrid dynamical languages [15] like Modelica and Simulink. Our overarching goal is a comprehensive assurance framework supported by a variety of integrated formal methods in order to support complex certification tasks for cyber-physical systems such as autonomous robots [18,19].

References

1. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the Tokeneer enclave protection software. In: Proceedings of IEEE International Symposium on Secure Software Engineering (ISSSE) (2006)
2. Bishop, P.G., Bloomfield, R.E.: A methodology for safety case development. In: Redmill, F., Anderson, T. (eds.) Industrial Perspectives of Safety-Critical Systems, pp. 194–204. Springer, London (1998). https://doi.org/10.1007/978-1-4471-1534-2_14
3. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 12–27. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24364-6_2
4. Brucker, A.D., Ait-Sadoune, I., Crisafulli, P., Wolff, B.: Using the isabelle ontology framework. In: Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A. (eds.) CICM 2018. LNCS (LNAI), vol. 11006, pp. 23–38. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96812-4_3
5. Common Criteria Consortium: Common criteria for information technology security evaluation - part 1: introduction and general model. Technical report CCMB-2017-04-001 (2017). https://www.commoncriteriaportal.org

6. Cooper, D., et al.: Tokeneer ID station: formal specification. Technical report, Praxis High Integrity Systems, August 2008. https://www.adacore.com/tokeneer
7. Cooper, D., et al.: Tokeneer ID station: security properties. Technical report, Praxis High Integrity Systems, August 2008. https://www.adacore.com/tokeneer
8. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the evidential tool bus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 275–294. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_18
9. Denney, E., Pai, G.: Tool support for assurance case development. Autom. Softw. Eng. **25**, 435–499 (2018)
10. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
11. Diskin, Z., Maibaum, T., Wassyng, A., Wynn-Williams, S., Lawford, M.: Assurance via model transformations and their hierarchical refinement. In: MODELS. IEEE (2018)
12. Foster, S., Baxter, J., Cavalcanti, A., Miyazawa, A., Woodcock, J.: Automating verification of state machines with reactive designs and Isabelle/UTP. In: Bae, K., Ölveczky, P.C. (eds.) FACS 2018. LNCS, vol. 11222, pp. 137–155. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02146-7_7
13. Foster, S., Cavalcanti, A., Canham, S., Woodcock, J., Zeyda, F.: Unifying theories of reactive design contracts. Theoretical Computer Science, September 2019
14. Foster, S., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying theories of time with generalised reactive processes. Inf. Process. Lett. **135**, 47–52 (2018)
15. Foster, S., Thiele, B., Cavalcanti, A., Woodcock, J.: Towards a UTP semantics for modelica. In: Bowen, J.P., Zhu, H. (eds.) UTP 2016. LNCS, vol. 10134, pp. 44–64. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52228-9_3
16. Foster, S., Zeyda, F., Nemouchi, Y., Ribeiro, P., Wolff, B.: Isabelle/UTP: mechanised theory engineering for unifying theories of programming. Archive of Formal Proofs (2019). https://www.isa-afp.org/entries/UTP.html
17. Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: Sampaio, A., Wang, F. (eds.) ICTAC 2016. LNCS, vol. 9965, pp. 295–314. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46750-4_17
18. Gleirscher, M., Foster, S., Nemouchi, Y.: Evolution of formal model-based assurance cases for autonomous robots. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 87–104. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_5
19. Gleirscher, M., Foster, S., Woodcock, J.: New opportunities for integrated formal methods. ACM Comput. Surv. (2019, in Press). Preprint: https://arxiv.org/abs/1812.10103
20. Greenwell, W., Knight, J., Holloway, C.M., Pease, J.: A taxonomy of fallacies in system safety arguments. In: Proceedings of the 24th International System Safety Conference, July 2006
21. Habli, I., Kelly, T.: Balancing the formal and informal in safety case arguments. In: VeriSure Workshop, colocated with CAV, July 2014
22. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Upper Saddle River (1998)
23. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2000)
24. Kelly, T.: Arguing safety - a systematic approach to safety case management. Ph.D. thesis, University of York (1998)

25. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9

26. Paige, R.F.: A meta-method for formal method integration. In: Fitzgerald, J., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 473–494. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63533-5_25

27. Rivera, V., Bhattacharya, S., Cataño, N.: Undertaking the tokeneer challenge in event-B. In: FormaliSE 2016. ACM Press (2016)

28. Rushby, J.: Logic and epistemology in safety cases. In: Bitsch, F., Guiochet, J., Kaâniche, M. (eds.) SAFECOMP 2013. LNCS, vol. 8153, pp. 1–7. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40793-2_1

29. Rushby, J.: Mechanized support for assurance case argumentation. In: Nakano, Y., Satoh, K., Bekki, D. (eds.) JSAI-isAI 2013. LNCS (LNAI), vol. 8417, pp. 304–318. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10061-6_20

30. Wei, R., Kelly, T., Dai, X., Zhao, S., Hawkins, R.: Model based system assurance using the structured assurance case metamodel. Syst. Softw. **154**, 211–233 (2019)

31. Wenzel, M., Wolff, B.: Building formal method tools in the Isabelle/Isar framework. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 352–367. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74591-4_26

32. Wenzel, M.: Isabelle/jEdit as IDE for domain-specific formal languages and informal text documents. In: Proceedings of the 4th Workshop on Formal Integrated Development Environment (F-IDE), pp. 71–84 (2018). https://doi.org/10.4204/EPTCS.284.6

33. Woodcock, J.: First steps in the verified software grand challenge. IEEE Comput. **39**(10), 57–64 (2006)

34. Woodcock, J., Aydal, E.G., Chapman, R.: The tokeneer experiments. In: Roscoe, A.W., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare, pp. 405–430. Springer, London (2010). https://doi.org/10.1007/978-1-84882-912-1_17