Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01201-0

SPECIAL SECTION PAPER

Evaluating formal model verification tools in an industrial context: the case of a smart device life cycle management system

Maxime Méré[1,2] · Frédéric Jouault[3] · Loïc Pallardy[2] · Richard Perdriau[3,4]

Received: 4 May 2023 / Revised: 22 July 2024 / Accepted: 25 July 2024

Abstract
The formal verification of the properties of semi-formal models can make it easier to ensure their security and safety. However, this task is generally cumbersome for non-specialists in formal verification, particularly in an industrial context. This paper introduces an evaluation of four formal verification tools on an industrial case, called a Life Cycle Management System (LCMS). This LCMS makes it possible to deploy Product-Service Systems (PSSs) to customers using Systems-on-Chip (SoC). A PSS is a business model in which products and services are tightly connected and whose objective is to optimize the use of products, with a positive environmental impact. A SoC can embed hardware security; however, a LCMS must be secure from end to end, which requires a verification not only of the used protocol (in this case, a blockchain-based protocol), but also of the whole architecture. For that purpose, semi-formal UML models of a LCMS were first specified and designed with their associated properties, then improved in order to be formally verifiable. Despite being more complex, they remain capable of being processed by dedicated tools. In this paper, Verifpal and ProVerif, two formal cryptographic protocol verifiers, are used and evaluated for the cryptographic protocol and AnimUML (developed by one of the authors) and HugoRT, two verification tools for behavior and UML for the architectural model are evaluated. These tools are assessed and compared according to their coverage of properties and state spaces, limitations, and usability for non-specialists. Some limitations of the approach itself are also provided.

Keywords UML verification · Cryptographic protocols · Formal verification tools · Formally verifiable models · Life cycle management systems · Smart devices

1 Introduction

Smart devices and the Internet of Things (IoT) are two of the fastest growing areas of the electronics industry [1]. These devices are generally built around a System-on-Chip (SoC), which, in addition to the central processing unit, integrates multiple hardware blocks implementing useful features. For instance, a SoC typically integrates random access memory, flash memory, communication peripherals, cryptographic accelerators, and power management, which would otherwise have to be provided by separate components. To the authors' knowledge, apart from a few research projects, activating new features in such SoCs by the way of customization can be achieved either in the very first manufacturing stages or for a limited number of times. The main reason is that there is currently no secure industrial solution to achieve this yet. Therefore, the introduction of a Life Cycle Management System (LCMS), which enables secure actions in untrusted environments throughout a product's life

Communicated by N. Bencomo, M. Wimmer, H. Sahraoui, and E. Syriani.

✉ Maxime Méré
maxime.mere@st.com

Frédéric Jouault
f.jouault@gmail.com

Loïc Pallardy
loic.pallardy@st.com

Richard Perdriau
richard.perdriau@eseo.fr

1 Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes 35000, France

2 STMicroelectronics, Le Mans 72100, France

3 ESEO-Tech, Angers 49100, France

4 CNRS, IETR - UMR 6164, Rennes 35000, France

cycle, would make it possible to reconfigure a SoC dynamically in a secure manner and, consequently, to implement Product-Service Systems (PSSs) [2]. Sometimes referred to as Product-as-a-Service (PaaS), this way of providing products is considered more sustainable. A PSS considers that a user does not pay for goods (physical objects) but for a service provided by an object. Therefore, the manufacturer is encouraged to produce long-lasting and maintainable products, of which it is in charge of their maintenance. For the time being, this type of services is mainly offered to companies [3]. The way printers are now often associated with consumable subscriptions is a typical PSS example, although the manufacturer is not necessarily in charge of the printer's maintenance. However, this has not been widely extended to electronic devices yet. A LCMS could, for instance, automatically disable a device unless its associated subscription is renewed.

To be deployed, a LCMS requires a high level of security to guarantee object functionalities cannot be compromised by an attacker. For this reason, formal verification of such a system's properties is highly desirable. Model checking tools are however often difficult to use by non-specialists. Therefore, the objective of this work is to obtain results from models designed with techniques more suitable to an industrial context. This paper presents a methodology and provide feedback on the usage and features of the associated tools.[1]

Within the team, the authors are organized as follows:

- An (internal) industrial client (third author) who requests an LCMS component.
- A designer (first author) who establishes the specification and design models, and who verifies the design models.
- A scientific and modeling expert (second author) who gives advice on modeling, model debugging, and LTL formulation.
- And finally, a SoC scientific expert (fourth author) who helps in identifying the points of scientific value.

The approach we followed consists in first defining the requirements, including informal properties, then creating design models that need to be verified. Two aspects are modeled: the architecture, and the cryptographic protocol. For each of these, two verification tools are evaluated.

This article is an extension of a conference paper presented at MODELS 2022 [4]. It includes the original article content with some additional Verifpal and AnimUML verification result details. Moreover, the context and motivation are further detailed to give a better understanding of the use case. In addition to the use of AnimUML, HugoRT, a second UML-based verification tool, is used for comparison and feedback. In a similar way, a new ProVerif verification part, as well as the feedback associated with this tool are added for protocol verification analysis. Finally, a new section called “Limitations" discusses the different limitations of the case study. The limitations of each tool's evaluation are given in their respective sections.

The main results of this work are the following. (1) AnimUML is relatively simple to use by engineers knowing UML, whereas HugoRT has more capabilities, but requires expertise in its backend model checkers from its users. (2) ProVerif is more powerful than Verifpal but also more complex to use. In contrast, Verifpal models are quite similar to UML sequence diagrams. Meanwhile, ProVerif employs a process-based modeling approach, where messages are exchanged through channels.

The main expected benefits of this work are threefold. Firstly, engineers can apply the presented methodology, as well as decide which verification tools to use based on our evaluation. Secondly, tool providers get some feedback from an application of their tools to an industrial case study. Finally, researchers may get some ideas about which model verification topics require additional research.

The paper is organized as follows. Section 2 presents the background case study and the motivations behind the intention to verify the model of such a system. Section 3 describes the case's design and the properties to be checked. It also describes the approach to create a design model that can be verified. Problems that may be encountered during the verification process are described in Sect. 4, while Sect. 5 describes some related work. Sections 6, 7 and 8 detail the approach to formally verify the models using selected tools. The feedback on modeling and tool use is then presented in Sect. 9. Finally, Sect. 10 presents the limitations of the case study, before Sect. 11 gives some concluding remarks.

2 Context and motivation

As mentioned previously, a LCMS offers the ability to perform secure remote actions on a product at any state of its life cycle. There are already a few proposals in the literature for applications similar to this LCMS concept [5, 6]. However, these proposals either do not fully meet the full LCMS requirements [7], falling short of enabling a PSS or do not seem to be implementable in the industry. Moreover, none of them has been formally verified so far.

2.1 LCMS procedure

In order to ensure that an SoC is capable of implementing an LCMS, certain features must be present [7]: Dynamically reconfigurable SoCs, SoC ownership management, system trustability. Firstly, a LCMS must have the ability

[1] Additional material is available at https://github.com/meremST/Extended-LCMS-Models

to be reconfigured. This capability guarantees that the product only gives access to services subscribed to by users. It also makes it possible to modify (i.e., activate/deactivate) specific features, typically in the form of IP (for intellectual property) blocks (also more concisely called “IPs"), according to future requests. Secondly, the possibility to manage rights on the functionalities associated with a given device must be ensured. This enables the management, at a component level, of which features are accessible by chip users and which actors can lock or unlock the associated services. Finally, both the consumer and the service provider must have guarantees that the system does not allow one of them to trick the other.

A communication diagram presenting the process of a simple, naive LCMS procedure for realizing these operations is presented in Fig. 1. In this diagram, two users are represented: the Manufacturer, which is the initial owner of an SoC component, and Alice, the manufacturer’s client, who is about to acquire rights or a configuration on the SoC. The different steps of this procedure are as follows.

0. The Manufacturer produces and initializes the LCMS, and the component is provided in an inactivated state to Alice (not shown in Fig. 1).
1. Alice requests specific rights and configurations and pays for them.
2. The Manufacturer gives Alice a corresponding certificate.
3. Alice can transmit the certificate to the SoC to activate the configuration.

This naive LCMS puts forward the risks that such a system implies. Indeed, for now, the currently presented system can be compromised by both the Manufacturer and Alice.

Firstly, the Manufacturer can send a certificate, and Alice can choose not to send payment if the Manufacturer sends the certificate first, and vice versa. This implies the need for a “trusted by both" service to proceed with the transaction.

Secondly, some trust issues can be found with the LCMS element itself. In a standard situation, it is relatively straightforward to ensure that the manufacturer has confidence in the LCMS component. As the designer and manufacturer, they know how the component works. In addition, technologies such as OTP registers [8] (non-rewritable registers based on a fuse system), secure memories [9], trusted firmware [10], or dedicated IP blocks may be considered difficult enough to break to make their exploitation uneconomical. However, once the component is in the user’s environment, it is impossible for the manufacturer to apply patches to the component without their approval. At the opposite end of the spectrum, Alice can only trust the Manufacturer and has no guarantee as to what is implemented. The means of ensuring confidence in the SoC’s implementation are very rare in the literature. The practice that comes closest is the study of ways to limit the impact of hardware Trojans [11]. A hardware Trojan is a malicious modification of an integrated circuit. When activated, hardware Trojans attempt to bypass or disable a system’s security barriers, for example, by divulging confidential information such as private keys.

Other approaches propose to allow the user to verify the implementation [12]. However, they are mainly aimed at manufacturers as opposed to third-party implementations. To avoid implementation mistrust issues in Alice’s case, it is possible to rely on external certifications to guarantee the component’s implementation [13].

2.2 Blockchain-based LCMS

In this industrial case study, it was decided to use a secure element embedded in the SoC, in association with a *blockchain*-based cryptographic protocol. This allows component access rights to be securely transferred among users without the need to trust a third party. Blockchain is a popular technology enabling decentralized, automated and trustworthy digital applications [14].

Blockchain-based LCMSs are present in the literature [4] and are mainly based on the work of Islam et al. [5]. The schematic diagram shown in Fig. 2 presents such a system. Here, a blockchain serves as a trusted element to implement an exchange service. First, the manufacturer registers the component’s information using a smart contract, which then allows users to configure and pay for unlocking elements in the component. The state of the component is then transmitted to the SoC so that it can update itself accordingly.

To the authors’ knowledge, there are no proposals in the literature to implement a blockchain-based LCMS in an industrial environment [7]. This is mostly due to the number of blockchain calls needed for a unique component. Indeed, ownership transfer in the state-of-the-art case is made component by component, as shown in Fig. 3 [5]. This implies too many blockchain transactions, which would typically become too costly when managing a large number of components.

To make this type of operation compatible with industrial production constraints, the case study’s component exchange protocol is designed to operate in batches instead of individual components. The sequence diagram in Fig. 4 presents a batch-based ownership transfer. SoCs inside batches have the same properties (same configuration, same users). The protocol allows the division of a batch into two sub-batches to enable the exchange or modification of these sub-batches independently. This retains the ability to modify component sets in different ways while taking advantage of the fact that, as a general rule, several components will be configured in the same way.

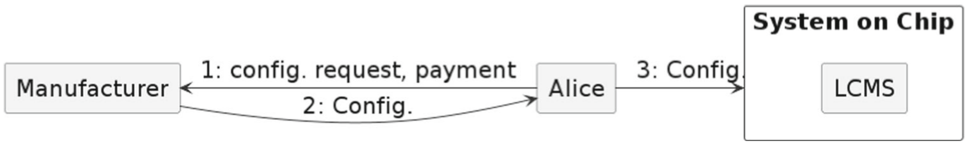


Fig. 1 Communication diagram of a naive life cycle management system

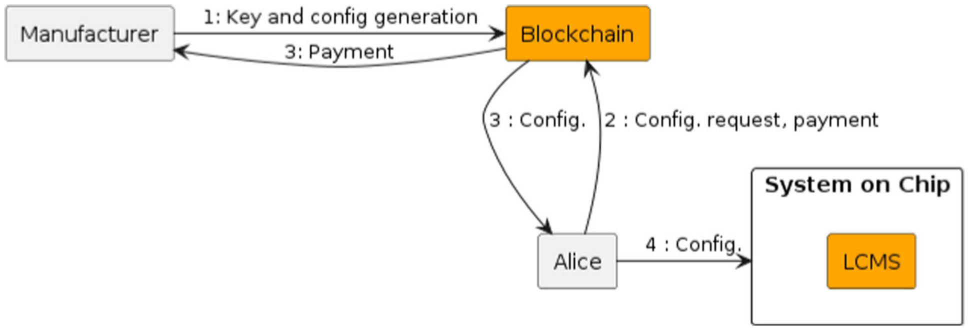


Fig. 2 Communication diagram of a life cycle management system. Colored boxes indicate elements that users must trust (i.e., Blockchain and LCMS)

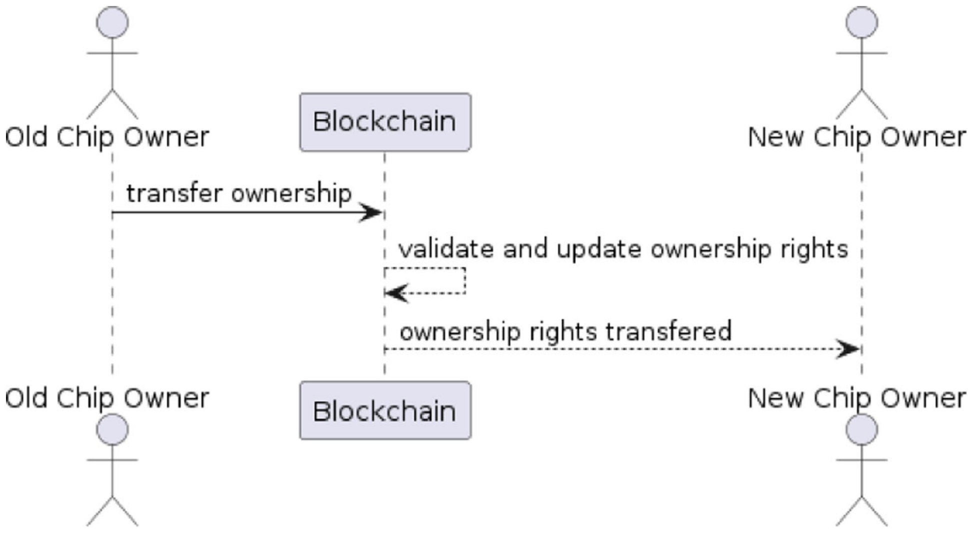


Fig. 3 Standard ownership transfer sequence diagram as is it made in state of the art

