Constructing security cases based on formal verification of security requirements in alloy

Marwa Zeroual, Brahim Hamid, Morayo Adedjouma, Jason Jaskolka

Constructing Security Cases Based on Formal Verification of Security Requirements in Alloy

Marwa ZEROUAL [⋆1,2], Brahim HAMID[2], Morayo ADEDJOUMA[1], and Jason JASKOLKA[3]

[1] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
{marwa.zeroual,morayo.adedjouma}@cea.fr
[2] IRIT, Université de Toulouse, CNRS, UT2 118 Route de Narbonne, 31062 Toulouse Cedex 9, France
brahim.hamid@irit.fr
[3] Carleton University, Ottawa, ON, Canada
jason.jaskolka@carleton.ca

Abstract. Assuring that security requirements have been met in design phases is less expensive compared with changes after system development. Security-critical systems deployment requires providing security cases demonstrating whether the design adequately incorporates the security requirements. Building arguments and generating evidence to support the claims of an assurance case is of utmost importance and should be done using a rigorous mathematical basis, namely formal methods. In this paper, we propose an approach that uses formal methods to construct security assurance cases. This approach takes a list of security requirements as input and generates security cases to assess their fulfillment. Furthermore, we define security argument patterns supported by the formal verification results presented using the GSN pattern notation. The overall approach is validated through a case study involving an autonomous drone.

Keywords: Formal methods · Assurance case · Argument pattern · Security requirements · Security case

1 Introduction

Security-critical systems are prone to failure and security violations. Vulnerabilities can be caused by many factors: poor requirement specifications, underestimating the threat, and malicious exploitation of all the above by an attacker. Consequently, security engineers are required to provide assurance cases containing evidence and arguments to prove the security property of the system. Assurance cases are bodies of evidence organized in structured arguments that justify specific claims about a system property hold [12]. When assurance cases aim to demonstrate the security of a system, they are known as security cases. Formal model-based security cases are the ones that contain a formal model from which evidence for the top-level claims is derived. Formal methods are applicable in specifying and verifying critical systems from various industrial domains[3]. Despite this, there are limitations to formalizing large-size systems, such

⋆ Corresponding author: marwa.zeroual@cea.fr

as ensuring that the program works correctly with the hardware and operating system and the complexity involved in creating formal definitions of semantics for language constructs and software system components [2]. Building security cases to document and demonstrate that a system design meets the primary security requirements (SRs) is challenging, especially because sufficient evidence is needed to support assurance claims and traceability for compliance checks. In this paper, we propose an approach for demonstrating the compliance of SRs at the design level in the form of security cases. First, we assume a complete list of SRs that are determined following a secure development methodology. Next, we formalize the system model and the SRs using Alloy language. Then, we verify the SRs compliance and the system model. Afterward, security cases are defined and supported by the formal verification results. This is followed by the derivation of re-utilizable security argument pattern.

The remainder of this paper is organized as follows. Section 2 presents some definitions and the illustrative example. Section 3 details the proposed approach. Section 4 presents the proposed security argument patterns and exemplifies their application. Section 5 reviews related works. Finally, Section 6 concludes with future work directions.

2 Background

2.1 Overview of Alloy

Alloy is a lightweight formal modeling language based on first-order relational logic. An Alloy model comprises a set of signatures, each defining a set of atoms. There are several ways to specify constraints in the model. One is to treat them as *facts* that should always hold. Another is to treat them as *predicates* defined in the form of parameterized formulas that can be used elsewhere and as *assertions* that are intended to follow from the facts of a model. The semantics of Alloy is defined using *instances*. An instance is a binding of values to variables (e.g., signatures, signatures fields). *A core instance* is an instance associated with the model's facts, and the implicit constraints in the signature *declarations*. We can instruct Alloy Analyzer to verify whether the property $prop$ of the system design holds, with the command: **check** $prop$ **for** n, which would exhaustively explore every model instance within n atoms typed by each signature. If the property does not hold, the analyzer generates a counterexample we can visualize. The absence of counterexamples guarantees that the property holds in the modeled system within the specified scope. As claimed in [7], most counterexamples are found in a reasonably small scope.

2.2 Goal Structuring Notation

There are several notations and existing tools for developing and documenting assurance cases, and the most popular of these is *Goal Structuring Notation (GSN)*. GSN is a graphical notation that can be used to visualize arguments that assure critical properties: safety, security, and resilience of systems, services, or organizations. This paper adopts the GSN pattern notation (an extension of core GSN [6]) to visualize and present the argument structure. A summary of the graphical elements of the GSN Pattern Notation is provided in Fig. 1.