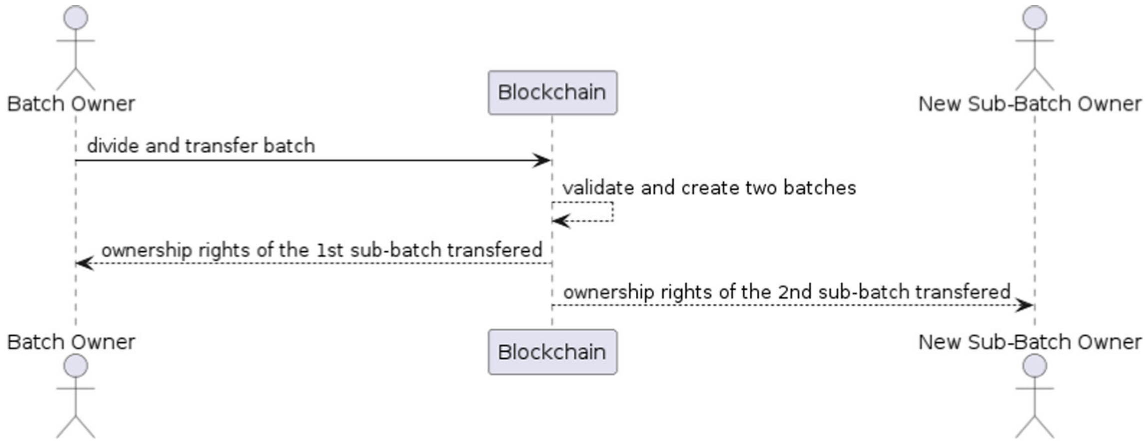

Batch Owner
Blockchain
New Sub-Batch Owner
divide and transfer batch
validate and create two batches
ownership rights of the 1st sub-batch transfered
ownership rights of the 2nd sub-batch transfered
Batch Owner
Blockchain
New Sub-Batch Owner

Fig. 4 Batch ownership transfer sequence diagram. This approach can reduce the number of blockchain transactions

In theory, this approach makes it possible to limit the number of transactions on the blockchain. Indeed, in the standard case, for N components, a minimum of N transactions must be carried out on the blockchain. In a batch system, this number of transactions varies from 1 to N, depending on the level of required component differentiation.

The next section will take a step toward the realization of a blockchain-based LCMS by modeling and verifying its design properties.

3 Approach

This section is intended to present the specification models and provide guidance on how to proceed with the design. In Sect. 3.1, the model specification and the properties to be verified are discussed. Subsequently, in Sect. 3.2, an explanation is provided regarding how the design model was created.

3.1 Model specification and properties

The system under study (the batch-based LCMS) is represented as a black-box interaction diagram in Fig. 5. All actors able to interact with the system are represented with associated connections showing possible exchanged messages. *User* represents service providers and device users. Because manufacturers already know how to perform secure provisioning at production time, and in order to simplify explanations, component provisioning will be considered as already performed. This means that there is no need for a special *Manufacturer* actor, who would be able to send specific provisioning messages to the LCMS. *SecureStorage* and *SoCIPs* represent parts of the SoC in which the system is embedded and with which it interacts. The former corresponds to a kind of storage that is considered secure: it is not possible to recover or modify data by means other than the designed ones. *SoCIPs* represents the set of hardware IP blocks that the system can activate or not. *Blockchain* refers to a decentralized application (or smart contract) running on a blockchain, and the messages it exchanges correspond to smart contract functions that run on the blockchain. Smart contracts are tamper-proof computer programs with self-checking and self-executing properties [15]. Here, they allow users to transfer ownership or to activate or deactivate some IPs/features on a given component or batch. Users can then present corresponding certificates to the LCMS in order to update its internal state, and actually transfer access rights, or enable specific IPs. Actual contents of these smart contracts is out of the scope of this model, which focuses on the LCMS.

Possible interactions between actors and system are the following.

- *getScript*: unauthenticated message sent by any *User* to the system, or by the system to *SecureStorage*. The script corresponds to code written in a specific language used to describe the purchased features of the chip. The details of this scripting language are beyond the scope of this article. *scriptResp* is the corresponding response containing the script, which is a set of instructions that configure the SoC as agreed between the different actors involved. This configuration can be time-dependent.
- *updateScript(isValid)*: authenticated message sent by a specific *User*. *scriptResp* is the corresponding response. *isValid* is a Boolean parameter that abstracts whether the update is valid or not. In practice the SoC will check a certificate's authenticity, which is beyond the scope of this model.
- *powerOn*: message sent by *User*, modeling the startup of the SoC.
- *powerOff*: message sent by *User*, modeling the physical shutdown of the SoC.
- *setScript*: message sent to *SecureStorage*. *writeAck* is the corresponding response.
- *getConfig*: message sent to *SecureStorage*. *configResp* is the corresponding response. This exchange corresponds to the data recovery that enables to configure the SoC IPs.
- *setConfig*: message sent to *SecureStorage*. *writeAck* is the corresponding response. This exchange corresponds to the modification of the data that represents the IP configuration.
- *setIPs*: message sent to *SoCIPs*. This message corresponds to the ability of the system to physically activate or deactivate SoC IPs.

To be used in a trustworthy way, the system must satisfy the following, informally expressed, properties:

- **P1.** Someone taking ownership of a batch of devices should not be able to know which components are present in other batches. It helps in preserve a specific level of confidentiality within data that is publicly accessible.
- **P2.** It must be impossible for an attacker to acquire SoC ownership without the current user's approval.
- **P3.** It must be impossible for an attacker to forge a certificate.
- **P4.** When the user makes a script request with LCMS mode active, the system must answer.
- **P5.** When a write is made to *SecureStorage* it must be due to a valid update.
- **P6.** In the *SecureStorage* memory, a script modification must be followed by a configuration update before the IP setup.

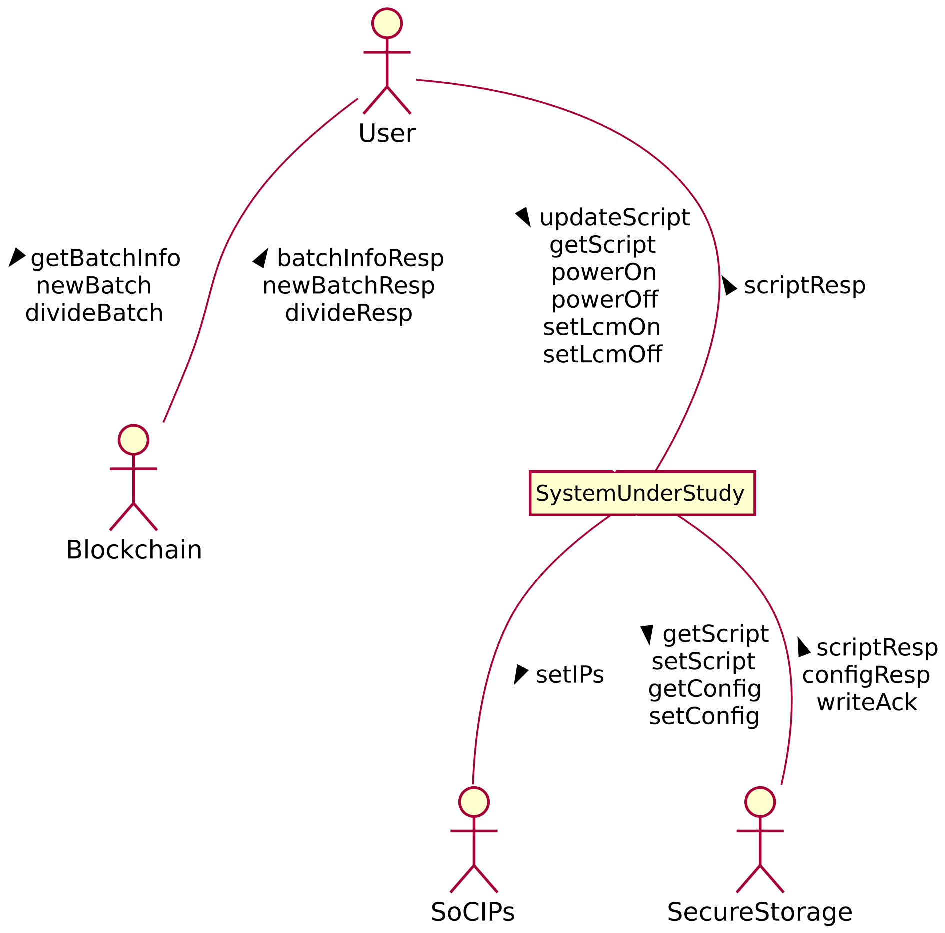


Fig. 5 Interaction diagram for the system under study. Here User is an agglomeration of both Manufacturer and Client roles

Verifying these properties would guarantee a better functioning and security of the system, at least at the design level. Other properties could be desired and verified, but this study focuses on these six only.

3.2 Designing the models

There are two main aspects to model: the system's architecture and the cryptographic protocol. There are therefore, two complementary parts. A first one covers the system's architecture, which focuses on the interactions of the LCMS with its external actors, as well as between its internal elements. A second one dedicated to the cryptographic protocol that allows the generation of certificates at the end of an exchange between involved users and the blockchain. Certificate generation is not the only possible approach [16], but is the one used in this paper's case study. These two aspects complement each other. The first focuses on the hardware operation and abstracts away the interactions between users, while the other is dedicated to protocol and cryptographic functions and does not focus on the internal operation of the SoC. The objective is to obtain security guarantees for a global solution. The following sections provide a description of how they are modeled, which assumptions were made, and how the properties presented above are impacted.

3.2.1 Architecture

The LCMS architecture was modeled using UML [17] and is presented in Fig. 6. All object behaviors were modeled as UML state machines. There are too many state machines to show them all in this paper, but Fig. 7 shows the *ipCtrlr* state machine, the part of the LCMS responsible for the physical configuration of the SoC. More state machines are available in the additional material. This model aims at being as accurate as possible about interactions between the system and the actors, as well as between the former's internal elements.

In order to simplify the creation of this model while being complementary to the protocol model, several assumptions were made.

- Cryptographic and authentication operations were abstracted. Since the protocol model is better suited to check these operations, it is not necessary to overload the architecture model with them.

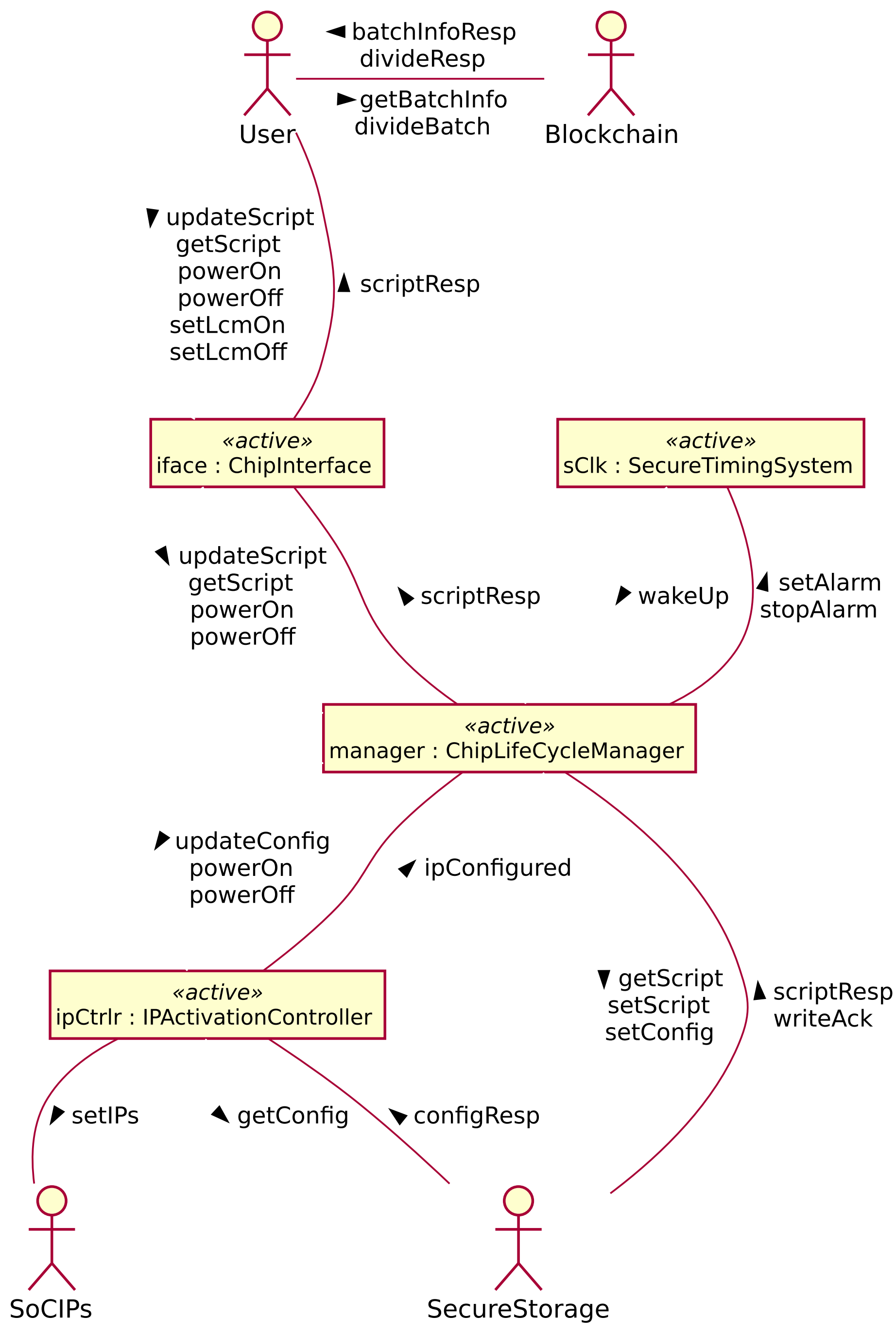


Fig. 6 Architecture diagram of the LCMS

- The concept of batch is not modeled. For the same reasons as for cryptographic operations, this concept is inherent to the protocol model and, therefore, can be abstracted out here.
- SoC provisioning is considered to be already performed.
- All users (service providers and consumers) are represented as a single entity capable of sending messages to the SoC. This abstraction can be made because the model is focused on the behavior of the system itself.

manager is the central part of the system. It is in charge of analyzing proof of ownership, updating the configuration script, and executing it to determine how the SoC should be configured. It controls all other objects in the system. It interacts with the other elements through the following messages.

- *setAlarm*: message sent by manager to *sClk*. *wakeUp* is the corresponding response that can occur at any time.

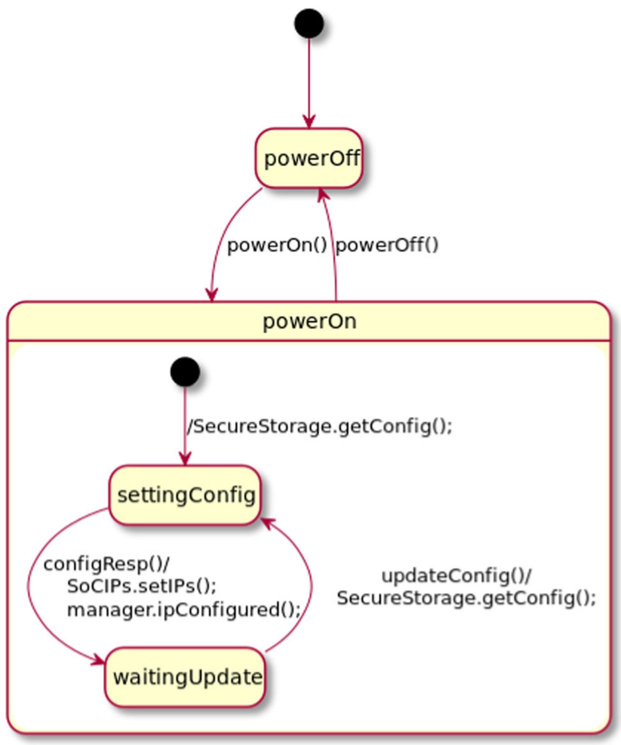


Fig. 7 State machine of the *ipCtrlr* object

- *stopAlarm*: message sent by manage to *sClk* in response to a *wakeUp* message. Its role is to reset *sClk*.
- *updateConfig*: message sent by manager to *ipCtlr*. Its function is to ask *ipCtrl* to load a new configuration.
- *powerOn*: message sent by manager to *ipCtlr*.
- *powerOff* : message sent by manager to *ipCtlr*.

iface represents the interface that the SoC provides to the LCMS component. All user messages go through this element. It is also used to model the physical states of the component, for example powering on the circuit or activating the LCMS mode. These will be performed electrically, but are abstracted as messages here. *iface* communicates using the following messages.

- *getScript*: unauthenticated message sent by *iface* to *manager*. *scriptResp* is the corresponding response always sent.
- *updateScript*: message authenticated by *User* but sent by *iface* to *manager*. *scriptResp* is the corresponding response always sent.
- *powerOn*: message sent by *iface* to *manager*. It is the startup command. Depending on its *lcmPin* parameter, the LCMS will start in normal mode or in life cycle management mode.
- *powerOff* is a message sent by *iface* to *manager*. It is the shutdown command.

sClk is a secure time counter. It allows the LCMS to limit access time to a given configuration or service. This element is not essential for a LCMS but mandatory for a PSS, to make sure service stops at the end of the contracted period, unless it is renewed. The principle of operation of this element is beyond the scope of this paper. *ipCtrl* is an element capable of enabling or disabling hardware features in the SoC. It is thus able to retrieve and implement a configuration generated by the *manager*.

With this architecture-oriented model, properties **P4**, **P5**, and **P6** can be rephrased in the following way:

- **P4.** When User sends *updateScript* or *getScript* to *iface* without sending *powerOff* afterward, and SoC is in LCMS mode, then *User* must receive a *scriptResp* message.
- **P5.** Writes (*setScript* and *setConfig*) on *SecureStorage* must correspond to *updateScript(true)* messages sent to the manager.
- **P6.** When *SecureStorage* receives a *setScript* message, it must always be followed by *setConfig* before the next *setIPs* message is sent to *SoCIPs* by *ipCtrlr*.

3.2.2 Cryptographic protocol

The design of the cryptographic protocol was modeled with a semi-formal sequence diagram, as shown in Fig. 8. This diagram represents the necessary exchanges between actors and system to manage ownership transfer of a specific SoC (*SoC[0]*) from a whole batch (*SoC[0..N]*). In these exchanges, *Alice* is a *User* who is a customer of the PSS. *Manufacturer* is the *User* who provides the service. In this scenario, *Manufacturer* transfers the ownership of one SoC to *Alice*, and then establishes a certificate for the specified SoC configuration modification. *Blockchain* plays the role of a decentralized and automated trusted third party.

Simplification assumptions were made for this model. First of all, the SoC is considered already powered on and set in LCMS mode. Provisioning and configuration modification were modeled but are not detailed on this figure for the sake of brevity. The provisioning step consists in a factory initialization of the cryptographic elements of the chip, and in the initialization of the smart contracts inside the blockchain. Configuration modification consists in updating SoC scripts in a secure way, for instance to activate some IP blocks.

Messages sent among participants are the following.

- Some messages described above: *getScript*, *scriptResp*, and *updateScript*.
- *getBatchInfo*: unauthenticated message that can be sent to the blockchain smart contract by anyone. The batch information represent the proof of ownership of the SoC by an user. This is needed to transfer ownership. *batch-*

InfoResp is the corresponding response always sent back by the smart contract, with parameters (may not be all used at all times).
- *sendUidAndAccProof*: authenticated message that can be sent by the current owner to the future one.
- *sendNewBatch*: unauthenticated (anyone can send it) but signed message sent to the previous owner.
- *divideBatch*: authenticated message sent by a current owner to the smart contract. *divideResp* is the corresponding response.
- *sendProof*: authenticated proof (signed by current owner).

This model should satisfy the **P1**, **P2**, and **P3** properties, which can be further elaborated as:

- **P1.** The secrecy of a non-transfered SoC UID must be preserved. Ideally, the number of non-transferred SoC UIDs should also be confidential.
- **P2.** *batchInfoResp* message data must not be forged by an attacker.
- **P3.** *updateScript* messages must not be forged by an attacker.

After this presentation of the informal design models, the following sections will focus on the verification process as well as on the tools that can be used for that purpose.

4 Verification problems

This section presents the main problems encountered when using formal verification tools, as well as what are the expectations for the choice of a verification method. Moreover, it presents which tools were chosen to perform the verification.

4.1 Common problems

There are two main problems: selecting a tool, and avoiding state space explosion.

4.1.1 Tool requirements

In this case study, tools used to perform automatic property verification must meet two main criteria. First, it should be relatively easy for users with engineering backgrounds to model their systems. Ideally, verifiable models should be as close to the design models (here UML) as possible. Second, the properties to verify should ideally be relatively simple to express for these users.

4.1.2 State space explosion

Most of the tools able to automatically check properties on models perform state space explorations [18]. The state space of a given model is a graph with all its possible configurations as nodes, and all possible transitions between configurations as edges. Checking a property over this graph corresponds to checking whether the property is true on the whole state space. To be able to check properties on a model, its state space must be finite and small enough to be storable, and explorable in a reasonably short time. Otherwise, its state space is said to *explode*.

In general, the more elements a model contains, the more likely its state space will explode. For instance, adding a simple Boolean variable may double the number of possible configurations. Several heuristics help to keep state spaces small enough for verification to be tractable.

- Model only what is necessary for property verification, abstracting the rest.
- Limit the use of variables that can take many distinct values (e.g., floats).
- Constrain the model to “artificially" reduce its state space. It is important to avoid constraining it excessively, otherwise the model may become unrealistic.

4.2 Protocol-related verification

The main challenge with the cryptographic protocol is to make sure that it uses cryptography in a secure way. It is therefore necessary to use a cryptography verification tool. One of the most popular is ProVerif [19], which relies on the Dolev-Yao model [20]. This tool represents a protocol in the form of Horn clauses, and can prove secrecy and authentication properties (and more globally correspondence properties). Secrecy properties assert that an attacker cannot obtain a secret. Authentication properties assert that an attacker cannot impersonate another actor. ProVerif has been shown to be a reliable tool for protocol verification. It offers extensive protocol modeling capabilities, fast execution and reliable results. However, the way protocols are modeled in ProVerif is not necessarily obvious to engineers who are not used to formal verification tools.

Greatly influenced by ProVerif, Verifpal [21] is also designed for verifying the security of cryptographic protocols using the Dolev-Yao model. However, its protocol specification language is significantly simpler. It is relatively close to a sequence diagram, which makes it more suitable for users with engineering backgrounds. Protocols specified with Verifpal can actually be visualized as sequence diagrams. It nonetheless offers formal verification features similar to ProVerif's. To check the security of a protocol, Verifpal, unlike ProVerif, uses an algorithm that performs transfor-

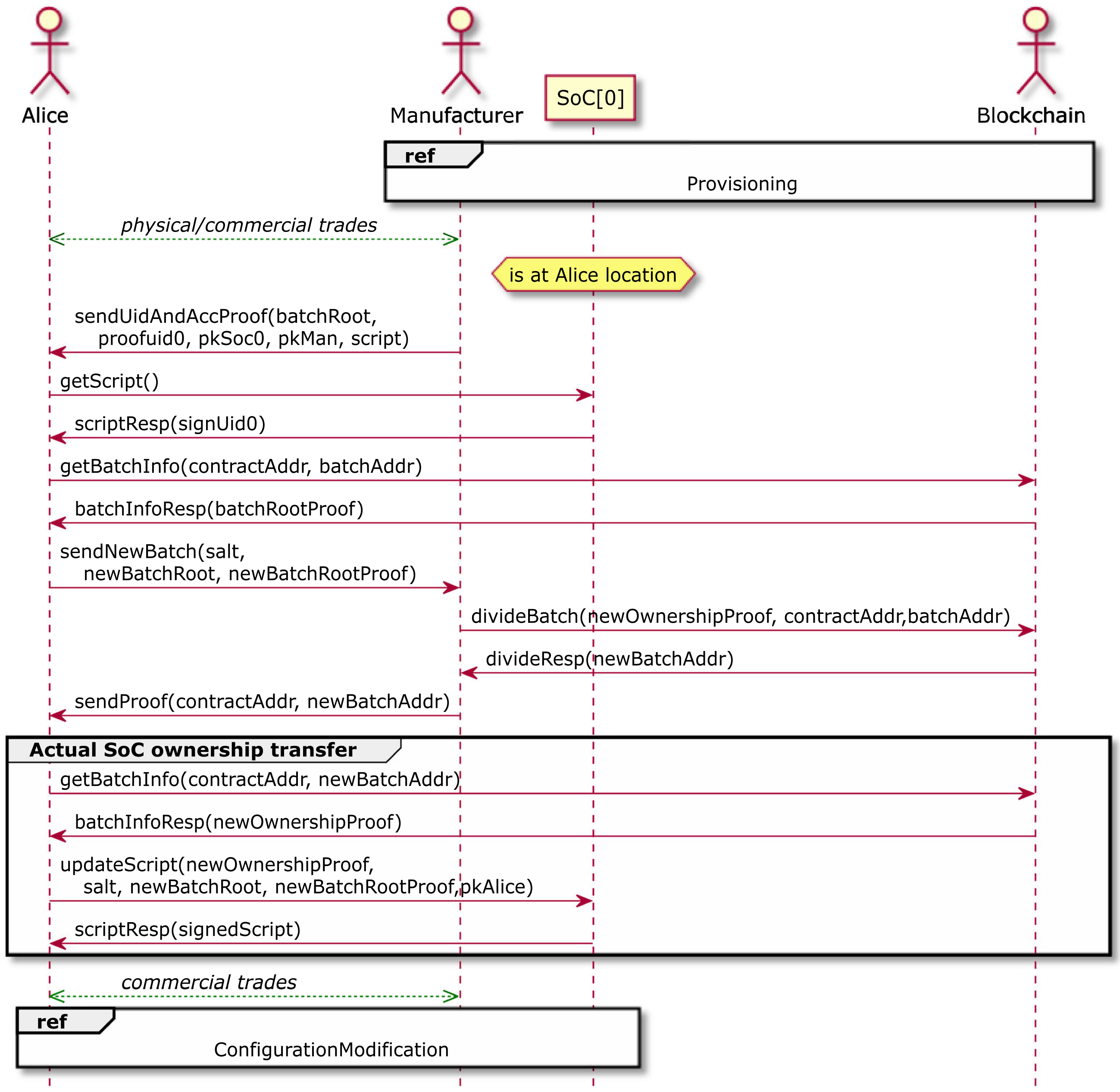

Alice
Manufacturer
SoC[0]
Blockchain
ref
Provisioning
physical/commercial trades
is at Alice location
sendUidAndAccProof(batchRoot, proofuid0, pkSoc0, pkMan, script)
getScript()
scriptResp(signUid0)
getBatchInfo(contractAddr, batchAddr)
batchInfoResp(batchRootProof)
sendNewBatch(salt, newBatchRoot, newBatchRootProof)
divideBatch(newOwnershipProof, contractAddr,batchAddr)
divideResp(newBatchAddr)
sendProof(contractAddr, newBatchAddr)
Actual SoC ownership transfer
getBatchInfo(contractAddr, newBatchAddr)
batchInfoResp(newOwnershipProof)
updateScript(newOwnershipProof, salt, newBatchRoot, newBatchRootProof, pkAlice)
scriptResp(signedScript)
commercial trades
ref
ConfigurationModification

Fig. 8 Sequence diagram of the cryptographic protocol

mations on protocol messages. These transformations come from a set of mutations that an attacker would be able to perform from the information that can be gathered by observation or that can be reconstructed. Verifpal then checks if it has found a contradiction to one of the queries (i.e., properties) specified in the model. Verifpal allows to relatively easily and efficiently model a given protocol. But because of its relative youth, its performance has not been evaluated rigorously yet. Finally, this tool does not support user-defined cryptographic primitives, which can be an obstacle to describe specific protocols. However, this has the advantage of preventing the use of ill-defined primitives.

For our project, we decided to use both Verifpal and ProVerif in order to compare their results. Verifpal could be considered as a better fit with our criteria due to its user-friendly modeling language. But the robustness of the verification algorithm and the better modeling capabilities of ProVerif argue in its favor. More discussion about this trade-off can be found in Sect. 9.

4.3 Architecture-related verification

The initial tool chosen to model check the architecture is AnimUML [22, 23]. However, other UML model checking tools are available, like HugoRT [24]. According to a recent survey [25], there are only three still-maintained UML verification tools: HugoRT, AnimUML, and EMI [26]. Moreover, AnimUML is actually EMI's successor, and most maintenance activity is now focused on the former. For this reason, AnimUML and HugoRT are the two tools that will be used in the remainder of this paper. Those tools are based on UML, which is widely known by engineers.

AnimUML offers the possibility to animate partial UML models. This makes it relatively easy to incrementally work on a testable model, similarly to how developers incrementally build software. AnimUML models can be specified using the PlantUML syntax [27], a relatively simple to use textual UML modeling tool. Furthermore, AnimUML can verify Linear Temporal Logic (LTL) properties by connecting to the OBP2 model checker.[2] Verification thus uses the same UML interpreter that is used for AnimUML model animation. Therefore, verification directly operates at the UML level, without requiring translation to a tool-specific formal language. One benefit is that the counter-examples it produces are also directly expressed at the UML level.

We chose AnimUML because it can be used for specification (e.g., Fig. 5), simulation and debugging of incomplete design models as a way to detect shortcomings to improve, as well as expression of LTL properties and formal model checking from within a single user interface. However, this choice is subjective, considering that AnimUML is developed by one of the authors (the second) of the present paper.

HugoRT supports code generation to Java, C++, and Arduino, as well as model verification. It achieves the latter by translating UML models to the formal language of model checkers used as verification backends: SPIN [28] and UPPAAL [29], which are well known software verification tools. This means that a UML model, which includes active classes featuring state machines, collaborations, interactions, and OCL constraints, can be transformed into the system languages used by these model checkers. Verification is performed by the model checker backends at the level of their respective modeling languages. This means that verification counter-examples have to be translated back to UML, which HugoRT supports. Although AnimUML and HugoRT focus on similar UML subsets, the semantics implemented in HugoRT is slightly different from the one implemented in AnimUML. Moreover, HugoRT does not support partial modeling, and requires complete models to operate. Finally, whereas AnimUML is based on a single semantics definition, in the form of an interpreter, HugoRT consists of multiple transformations from UML to various backends with different semantics. The transformations are responsible for making sure the semantics remain consistent.

[2] http://www.obpcdl.org/

5 Related work

This section describes research work around different modeling and verification approaches for similar projects, also including some examples of work on blockchain modeling.

Dewoprabowo et al. proposed a formal verification of a protocol using ProVerif and TLA+ [30]. In their work, they try to formally verify a divide and conquer key distribution protocol. They use TLA+ to check whether all participants in the protocol retrieve the mutual key simultaneously. Then they use ProVerif to verify the protocol's correctness as well as its security against passive attackers. This approach is similar to ours, which consists in checking architecture with a general-purpose model checker, and cryptography with more specific tools such as Verifpal or ProVerif. Latif et al., have presented an IoT and blockchain-based project modeled with sequence diagrams translated and verified with the TLA+ tool [31]. They used UML to specify a blockchain-based smart waste management system. Then they used the TLA+ toolbox to create and verify a formal model. Rocha et al., have introduced the modeling of blockchain-oriented applications [32]. They proposed three ways to model blockchain applications with well-known modeling languages, including UML class diagrams. Among the research on protocol modeling, Koch et al. proposed a way to automatically transform scenario-based security protocol specifications into equivalent inputs to multiple model checkers [33]. The authors present VICE, a tool able to produce verifiable models from a sequence diagrams. Zhang et al. depict how they used symbolic model checking to verify QUIC, an HTTPS handshake protocol improvement [34]. They used two model checking tools (namely, ProVerif and Verifpal) to perform a formal security analysis of their protocol, to identify design flaws, and to propose a fix. Lauser et al. also discussed how formal models can be used to verify the security of protocols used in modern vehicles: how to model them and how to verify them with tools [35]. Chen et al. have presented work where they use UML sequence diagrams to express formal safety requirements. They then transform them into intermediate semantic models that can be formally verified [36].

There are also several articles that present ways to transform UML models to allow formal verification. Csertan et al. and Cabot et al. proposed tools which follow this line of thought [37, 38]. The first one uses the VIATRA transformation tool to automatically check consistency, completeness, and dependability requirements on UML models. The second one uses UMLtoCSP, a tool that can automatically check several correctness properties starting from UML class diagrams and OCL constraints. Each of these approaches addresses part of the solution we had to build for this work. To the best of our knowledge, no similar blockchain-based hardware system has been modeled and verified with UML yet.

Glouche et al. propose a work that looks like Verifpal's ability to visualize protocols modeled with a cryptographic verifier [39]. They proposed a tool named SPAN that is able to visualize and animate AVISPA models [40]. AVISPA is a cryptographic protocol checker that has the ability to use different verification techniques on the same protocol specification.

Finally, this article focuses on the design and specification of the case study and not on smart contracts. However, there are tools capable of analyzing them, such as Manticore [41, 42], a symbolic execution tool for the analysis of binary programs, and also of smart contracts.

6 Architecture model verification

AnimUML is the tool chosen to create the architecture model. Section 6.1 describes the AnimUML modeling capabilities. Section 6.2 explains the model conversion from AnimUML to HugoRT, which is necessary in order to be able to evaluate the verification capabilities of HugoRT. Section 6.3 is focused on the simplification/abstraction performed on the model to make it verifiable. Section 6.4 explains how the properties to be verified were expressed, and gives the verification results.

6.1 Improving the AnimUML model

With AnimUML, active object behaviors can be defined as state machines, which communicate via asynchronous operation calls. These objects can then be linked to communicate with each other. The preferred method to create a custom AnimUML model is to start from a HTML template file with embedded JavaScript data representing the model. The overall model structure is defined as a JavaScript object, and specific parts can use an extended PlantUML syntax: state machines, interactions (using the PlantUML sequence diagram syntax), and class diagram. Although the PlantUML syntax is relatively permissive, AnimUML enforces more constraints. For instance, it makes it mandatory to correctly write transition labels with:

- A *trigger* that makes it possible to match incoming messages in order to activate the transition.
- A guard so that the transition is only fireable if a specific condition is met.
- An effect that will be executed when the transition is fired.

The syntax used to describe a transition label follows the UML specification, and is in the form: `trigger[ guard] /effect`. With AnimUML, user can construct and visualize an executable model relatively quickly. This makes it possible to animate and test the model's behavior during its construction. Once the model is finalized, it is possible to explore its state space. This helps make sure the model is sufficiently constrained to avoid state space explosion. As a matter of fact, if state space exploration takes too long for the user's taste, it can be stopped, and a heat map of fired transitions is superimposed on the state machine diagrams. A colored disk is displayed next to each transition that shows if it was never fired (in red), or how often it was fired (as shades of green). Moreover, the last reached configuration is also displayed, which shows active objects current states, attribute values, and event pool contents. This significantly helps understand what happens during state space exploration in order to optimize the model. Once the system is modeled, AnimUML allows to write the LTL properties to be verified. Linear Temporal Logic (LTL) [43] is a form of logic, originally designed to formally verify computer programs, which adds time-related modalities in addition to the classical Boolean operators.

LTL formulas depend on predicates over the model. With AnimUML, it is possible to define these predicates as watch expressions that can be evaluated during model execution. In addition to being usable in LTL formulas, these watch expressions can also be used like regular debugging watch expressions. This helps users check that their expressions mean what they intend. AnimUML watch expressions are written in the same JavaScript language as its guards and effects, with support for observation/introspection operators:

- `__ROOT__` can be used to start a path to a specific object.
- `IS_IN_STATE` makes it possible to check whether a given object is in a particular state.
- `EP_IS_EMPTY` checks whether an object has received a given message or not (with *EP* standing for *event pool*).
- `EP_CONTAINS` checks whether a given object has received a specific message or not.

6.2 Converting the model to HugoRT

The fundamental modeling capabilities of HugoRT are similar to those of AnimUML: state machines, and connected objects. However, HugoRT does not support partial modeling, which means that features and behaviors must be defined in classes, and cannot be defined on objects, as allowed by AnimUML. Classes are thus defined along with their properties, and their behavioral features, which may be either operations or receptions. The latter are asynchronous, whereas HugoRT only supports synchronous operation calls. Active classes have their behaviors defined as state machines.

Because of the lack of partial modeling support in HugoRT, and even though AnimUML supports the verification of an incomplete model (e.g., with objects and state machines but no classes), it must be completed before being fed into HugoRT (e.g., classes must be defined). Fortunately, AnimUML is generally able to automatically complete partial models. In practice, it performs model static analysis in order to report issues, including those related to model incompleteness. Most of the reported issues come with quick fixes, which can be used to make the model more complete step-by-step, by solving one issue at a time. Finally, it is also possible to ask the tool to apply all quick fixes in one go, which is exactly what is required here.

Once the AnimUML model has been completed, there remains 5 notable differences that must be addressed for the conversion to work properly:

1. **Communication.** Because AnimUML objects communicate asynchronously, receptions must be used with HugoRT instead of operations.
2. **Trigger arguments.** Whereas AnimUML makes it possible to use any name for transition trigger arguments, only the reception parameter names can be used with HugoRT. This requires some careful renaming.
3. **Associations and connectors.** HugoRT does not support associations, which means that they must be translated into class properties. Similarly, connectors between objects must be translated into object slots.
4. **Choices.** In the case of choice pseudostates, HugoRT does not make it possible to directly access arguments from incoming transition triggers in outgoing transition guards. It is therefore necessary to store them temporarily in specific properties, which must be added to the class.
5. **Deferred triggers.** AnimUML implicitly defers all incoming events that cannot be immediately processed by a state machine, whereas HugoRT drops such events by default. They must therefore be explicitly declared as deferred triggers.

Other than that, the translation is relatively straightforward.

Although we decided to translate our AnimUML model to HugoRT, it would also be possible to directly create a model for HugoRT: either in its own textual syntax, or with a compatible Eclipse UML tool. However, HugoRT does not support debugging. It is actually necessary to translate the model (e.g., to SPIN) in order to see it in action, albeit at the model checker's level, not at the UML level. Because we assumed engineers know UML, but not any specific model checker, directly modeling the system with HugoRT would be relatively difficult.

6.3 Simplifying and abstracting assumptions

Even with the simplifications discussed in Sect. 3.2.1, the state space is still too large. Additional actions are required to make it small enough to be fully explored. AnimUML has an option to consider that the elements of the system (objects) are necessarily faster than its environment (actors) who request them. This is called the reactive system hypothesis. This means that during exploration, the actors will be able to make progress only if there are no more transitions in the system to fire. Therefore, situations where the environment progresses without waiting for the system's reaction are avoided. For instance, this prevents actors from flooding object event pools. However, some actors, such as *User* here, should also have less priority than others, such as *SecureStorage*. AnimUML does not directly allow to prioritize one actor over others, and this can lead to state space explosion. One solution to this problem is to add specific guards on an actor transitions to force it to wait for other actors to progress. This was performed on *User*'s transitions in order to force the User to wait for the more reactive actors (*SecureStorage* and *SoCIPs*) to complete their operations. These guards can be seen in Fig. 9, which shows *User*'s state machine. They make use of the `EP_IS_EMPTY` introspection action. The way *User* is modeled is also important. As shown in Fig. 9, apart from the priority limitations with respect to other actors, *User* has few constraints and can send any message to the component at any time. Thus, all possible combinations of messages that it can send will be checked.

6.4 Formalizing and verifying properties

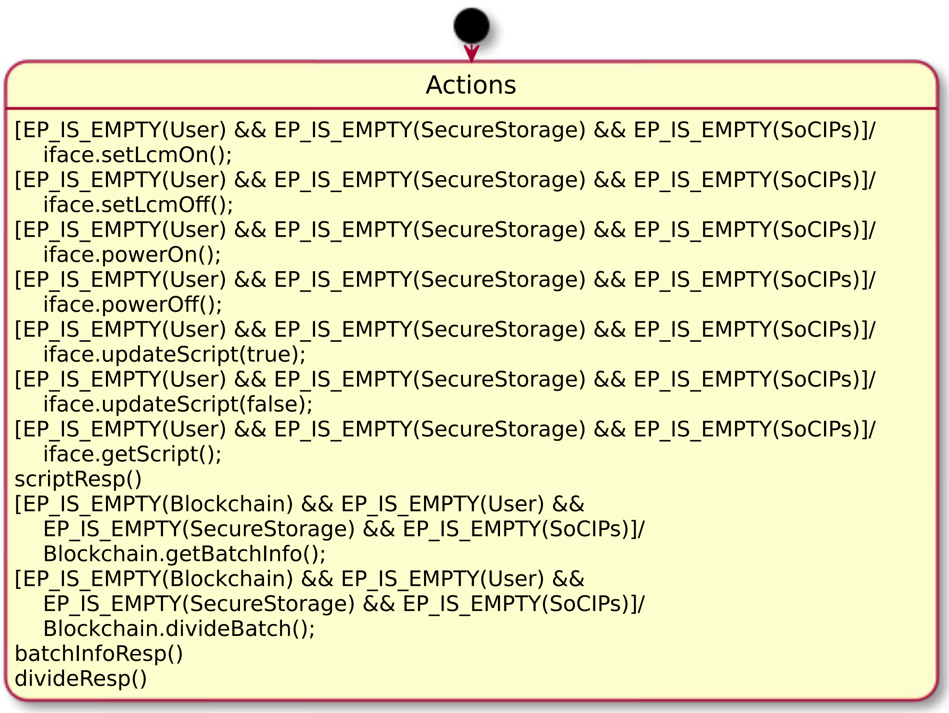
Translation from the informal expression of properties into LTL versions verifiable by AnimUML coupled to OBP2 was performed in three steps detailed in the following sections.

1. Define watch expressions (Sect. 6.4.1).
2. Rephrase the properties with the formulation of Dwyer's patterns [44] (Sect. 6.4.2).
3. Convert these into LTL properties while using the proper watch expressions (Sect. 6.4.3).

6.4.1 Defining watch expressions

Watch expressions represent the elements of the model that will be observed. They must correspond to parts of the informal property expressions. For instance, in the case of property **P4**: "When *User* sends *updateScript* or *getScript* to *iface* without sending *powerOff* afterward, if SoC is in LCMS mode then *User* must receive a *scriptResp* message." can be translated into: "When *ifaceHasScriptMessage* without *ifaceHasPowerOff*, if *ifaceIsInStatePOnLcmOn* then *userHasScriptResp* must be emitted." The same rewriting can be

Fig. 9 User state diagram



```
[EP_IS_EMPTY(User) && EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    iface.setLcmOn();
[EP_IS_EMPTY(User) && EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    iface.setLcmOff();
[EP_IS_EMPTY(User) && EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    iface.powerOn();
[EP_IS_EMPTY(User) && EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    iface.powerOff();
[EP_IS_EMPTY(User) && EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    iface.updateScript(true);
[EP_IS_EMPTY(User) && EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    iface.updateScript(false);
[EP_IS_EMPTY(User) && EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    iface.getScript();
scriptResp()
[EP_IS_EMPTY(Blockchain) && EP_IS_EMPTY(User) &&
    EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    Blockchain.getBatchInfo();
[EP_IS_EMPTY(Blockchain) && EP_IS_EMPTY(User) &&
    EP_IS_EMPTY(SecureStorage) && EP_IS_EMPTY(SoCIPs)]/
    Blockchain.divideBatch();
batchInfoResp()
divideResp()
```

performed on the other two properties **P5** and **P6**. The whole set of watch expressions is given in Table 1. Ideally, watch expressions should be expressed only with elements already in the black-box diagram. In our case watch expressions *ifaceIsInStatePOnLcmOn* and *managerHasValidUpdate* do not follow this rule. For the first one, the rule is not respected to avoid too much complexity. In practice, it is difficult to express that the system is on or off from an interface point of view in a model with asynchronous communication. It is often more convenient to express this by querying its internal state. For the other one, the reason comes from the fact that it is precisely when the component is in this particular state that we want to verify that the state has not been reached by abnormal means.

6.4.2 Rephrasing properties

LTL can be seen as reserved to users with good mathematical background because formulas can be relatively hard to understand. Dwyer's patterns [44] enable translating natural language expressions to their mathematical representations. In order to simplify the conversion, it was decided to reformulate the properties to match the way Dwyer's patterns are expressed. Properties **P4**, **P5**, and **P6** are reformulated into:

- **P4.** Globally ((*userHasScriptResp* or *chipHasPowerOff*) responds to *ifaceHasScriptMessage* After (Globally *ifaceIsInStatePOnLcmOn*)).
- **P5.** Globally *managerHasValidUpdate* respond to *memoryWrite* or *ifaceHasPowerOff*.
- **P5 bis.** Globally *managerHasValidUpdate* precedes to *memoryWrite*.
- **P6.** Globally (*sstrHasSetScript* implies *sstrHasSetConfig* before *IPsHasSetIP*).

It should be noted here that **P5** can be interpreted in two different ways. It depends on whether one interprets **P5** as a liveness property (if something is going to happen) or a safety property (if something is not going to happen). These two variations are relevant and will both be evaluated. Here, **P5** corresponds to the liveness property and **P5 bis** to the safety property.

After this rephrasing, the only thing left to do is to convert them.

6.4.3 Converting to LTL

Once the properties have been reformulated, their conversion to LTL is direct. This way it is possible to obtain the following LTL properties:

- **P4.** `[]([] ifaceIsInStatePOnLcmOn -> [] (ifaceHasScriptMessage -> <> (userHasScriptResp || ifaceHasPowerOff)))`
- **P5.** `[](managerHasValidUpdate -> <> (memoryWrite || ifaceHasPowerOff))`

Table 1 Definition of watch expressions

Name	AnimUML signification
ifaceHasPowerOff	`EP_CONTAINS(iface,powerOff)`
ifaceHasScriptMessage	`EP_CONTAINS(iface,updateScript)||EP\_CONTAINS(iface,getScript)`
userHasScriptResp	`EP_CONTAINS(User,scriptResp)`
ifaceIsInStatePOnLcmOn	`IS_IN_STATE(iface,iface.icInterface.pOnLcmOn)`
sstrHasSetScript	`EP_CONTAINS(SecureStorage,setScript)`
sstrHasSetConfig	`EP_CONTAINS(SecureStorage, setConfig)`
memoryWrite	`sstrHasSetScript || sstrHasSetConfig`
managerHasValidUpdate	`IS_IN_STATE(manager,manager.clcmMode.executingScript)`
IPsHasSetIP	`EP_CONTAINS(SoCIPs,setIPs)`

- **P5 bis.** `!memoryWrite W managerHasValid Update`
- **P6.** `[](<>IPsHasSetIP -> (sstrHasSetScript -> (!IPsHasSetIP U (sstrHasSetConfig && !IPsHasSetIP))) U IPsHasSetIP)`

6.5 Results

All these properties, when verified on the model with AnimUML coupled to OBP2, were found to be satisfied. It should be noticed that despite the use of Dwyer's patterns, it was required to make several iterations before obtaining a good formulation. Indeed, some properties initially failed, but by observing counter-examples, it became clear that it was the property rather than the design that was incorrectly formulated. However, we were not able to verify these properties with HugoRT (more details in Sect. 9.2.2).

7 Cryptographic protocol verification with verifpal

As explained in Sect. 4, Verifpal is used as well as ProVerif to verify the protocol model. Section 7.1 presents how the Verifpal language works. Section 7.2 describes how our protocol sequence diagram can be translated to Verifpal. Assumptions are listed in Sects. 7.3, 7.4 discusses property formalization and finally, Sect. 7.5 details verification results.

7.1 Verifpal language overview

Protocol modeling with Verifpal is performed as follows. First, an attacker type must be declared as passive or active. A passive attacker can read transmitted messages but cannot modify them. An active attacker is able to mutate messages and perform man-in-the-middle attacks. Then, initial cryptographic operations performed by each actor are declared. Actors are automatically declared when they are involved in the protocol. It is possible to generate constants, and to apply cryptographic primitives such as public key generation, hashing, symmetric and asymmetric encryption, message signing or concatenation and de-concatenation. After these operations, actors can message some of the resulting data to other actors. Actors who receive such messages can use available cryptographic primitives on the received data to check their validity or continue to describe the protocol flow. Once the protocol exchange is described, it is possible to check properties by expressing them as queries. Verifpal can test four types of queries.

- *Confidentiality* checks that no attacker is ever able to obtain data marked as confidential.
- *Authentication* checks that no attacker can ever forge specific messages by pretending they come from legitimate actors.
- *Freshness* checks that replay attacks, where old messages are resent by an attacker as if they were new, are not possible.
- *Unlinkabilities* checks that attackers cannot distinguish whether two values belong to the same user or to two different users.

7.2 Translating the sequence diagram

Switching from a sequence diagram to Verifpal code requires some work. Listing 1 lists the code for the Verifpal model excerpt that corresponds to the *Actual SoC ownership transfer* part of Fig. 8. This exchange represents the phase during which the new component owner (*Alice*) retrieves the data needed to update the component from *Blockchain*, and updates the component. The *getBatchInfo* query message is not represented in Verifpal. It is an unauthenticated request which allows *Alice* to get the *batchInfoResp* response. Because anyone can perform this query at any time, it does not play any cryptographic role, and can thus be omitted. The *batchInfoResp* response is represented. It contains the

Listing 1 Verifpal sample code that correspond to the SoC ownership transfer phase

```
// not modeled: getBatchInfo
// batchInfoResp:
Blockchain -> Alice: [newOwnershipProof]
principal Alice[
 _ = SIGNVERIF(pkMan, newBatchRootProof, newOwnershipProof)?
]
// updateScript:
Alice -> SoC: newOwnershipProof, salt, newBatchRoot, newBatchRootProof, pkAlice
principal SoC[
 // SoC checks exchange proof
 _ = ASSERT(HASH(HASH(uid0),HASH(salt)), newBatchRoot)?
 _ = SIGNVERIF(I_pkMan,newBatchRootProof, newOwnershipProof)?
 _ = SIGNVERIF(pkAlice, newBatchRoot, newBatchRootProof)?
]
//not modeled: scriptResp
```

relevant cryptographic elements for the verification. For the same reasons *scriptResp*, the response to *updateScript*, is not represented because it only acts as an acknowledgment. A Verifpal message takes the form: `A -> B: message`. Only arguments of messages are actually modeled, because only arguments, not message names, are used to perform cryptographic operations. Furthermore, it is not possible to reassign another value to a variable. Square brackets `[ ]` around an argument indicate that it is *guarded*. This means that it can be read by an active attacker but not tampered with. All parameters sent and received by *Blockchain* are guarded to model the immutable characteristic that it guarantees.

In addition to messages, the Verifpal model includes additional information. Indeed, it is necessary to model which cryptographic actions actors perform to construct messages. The checks (e.g., of signatures) made by actors on messages must also be modeled. Verifpal represents these actions in the format `principal Actor[ code ]`. Listing 1 contains two occurrences of these. First, *Alice* performs a signature verification (with `SIGNVERIF`) on the proof of ownership she has just received. This gives her the guarantee that this proof has been signed by the previous owner. Second, the SoC checks (with `ASSERT`) that it is actually included in *newBatchRoot*. It then verifies that this new configuration is signed by the new owner and countersigned by the old one. It already knows the previous owner's public key from a previous ownership transfer, or from initial provisioning for its first owner. Therefore, it can validate the proof and update its status.

7.3 Assumptions for model simplification

In order to make verification tractable, state space explosion must be contained by introducing some abstractions. First, the proposed batch system has been modeled as a set of two components only. There is the component sold to *Alice* with unique identifier `uid0`, and the one kept by Manufacturer with unique identifier `uidN`. By using only two components, we artificially simplify the computations required to model the accumulator and reduce the amount of data the attacker can manipulate. This simplification prevents state space explosion while being expressed with the same parameters as for N element. Secondly, Verifpal offers a phases system. Phases allow to compartmentalize the protocol. An attacker can only manipulate variables within the limits of the phases in which they are communicated. This reduces the number of attempts made by the attacker to manipulate each variable. Phases must be chosen carefully. Using too many of them can lead to missed potential attacks. In this project, four phases have been used. The first phase corresponds to SoC provisioning done by the manufacturer. The second one is for contract initialization in the blockchain. The third one encompasses the SoC ownership transfer exchanges. Internal component state update forms the fourth and last phase. These four phases correspond to independent steps in the protocol. It is therefore not necessary to allow the attacker to change the parameters of the previous phases.

7.4 Formalizing and verifying properties

As a reminder, the properties that can be verified on the protocol model are **P1**, **P2** and **P3**. Listing 2 shows the Verifpal code that specifies the corresponding queries.

Listing 2 Verifpal queries code

```
queries[
 confidentiality? uidN                                    //P1
 authentication? Blockchain -> Alice: batchRootProof      //P2
 authentication? Alice -> SoC: newOwnershipProof          //P3
 authentication? Alice -> SoC: newBatchRootProof          //P3
 ]
```

These queries can be understood in the following way.

- **P1.** can the attacker get the value of `uidN`?
- **P2.** when *Blockchain* sends *batchInfoResp*, is it possible for an attacker to change the *batchRootProof* parameter without *Alice* noticing?
- **P3.** is translated into two queries. The first one asks whether, when *Alice* sends *updateScript*, it is possible for an attacker to change the *newOwnershipProof* parameter without the SoC noticing it or not. The second one asks whether, when *Alice* sends *updateScript*, it is possible for

an attacker to change the *newBatchRootProof* parameter without the SoC noticing it or not.

These four queries and the three properties to be checked can be linked as described in the following. The first query makes sure that it is not possible for the attacker to be able to retrieve the value of `uidN`. It is used to prove that it is not possible for someone to identify elements that are not part of the exchange, as mandated by **P1**. The second query ensures that *batchInfoResp* messages cannot be corrupted by an attacker and be seen as a valid message. This corresponds to the definition of the **P2**. It is important to note that this property is mainly fulfilled by the fact that the message is retrieved from a blockchain. As our simplifying assumptions make us consider this kind of message as intrinsically unmodifiable, this query is mostly a sanity check. The last two queries, when combined, verify **P3**. The third query checks that *newOwnershipProof* cannot be modified. This parameter is used by the *update* message to guarantee to the SoC that the ownership modification is valid. An attacker able to forge this parameter could then create fake ownership modifications or load illegitimate configurations. The last query goes hand in hand with the third. It consists in checking that *newBatchRootProof*, which is the signed content of *newOwnershipProof*, is itself signed by the new owner.

7.5 Results

The verification results show that **P1** and **P2** requests passed. This means that if the queries properly match the properties, and the exploration performed by Verifpal is sufficiently exhaustive, then the protocol model conforms to its specifications. In the case of **P3**, the latest version of Verifpal (0.27 as of writing this paper) corrects a bug and now, unlike with version 0.26, it prints a trace of a potential attack in both queries dedicated to **P3**. After analysis of the traces given by the tool, it appears that this attack is a false positive. Indeed, in the Verifpal model, there are several successive checks. It seems that when earlier checks should stop the iteration, Verifpal continues exploring it. Faced with these findings, the results obtained with ProVerif should help to clarify the situation, so that the results of the two tools can be compared.

8 Cryptographic protocol verification with ProVerif

In order to compare the results obtain with Verifpal with a more well-known cryptographic verifier, ProVerif is used to perform the same protocol verification of the model. This section is ordered in the same way as Sect. 7. Thus, the way ProVerif works is presented in Sect. 8.1. Section 8.3 explains how to adapt our model in order to be able to verify it. Section 8.2 lists the simplification assumptions made when converting the model into the ProVerif language. Finally, Sect. 8.4 explains how properties to check can be expressed, and Sect. 8.5 presents the verification results.

8.1 ProVerif language overview

The ProVerif modeling language[3] is less straightforward than the Verifpal one. In ProVerif, protocols are modeled as processes that send messages over channels. They use different user-defined primitives that are used to build, cipher, and decipher messages. In general, a ProVerif file is organized in four parts: (1) channel definition, (2) types and primitive declarations, (3) properties to check, and (4) processes declarations.

First of all, ProVerif uses channels to transfer messages between processes. These channels can be public or private, depending on whether one wants the attacker to be able to access them or not. Processes can use the *out(channel, data)* and *in(channel, data)* functions to send or retrieve data on a given channel.

After channel declarations, types and functions used in the protocol must be defined. Specific data such as private or public keys, host identifiers, typed data, can be declared as types. Cryptographic primitives are constructed with the use of constructors and destructors. Listing 3 presents the creation of an asymmetric encryption primitive with this technique. In this example, a private key (*skey*) type and a public key (*pkey*) type are declared with the type system. The *pk* constructor allows the construction of the public key from the private key. *encrypt* constructs an encrypted message of type *bitstring* (a standard type in ProVerif). It takes for arguments the clear message of type *bitstring* and the public key of type *pkey*. The destructor symbolized by "*reduc forall*" allows to retrieve the message *x* using the *decrypt* function, which needs the ciphered message (*encrypt(x,pk(y))*) and the secret key (*y*) to recover the clear message *x*. This system of constructors and destructors allows for the definition of many cryptographic primitives such as hashing, symmetric encryption, and authentication tags.

Once the channels and the cryptographic primitives are specified, the properties to be checked over the protocol can be defined. ProVerif can check secrecy properties (secrets are not accessible to the attacker) and even strong secrecy (the attacker cannot see if a secret changes over time, similar to unlinkability in Verifpal, but not used in the present case study). Authentication, and more generally "correspondence" properties, are also possible (the attacker cannot modify a message without being detected, which corresponds to authentication and freshness properties in

[3] More information and tutorial are available in the ProVerif manual [45].

Listing 3 ProVerif type and function definition example. example taken from [45]

```
type pkey (* Public key type *)
type skey (* Secret key type *)

fun pk(skey) : pkey.
fun encrypt(bitstring, pkey) : bitstring.
reduc forall x: bitstring, y: skey;
        decrypt(encrypt(x,pk(y)),y) = x.
```

Verifpal). Finally, ProVerif is able to prove equivalence between two processes (they cannot be distinguished) [46]. More details on how to write the properties will be given in Sect. 8.4.

Listing 4 ProVerif extract of the SoC process

```
let processSoC(pkMan: pkey) =
  (*Secure provisioning*)
  in(sc, (skSoC: skey, pkSoC: pkey, uidx1 : id,
     Hroot: bitstring));
  (* Not modeled: reception of getScript message*)
  if pkSoC = pk(skSoC) then
  let signedUidx = sign(id_to_bitstring(uidx1), skSoC) in
  out(c, signedUidx); (*Correspond to scriptResp*)
  (*The code continues...*)
```

The last part of a ProVerif script is composed of the actor processes and the main process, which launches the whole system. Listing 4 presents the first lines of the process that represents the behavior of “SoC" as an example. The definition of the process starts with “*let proccessName()*" and finishes with a period. The presented code shows the provisioning, and the first *getScript* received by the SoC from Alice.

More precisely, it shows:

- The reception of the provisioning data through a secured channel named *sc*.
- The verification of the integrity of the private and public keys.
- The construction and the sending of the message on the *c* channel (in response to the non-modeled *getScript* from Alice).

8.2 Model simplification assumptions

In order to facilitate designing the ProVerif model, some simplification has been performed. Firstly, the assumptions made for Verifpal are also made here. Secondly, the way the blockchain is modeled is also simplified. Indeed, in the original sequence diagram, as well as in the architecture diagram, the blockchain is considered as an actor with a behavior that corresponds to the decentralized application that runs on it. Conte de leon et al. [47], present the emergent and desired properties in the case of a blockchain. The properties of a blockchain can be summarized as being a decentralized register in which the registered data are immutable provided that the majority of the contributors are honest. The blockchain’s role in the case of the protocol is to transmit data with the guarantee that there will be consensus among all the application users and nobody will be able to modify it. It is therefore possible to represent the blockchain in ProVerif as a channel with public messages that are impossible to tamper with. To do this in ProVerif, a private channel named `bc` is dedicated to messages that pass through the blockchain. Messages are sent directly from the actor who sends them to the blockchain to the actor who uses them in the protocol. In order to make this data visible to the attacker, the other actors send duplicate blockchain data on a public channel. Finally, the actors compare the data received on the public channel with the ones received on the private channel to prevent the attacker from modifying them.

8.3 Turning a sequence diagram into a ProVerif model

The way a protocol model differs in comparison to a sequence diagram is important. The conversion from a sequence to a set of processes is not straightforward and prone to errors. Moreover, ProVerif does not indicate if the protocol works properly. Worse, a protocol that is not functional, and does not properly transmit messages, may not violate security properties. Thus, it may be considered as secure, although it does not work. To simplify this process and limit these problems, we propose to use “intermediate UML diagrams."

8.3.1 Modeling actor behavior in UML

Figure 10 presents such a diagram. On it are listed all the actors involved in the protocol, as well as the messages that they can send to each other. The goal is then to determine the behavior of each actor in the form of a state diagram, in order to reproduce the protocol sequence diagram. For this process, a tool that is able to animate and debug UML models, such as AnimUML, is helpful. After asserting the architecture state machines are able to reproduce the sequence diagram, they can be translated into a ProVerif process.

8.3.2 Converting intermediate diagrams to ProVerif

Figure 11 gives an example of a working state machine for the SoC behavior. On this state diagram, it is possible to see the first state, the provisioning step, as well as the first request

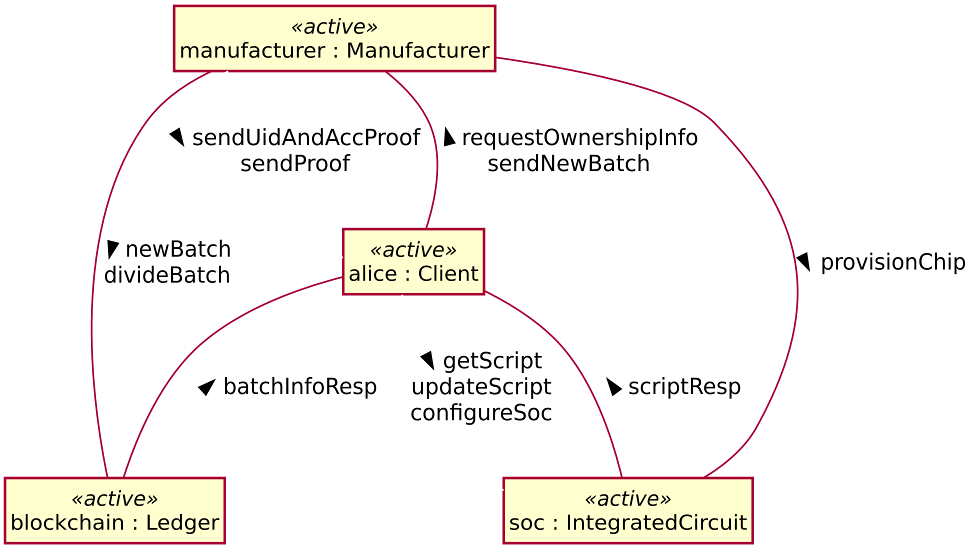


Fig. 10 Intermediate interaction diagram in order to go from the protocol sequence diagram to process for ProVerif

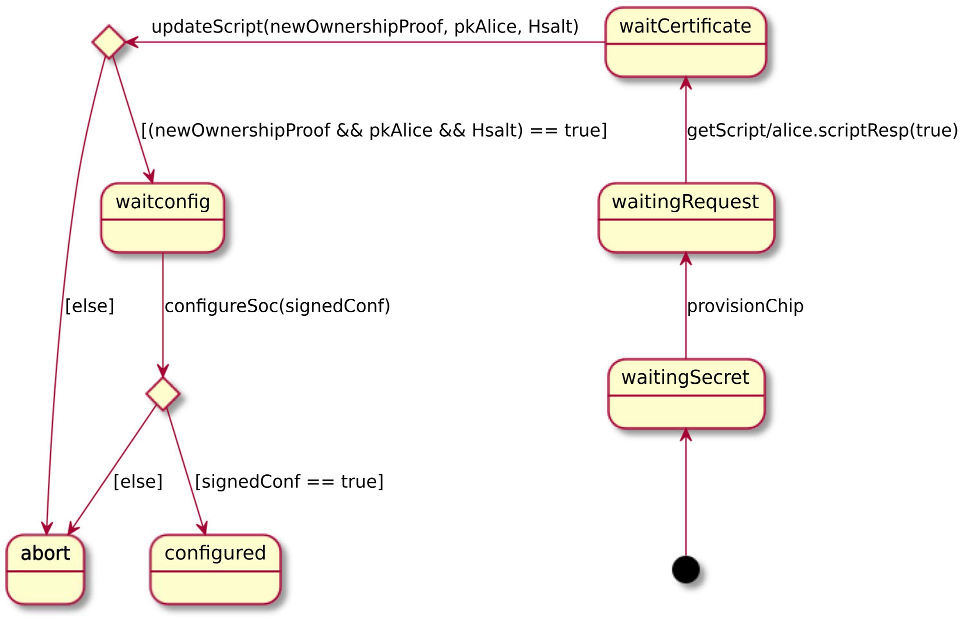


Fig. 11 SoC behavior in the intermediate interaction diagram. This state diagram can be translated to a ProVerif process

from Alice to the SoC, which consists of the SoC sending its signed UID. Converting this diagram into a ProVerif model can be done in the following way:

- The triggers correspond to the reception of data by the process and can therefore be translated into the *in* function of ProVerif.
- The effects are equivalent to sending data and therefore correspond to *out* in ProVerif.
- Guards and choice states can be transcribed as an “if then else” conditional function.

Based on these principles, Listing 4 presented earlier shows part of the translation of the SoC state diagram. Finally, it is possible to generate a trace in ProVerif to check if the modeled protocol is running properly. Indeed, ProVerif can evaluate the non-reachability of events within a model. This can be done by using a query with the form: “`query param:type; event(anEvent(param)).` ” with *anEvent* the name of the event to reach. If ProVerif can reach the event, it will return that the query does not hold and give a trace that proves that the event is reachable. It is then easy to analyze the trace and check that this corresponds to the correct protocol exchange.

8.4 Verifying properties

In our use case, the three properties to check with the protocol model are privacy and authentication properties. ProVerif can verify such properties. The declaration of the properties in ProVerif is done the following way.

8.4.1 Confidentiality properties

In the case of confidentiality properties, the user must declare the namespace of the data that must remain secret. The data are declared private and the keyword *not attacker(data-name)* must be used to clearly specify that the attacker must not know the data. Finally, the keywords *query attacker(data-name)* indicate to ProVerif that we want to check that the attacker is not able to recover the data (here *data-name*). Listing 5 shows how **P1** can be expressed in ProVerif. It can be noticed that this way of defining such a property is similar to what is done in Verifpal.

Listing 5 ProVerif confidentiality property example

```
(* Secrecy assumptions *)
free uidN : id [private].
not attacker(uidN).
(* Queries, correspond to P1*)
query attacker(uidN).
```

8.4.2 Authentication properties

The way to obtain authentication properties in ProVerif is less straightforward. It is based on an event system. Listing 6 shows how **P2** and **P3** can be expressed.

Listing 6 ProVerif authentication propertie example

```
(* Events declarations *)
event ManSendBCCertif(pkey, pkey).
event AliceRecBRProof(pkey, pkey).
event ManSendCertif(pkey, bitstring).
event socAcceptCertif(pkey, bitstring).

(* Authentication, Correspond to P2 *)
query pkA: pkey, pkM: pkey;
   inj—event(AliceRecBRProof(pkM,pkA))
        ==> inj—event(ManSendBCCertif(pkM,pkA)).
(* Correspond to P3 *)
query pkA: pkey, pkM: pkey, cert: bitstring;
   inj—event(socAcceptCertif(pkA,cert))
        ==> inj—event(ManSendCertif(pkM,cert)).
```

In this case, the events needed to express the property are defined first. Those events are called in the process at key moments of the protocol. For instance, the event *ManSendBCCertif* is sent just before Manufacturer sends the *newOwnershipProof* into the blockchain. In a similar way, *AliceRecBRProof* is emitted after Alice receives the *newOwnershipProof* from the blockchain. Each event contains types as parameters. This allows to take into account part of the context during which the event occurs. To define an authentication property, a query must be written in the form: "`query event(eventName1(paramsType)) ==> event(eventName2(paramsType)).`" with *eventName* 1 and 2, two specific events. This line of code can be interpreted as: "If event *eventName1* occurs then event *eventName2* necessarily occurs before." If an attacker is able to make one process emit an event without the other process emitting the other event before, this proves that the attacker is able to impersonate that process. Listing 6 is expressed this way. In the case of **P2**, we check if an attacker is able to tamper with messages that came from the blockchain. For **P3**, the ability of the attacker to forge *updateScript* messages is verified. Listing 6 makes the use of the *inj-event* keyword instead of *event*. *inj-event* stands for injective event. Injective events are more powerful than classical events. In the case of an injective event, ProVerif will also check that for each occurrence of the first event, there is a distinct earlier occurrence of the other event. With an injective event, it is possible to detect things like replay attacks. These are stronger properties than just authentication.

8.5 Results

Two models have been designed: one that abstracts provisioning and one that does not. The idea of abstracting comes from the fact that provisioning is considered as safe by assumption. Whatever the model, **P1** and **P2** are considered true. For **P3** the results are more complex. In the first model, the result is false and the trace seems to show that a replay attack and even an authentication attack are possible. This looks problematic because these attacks only exist because multiple SoC processes have the same UID, which in reality is not possible. The second model corrects this problem by modeling provisioning and thus allowing all SoC instances to have their own uid. This model does not find any attacks on **P3**, but is not able to prove that there are none. However, ProVerif is able to prove that it is not possible for the attacker to forge these messages. One additional property has been tested (it can be considered as a **P3.5** property). This one aims to ensure that the attacker is not able to forge configurations for the SoC. In the first model, the function could be forged by the attacker because the certificate receiver was not specified, and therefore the message could be reused. By modifying this in the second model, the results are better, but the absence of replay attack cannot be proven either. This is logical since **P3.5** can only be proven if **P3** is proven. It is important to note that only the properties with injective events are not proven. The case of classical events is shown to be true by ProVerif for both **P3** and **P3.5**.

9 Feedback

The specification, modeling, and formal verification of an LCMS component have allowed us to gain insights into the experience of modeling LCMSs. This was achieved with the help of tools adapted to engineers and UML users who are not necessarily formal verification specialists, shedding light on the interest of this approach. This section presents our feedback on using Verifpal, ProVerif, AnimUML and HugoRT to verify a model that consists of both a cryptographic protocol and a hardware architecture built around an LCMS component. Section 9.1 starts with the modeling process, and Sect. 9.2 is dedicated to the tools.

9.1 Modeling process

The first step was to survey existing tools, from which we realized that no single tool would be able to verify both the cryptographic protocol and the architecture. Originally, it was not obvious to go from UML to a protocol verification tool. This solution emerged when it was realized how complex it would be to model a verifiable cryptographic protocol with AnimUML or HugoRT. Because they are general-purpose modeling tools, all cryptographic support would have had to be manually modeled, with the risk of not being able to optimize enough to contain state space explosion. Tools specifically designed for this task, such as Verifpal or ProVerif, have dedicated algorithms and features not found in UML checkers, to the authors' knowledge. Separating the model into a cryptographic protocol model, and an architecture model was the next step. While it is not impossible to attempt to verify cryptographic protocols with tools like AnimUML, attempting to validate everything in a single model may make the task more time consuming and difficult. The following sections discuss models creation.

9.1.1 Architecture model

The decision to use UML to model the architecture and behavior of our hardware component is related to several factors. Firstly, even if UML is primarily designed for software development, it is also possible to use it to design hardware [48]. In our situation, the behavioral nature of our hardware description implies that the model can be either a software running on a secure microcontroller or a dedicated circuit. Therefore, the choice between a full hardware solution, and a software-hardware hybrid one can be taken at a later time, offering more flexibility. Secondly, UML is a language that is relatively well-known by engineers. Finally, the fact that AnimUML is developed by one of the authors of the present paper obviously impacted our choice. The main benefit of using AnimUML for model checking comes from the fact that the tool allows both design and verification from a single user interface. Being able to animate early design models was also useful in improving our model.

9.1.2 Protocol model

The translation from a sequence diagram describing a security protocol to a Verifpal model required some adaptations. A Verifpal model needs more details to model the protocol. The same thing is true for ProVerif. In the original sequence diagram, only the messages sent between participants are visible. Verifpal and ProVerif also need to know how the participants generate the parameters that are found in these messages. Protocol modeling as a sequence diagram is therefore not complete enough. On the other hand, the Verifpal and ProVerif models provide an accurate picture of the protocol behavior. However, the exchange of messages is limited to the strict minimum. Only messages that are directly relevant to protocol security evaluation appear. Moreover, only message parameters appear, not their names. In a way, the sequence diagram and the Verifpal code or the ProVerif model are complementary in modeling the protocol. We would have liked to try using an approach that supports direct sequence diagram annotation, such as [33], but we could not find the corresponding tools.

9.2 Tools

The following sections present our feedback on the AnimUML, HugoRT, Verifpal and ProVerif tools. The models were run on a laptop with a 2.6 GHz Intel Core i5-1145G7 processor and 16 GB of RAM. The Verifpal model check takes about 2 h while the verification of ProVerif model is done in a few seconds. In the case of AnimUML, a property takes less than 10 s to be checked with AnimUML and OBP2 running on the same machine. As for HugoRT, we were not able to perform the verification, as is explained in Sect. 9.2.2.

9.2.1 AnimUML

Before talking about our feedback on AnimUML, it is important to remember that one of the authors is at the origin of the tool. Even if the person in charge of modeling the LCMS is not directly involved in the development of the tool, the opinions presented in this paper are necessarily biased. The most interesting thing with AnimUML is that in the case of the architectural model, it was possible to do everything with a single tool, from architectural specification to LTL property verification passing through model debugging. Among the main highlights, there is the generation of sequence diagrams during a manual simulation, which is a relatively useful way to manually test the compliance of a model with its specifications. Being able to use multiverse debugging [49] with AnimUML is also useful, for instance to automatically find a configuration in which a given watch expression is true. The display of verification counter-examples as sequence diagrams in terms of the design model also significantly helps identifying the reasons for the failure. However, AnimUML also has some negative points: information about model analysis is not yet fully documented. Consequently, it is sometimes hard to debug a model unless first having been trained to know what functionalities are available. For instance, there is a heat map functionality that makes it possible to check how often a state machine's transitions have been explored. This is a useful way to detect if a model enters all states. Another possibility is to perform *depth-first* exploration rather than *breadth-first*, which allows to detect different problems while debugging. We can also regret that AnimUML does not offer more fine-grained scheduling control than the reactive system hypothesis yet. Indeed, in order to make state space exploration tractable, it is possible to consider the system as faster than its environment. This means that actor transitions will only be triggered when the system itself can no longer trigger its own. This functionality is useful but lacks precision, and is insufficient in our case. It is nevertheless possible to encode priorities into environment transition guards as shown in Fig. 9. To conclude, like Verifpal, AnimUML is a relatively recent tool. The available options on its graphical interface are numerous but not necessarily straightforward. Moreover, its graphical interface can still be improved.

9.2.2 HugoRT

The three main advantages of HugoRT that we have identified with respect to AnimUML are the following:

1. **Complex model verification.** Because HugoRT delegates verification to the SPIN and UPPAAL model checkers, it benefits from their capacity to explore relatively large state spaces, compared to what can be achieved with AnimUML.
2. **Timed verification.** With UPPAAL, it is possible to perform timed verification. This is invaluable when one needs it, although our case study does not in its present form.
3. **Sequence diagram properties.** Although we have not tried it, HugoRT offers the possibility to use sequence diagrams in order to define properties, in the form of observers.

However, although we have been able to launch SPIN verifications, we have not been able to verify our model with HugoRT. Indeed, it does not behave as expected: some parts of the model that are explored with AnimUML remain unexplored with SPIN. This is likely something that could be debugged, given more time, but we have already spent roughly as much time with HugoRT as we did with AnimUML. This is not a drawback of HugoRT, but it shows that working with different tools that encode slightly different UML semantics is hard.

The four main limitations of HugoRT that we have identified with respect to AnimUML are the following:

1. **Semantic Gap.** It is not possible to simulate or debug models directly at the UML level. We therefore found it necessary to understand SPIN, plus how the UML model is translated. The translations of the traces back to UML, although extremely useful, have some missing elements (e.g., some message sources), are relatively verbose, and do not support visualization as UML (e.g., sequence) diagrams.
2. **No partial modeling.** HugoRT does not support partial models, which was especially useful while creating the AnimUML model. Although we have not tried to create the HugoRT model from scratch, we expect that this would have been significantly harder than with AnimUML.
3. **Limited observation language.** HugoRT's "observation" language (what one writes LTL atomic properties in) is more limited than AnimUML's: one can query an object's current state (with inState), but not the contents of its event pool. Moreover, HugoRT's observation language cannot be used in transition guards, which is something we use when modeling the environment as active actors in AnimUML, as in Fig. 9.
4. **Priorities.** HugoRT does not support giving a lower priority to the environment, which is something we use in AnimUML (via the reactive system hypothesis) to "delay" state space explosion.

Remark: we nonetheless managed to encode the last two points (environment transition guards that observe other objects' event pools, and priorities) by editing the SPIN model generated by HugoRT.

In conclusion: using HugoRT can bring some benefits, but it requires additional expertise beyond what we assumed engineers had. Obviously, the validity of this section's contents is impacted by our bias toward AnimUML, and by the fact that we were not able to verify our model with HugoRT.

9.2.3 Verifpal

The main quality of Verifpal is the one that made us decide to use it, namely its relatively user-friendly language. Indeed, this tool is designed to easily obtain a working model and verify properties [21]. In addition to the relative simplicity of this language (compared to, e.g., ProVerif) for both modeling and property declaration, the tool also has clear documentation. However, Verifpal turns out to be inconvenient on several points. Firstly, the impossibility of designing cryptographic primitives can complicate, or even stop, the possibility to model some protocols. For instance, the original protocol of our LCMS system was supposed to use an *RSA Accumulator* [50]. Accumulators are cryptographic techniques that are *one way membership hash functions*. For a given set of elements, they produce a condensate as well as membership and nonmembership proofs for any element of the set. In order to still attempt to model the protocol, it was decided to design a new version that uses Merkle trees instead [51]. Merkle trees are also capable of producing a condensate as well as a set of evidence that prove the presence (but not the absence) of the elements inside. This other primitive is also not available on Verifpal, but it is possible to reconstruct a simplified version from the hash and concatenation primitives. This compromise was made because being able to verify the protocol is critical: more features should not threaten security. Secondly, Verifpal offers a conversion feature from its models into ProVerif ones. This is an interesting capability, because moving from a Verifpal model to a ProVerif model, even if incomplete, could be a good way to access ProVerif's features with reduced effort. In addition, ProVerif's verification methods are more tested, and can work on multiple protocol usage scenarios, whereas Verifpal focuses on a single one. However, in our case, this conversion tool produces a particularly verbose and complex non-working code. Therefore it is not usable as it is. Finally, Veripal is still a relatively recent tool and is currently in beta version. Even if the queries pass, it is not necessarily a complete proof that there is no flaw. The results obtained with Verifpal leave us doubtful. Indeed, for **P1** and **P2** they correspond to those of ProVerif. But for **P3**, the attack found by Verifpal was complex to analyze. Indeed the trace is unclear, and its interpretation leads us to believe that it is a false positive.

9.2.4 ProVerif

Since ProVerif has the same role as Verifpal, we rely on the latter to identify the qualities of the former. According to its designers, ProVerif's ability to reason with reachability, correspondences, and observational equivalence is sound. When ProVerif says that a property is satisfied, then the model really does guarantee that property holds. However, ProVerif is not complete and may not be capable of proving a property that holds. The results of ProVerif are easier to analyze than those of Verifpal. ProVerif gives a lot of details and the HTML interface available with the tool allows to have a graphical trace of potential attacks. ProVerif also has detailed and clear documentation. but it also has disadvantages. Its learning curve is higher and it is more error prone in comparison with Verifpal. The risk of error comes from the fact that if there is a problem in the model, it will not be obvious, and the properties to be checked will be considered as true. For instance, on the first version of the second ProVerif model, a transmission channel error was made by mistake. The receiver was listening to a different channel than the transmitter. This prevented the whole protocol from running because the "in" function used to retrieve the data is blocked. This error could be spotted by first using the intermediate UML diagram. Then, it is possible to check directly with ProVerif that the protocol is executed correctly by using a special property in ProVerif that fails if an event at the end of the protocol is produced. ProVerif will be able to give a trace of this "fail" and it is then possible to check that this trace corresponds to the protocol sequence diagram.

Using UML and AnimUML to prepare the conversion of the specifications diagram to the ProVerif model helps to limit errors. Firstly, the intermediate interaction diagram can be animated to prove that it is able to reproduce the sequence diagram. Secondly, more liveness properties can be verified with AnimUML. This was not done as part of this study, but is feasible.

In our case, ProVerif has not been able to prove that property **P3** is true, but has no evidence that it is false either. At least the non-injective property is considered true, which means that in a strict sense **P3** holds (messages cannot be forged), but we don't have the proof that the protocol is insensitive to replay attacks. Modeling the protocol in another way might give a clearer answer, but more work is needed to confirm this statement.

Finally, it should be possible to model an RSA accumulator with ProVerif. However, we decided to keep using a Merkel tree instead, to avoid adding to the differences between the Verifpal and ProVerif models.

10 Case study limitations

There are a number of limitations to our proposed approach and the results obtained:

1. **Industrial stakeholder validation.** Firstly, the main threat to validity of the work presented in this paper is that, although this is an industrial case study, we are only in the early phases of the project. Therefore, the techniques presented in this paper have only been validated by one industrial stakeholder: the third author, as an industrial client. However, as a direct consequence of the work presented here, AnimUML has already been applied to another project at STMicroelectronics, and more model verification applications are planned.
2. **Verification compositionality.** Our hypothesis of dividing the model into the two architecture and cryptographic protocol aspects was determined intuitively, given that available verification tools generally focus on only one aspect. Modeling the system separately from an architectural and from a cryptographic point of views could be insufficient. Moreover, for our case study, it would also likely be useful to model the contents of smart contracts, and formally verify them to guarantee their correctness.
3. **Model equivalence.** Another issue is that the different models we created are not necessarily equivalent. This is actually the reason why we were not able to verify our

model with HugoRT. As explained in Sect. 9.2.2, solving this issue is not trivial because of differing UML semantics. The difficulty of working with multiple UML semantics is actually one of the lessons learned in this work. Regarding cryptographic protocol models, Verifpal and ProVerif work quite differently: a Verifpal model focuses on one scenario, whereas a ProVerif model can be more general. We decided to leverage this in the case study by creating a more general ProVerif model than the Verifpal one. In theory, Verifpal can generate equivalent ProVerif models, but we did not manage to get it to work on our case study, which is also one of the lessons learned in this work. However, in practice, only one architectural, and one protocol verification tools are necessary for a given project. Therefore, the only remaining issue to apply the methodology presented in this paper is the architecture and protocol model verification compositionality (see point 2).

4. **Smart contract verification.** At the time being, smart contracts are only partially modeled using sequence diagrams. This can be considered sufficient at the specification or initial design stages, given that the implementation is not yet defined. However, this means their correctness cannot be verified on the model at this stage.
5. **Early stage verification.** Another limitation is that the methodology presented in this article is limited to model verification. The transition to a prototype or functional system is not taken into account, and we assume that current industry methodologies will be applied. However, this does not formally guarantees a match between the verified models and the final system. Nevertheless, the proposed approach makes it possible to obtain better quality models, which should at least avoid some issues in the later development phases. Moreover, proposals have already been made to allow UML models to directly run on embedded systems, while ensuring that they are verified with the same semantics as the one used in production [26]. Such an approach also avoids the pitfall of working with multiple UML semantics.
6. **Types of properties.** Another limitation is that property checking is only expressed in LTL for the architectural model. However, other types of properties, such as those related to response times, might be desirable to verify. In our case, the targeted applications do not have strict time constraints, as the systems in which the LCMS is embedded must be in configuration mode (not production) while the LCMS is used. In cases where timing issues are more important, the use of a tool such as HugoRT could prove judicious, because it can translate UML models into UPPAAL, which can verify this type of properties. However, from our experience with HugoRT and SPIN, we expect that the combination of HugoRT and UPPAAL becomes even more complex to use, and requires significantly more expertise than just knowing UML.
7. **Required efforts.** We have not measured the extra efforts required to verify models compared to the global project implementation efforts. This is not a question on which we focused, but answering it would be useful.
8. **Reproduction.** In order to further refine the evaluation, the experiment should be reproduced by others, on the same and on other case studies, as well as with the same and other verification tools. This could also help with point 7 about measuring the additional efforts required to apply model verification, provided these reproductions take care to measure spent efforts.
9. **Informal to formal specification.** A final limitation can be identified by the difficulty of converting informal specifications into formal properties, whether for architecture or protocol models. For architecture models, the use of Dwyer's patterns helps, but the transition from English to formal properties suffers from the limitations of natural language, requiring interpretation.

11 Conclusion

This paper evaluates formal model verification tools on a case study. The studied system consists of a SoC-embedded secure element as well as a secure protocol that allows to exchange chip ownership rights as well as to update its configuration.

This work shows that it is possible to model SoC-based systems with modeling techniques close to those used for specification in industry while still being able to formulate and formally verify their properties.

In the case where the modeling tools were more difficult to use, it is possible to facilitate their use by performing intermediate operations. The idea of using different tools, such as Verifpal or ProVerif to check the cryptographic protocol, and AnimUML or HugoRT for the architectural aspects, gives the benefit of coming to a complete solution more quickly.

Verifpal's modeling language, similar to a textually specified sequence diagram, allows non-experts to obtain a protocol model more easily than with other similar tools at the cost of a lesser degree of soundness. ProVerif is more powerful and offers more possibilities than Verifpal, but it requires more training to be mastered. It is also more complicated than with Verifpal to get a model that works, which is why using an intermediate AnimUML model can help.

The fact that AnimUML offers the possibility to simulate a design model while it is still very incomplete is a useful advantage in order not to forget any elements and to facilitate the design phase.

HugoRT can convert UML to code for multiple well-known model checking tools. However, one needs a certain level of expertise in using these model checking tools to check properties effectively.

Finally, specification verification in the form of LTL properties, while not necessarily obvious, is greatly simplified by the use of Dwyer's patterns.

Acknowledgements The authors would like to thank Pr. Alexander Knapp for his guidance and support in using HugoRT.

References

1. Dachyar, M., Zagloel, T.Y.M., Saragih, L.R.: Knowledge growth and development: internet of things (IoT) research, 2006–2018. Heliyon **5**(8), 02264 (2019). https://doi.org/10.1016/j.heliyon.2019.e02264
2. Mont, O.K.: Clarifying the concept of product-service system. J. Clean. Prod. **10**(3), 237–245 (2002). https://doi.org/10.1016/S0959-6526(01)00039-7
3. Exner, K., Schnürmacher, C., Adolphy, S., Stark, R.: Proactive maintenance as success factor for use-oriented product-service systems. Procedia CIRP **64**, 330–335 (2017). https://doi.org/10.1016/j.procir.2017.03.024
4. Méré, M., Jouault, F., Pallardy, L., Perdriau, R.: Feedback on the formal verification of UML models in an industrial context. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems. MODELS '22, pp. 121–131. ACM, New York, NY, USA (2022). https://doi.org/10.1145/3550355.3552454
5. Islam, M.N., Kundu, S.: Remote device management via smart contracts. IEEE Trans. Consum. Electron. **68**, 38–46 (2021). https://doi.org/10.1109/TCE.2021.3139584
6. Skudlarek, J.P., Katsioulas, T., Chen, M.: A platform solution for secure supply-chain and chip life-cycle management. Computer **49**(8), 28–34 (2016). https://doi.org/10.1109/MC.2016.243
7. Méré, M., Jouault, F., Pallardy, L., Perdriau, R.: Trustworthy SoC reconfiguration aimed at product-service systems: a literature review. In: COINS Conference, pp. 1–6. IEEE Computer Society, Barcelona, Spain (2022). https://doi.org/10.1109/COINS54846.2022.9854965
8. Robson, N., Safran, J., Kothandaraman, C., Cestero, A., Chen, X., Rajeevakumar, R., Leslie, A., Moy, D., Kirihata, T., Iyer, S.: Electrically programmable fuse (eFUSE): from memory redundancy to autonomic chips. In: 2007 IEEE Custom Integrated Circuits Conference, pp. 799–804 (2007). https://doi.org/10.1109/CICC.2007.4405850 . ISSN: 2152-3630
9. Tanaka, K., Nakamura, S.: Storage system and data protection method therefor. Google Patents (2009)
10. Shepherd, C., Arfaoui, G., Gurulian, I., Lee, R.P., Markantonakis, K., Akram, R.N., Sauveron, D., Conchon, E.: Secure and trusted execution: past, present, and future - a critical review in the context of the internet of things and cyber-physical systems. In: 2016 IEEE Trustcom/BigDataSE/ISPA, pp. 168–177 (2016). https://doi.org/10.1109/TrustCom.2016.0060. ISSN: 2324-9013
11. Bhunia, S., Tehranipoor, M.: The Hardware Trojan War. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-68511-3
12. Mera Collantes, M.I., Garg, S.: Do not trust, verify: a verifiable hardware accelerator for matrix multiplication. IEEE Embed. Syst. Lett. **12**(3), 70–73 (2020). https://doi.org/10.1109/LES.2019.2953485
13. SGS Société Générale de Surveillance: SGS Brightsight (2023). https://www.brightsight.com/system-on-chip Accessed 2023-06-12
14. Hakak, S., Khan, W.Z., Gilkar, G.A., Assiri, B., Alazab, M., Bhattacharya, S., Reddy, G.T.: Recent advances in blockchain technology: a survey on applications and challenges. Int. J. Ad Hoc Ubiquitous Comput. **38**(1–3), 82–100 (2021)
15. Mohanta, B.K., Panda, S.S., Jena, D.: An overview of smart contract and use cases in blockchain technology. In: 2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), pp. 1–4. IEEE, Bengaluru, India (2018). https://doi.org/10.1109/ICCCNT.2018.8494045
16. Méré, M., Jouault, F., Pallardy, L., Perdriau, R.: Modeling trust relationships in blockchain applications: the case of reconfigurable systems-on-chip. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), pp. 1–8 (2022). https://doi.org/10.1109/QRS-C57518.2022.00020 . ISSN: 2693-9371
17. OMG: Unified Modeling Language (2017). https://www.omg.org/spec/UML/2.5.1/PDF
18. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri Nets I: Basic Models: Advances In Petri Nets, pp. 429–528. Springer, Berlin (1998). https://doi.org/10.1007/3-540-65306-6_21
19. Blanchet, B.: Automatic verification of security protocols in the symbolic model: the verifier ProVerif. In: Aldini, A., Lopez, J., Martinelli, F. (eds.) Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures, pp. 54–87. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10082-1_3
20. Cervesato, I., Durgin, N.A., Lincoln, P.D., Mitchell, J.C., Scedrov, A.: A meta-notation for protocol analysis. In: Proceedings of the 12th IEEE Computer Security Foundations Workshop, pp. 55–69. IEEE, Mordano, Italy (1999).https://doi.org/10.1109/CSFW.1999.779762 . ISSN: 1063-6900
21. Kobeissi, N., Nicolas, G., Tiwari, M.: Verifpal: cryptographic protocol analysis for the real world. Published: Cryptology ePrint Archive, Report 2019/971 (2019). https://ia.cr/2019/971
22. Jouault, F., Besnard, V., Calvar, T.L., Teodorov, C., Brun, M., Delatour, J.: Designing, animating, and verifying partial UML models. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '20, pp. 211–217. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3365438.3410967
23. Jouault, F., Besnard, V., Brun, M., Le Calvar, T., Chhel, F., Clavreul, M., Delatour, J., Méré, M., Pasquier, M., Teodorov, C.: Animuml: a practical tool for partial model animation and analysis. Sci. Comput. Program. **232**, 103050 (2024). https://doi.org/10.1016/j.scico.2023.103050
24. Knapp, A.: In: Haxthausen, A.E., Huang, W.-l., Roggenbach, M. (eds.) An Intermediate Language-Based Approach to Implementing and Verifying Communicating UML State Machines, pp. 289–307. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-40132-9_18
25. André, E., Liu, S., Liu, Y., Choppy, C., Sun, J., Dong, J.S.: Formalizing UML state machines for automated verification-a survey. ACM Comput. Surv. **55**(13s), 1–47 (2023). https://doi.org/10.1145/3579821
26. Besnard, V., Brun, M., Jouault, F., Teodorov, C., Dhaussy, P.: Unified LTL verification and embedded execution of UML models. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '18, pp. 112–122. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3239372.3239395
27. Roques, A.: PlantUML: Open-source tool that uses simple textual descriptions to draw UML diagrams (2022). http://plantuml.com/ Accessed 2022-04-25
28. Holzmann, G.J.: The spin model checker: primer and reference manual **1003** (2004)

29. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0 (2006)
30. Dewoprabowo, R., Arzaki, M., Rusmawati, Y.: Formal verification of divide and conquer key distribution protocol using ProVerif and TLA+. In: 2018 International Conference on Advanced Computer Science and Information Systems (ICACSIS), pp. 451–458 (2018). https://doi.org/10.1109/ICACSIS.2018.8618173. ISSN: 2330-4588
31. Latif, S., Rehman, A., Zafar, N.A.: Blockchain and IoT based formal model of smart waste management system using TLA+. In: 2019 International Conference on Frontiers of Information Technology (FIT), pp. 304–3045. IEEE, Islamabad, Pakistan (2019). https://doi.org/10.1109/FIT47737.2019.00064. ISSN: 2334-3141
32. Rocha, H., Ducasse, S.: Preliminary steps towards modeling blockchain oriented software. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 52–57. IEEE, Gothenburg Sweden (2018)
33. Koch, T., Dziwok, S., Holtmann, J., Bodden, E.: Scenario-based specification of security protocols and transformation to security model checkers. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS ’20, pp. 343–353. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3365438.3410946
34. Zhang, J., Yang, L., Gao, X., Tang, G., Zhang, J., Wang, Q.: Formal analysis of QUIC handshake protocol using symbolic model checking. IEEE Access **9**, 14836–14848 (2021). https://doi.org/10.1109/ACCESS.2021.3052578
35. Lauser, T., Zelle, D., Krauß, C.: Security analysis of automotive protocols. In: Computer Science in Cars Symposium. CSCS ’20, pp. 1–12. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3385958.3430482
36. Chen, X., Mallet, F., Liu, X.: Formally verifying sequence diagrams for safety critical systems. In: 2020 International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 217–224. IEEE, Hangzhou, China (2020). https://doi.org/10.1109/TASE49443.2020.00037
37. Csertan, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varro, D.: VIATRA - visual automated transformations for formal verification and validation of UML models. In: Proceedings 17th IEEE International Conference on Automated Software Engineering, pp. 267–270. IEEE, Edinburgh, UK (2002). https://doi.org/10.1109/ASE.2002.1115027 . ISSN: 1938-4300
38. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of uml/ocl models using constraint programming. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. ASE ’07, pp. 547–548. Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1321631.1321737
39. Glouche, Y., Genet, T., Heen, O., Courtay, O.: A security protocol animator tool for avispa. In: ARTIST-2 Workshop on Security of Embedded Systems, Pisa (Italy) (2006). http://people.irisa.fr/Thomas.Genet/Publications/papier_artist.pdf
40. Viganò, L.: Automated security protocol analysis with the AVISPA tool. Electron. Notes Theor. Comput. Sci. **155**, 61–86 (2006). https://doi.org/10.1016/j.entcs.2005.11.052
41. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1186–1189 (2019). https://doi.org/10.1109/ASE.2019.00133
42. Leid, A., Merwe, B., Visser, W.: Testing ethereum smart contracts: a comparison of symbolic analysis and fuzz testing tools. In: Conference of the South African Institute of Computer Scientists and Information Technologists 2020. SAICSIT ’20, pp. 35–43. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3410886.3410907
43. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pp. 46–57. IEEE, Providence, RI, USA (1977). https://doi.org/10.1109/SFCS.1977.32 . ISSN: 0272-5428
44. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002), pp. 411–420. IEEE, Los Angeles, CA, USA (1999). https://doi.org/10.1145/302405.302672 . ISSN: 0270-5257
45. Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M.: ProVerif 2.04: automatic cryptographic protocol verifier, user manual and tutorial (2021). https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf
46. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. J. Logic Algebraic Program. **75**(1), 3–51 (2008). https://doi.org/10.1016/j.jlap.2007.06.002
47. Leon, D., Stalick, A.Q., Jillepalli, A.A., Haney, M.A., Sheldon, F.T.: Blockchain: properties and misconceptions. Asia Pacific Journal of Innovation and Entrepreneurship **11**(3), 286–300 (2017). https://doi.org/10.1108/APJIE-12-2017-034. Publisher: Emerald Publishing Limited
48. Vanderperren, Y., Mueller, W., Dehaene, W.: UML for electronic systems design: a comprehensive overview. Des. Autom. Embed. Syst. **12**(4), 261–292 (2008). https://doi.org/10.1007/s10617-008-9028-9
49. Singh, R.G., Lopez, C.T., Marr, S., Boix, E.G., Scholliers, C.: Multiverse Debugging: non-deterministic debugging for non-deterministic programs (Artifact). Dagstuhl Artifacts Series **5**(2), 4–143 (2019). https://doi.org/10.4230/DARTS.5.2.4. Place: Dagstuhl, Germany Publisher: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
50. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019, pp. 561–586. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26948-7_20
51. Becker, G.: Merkle signature schemes, merkle trees and their cryptanalysis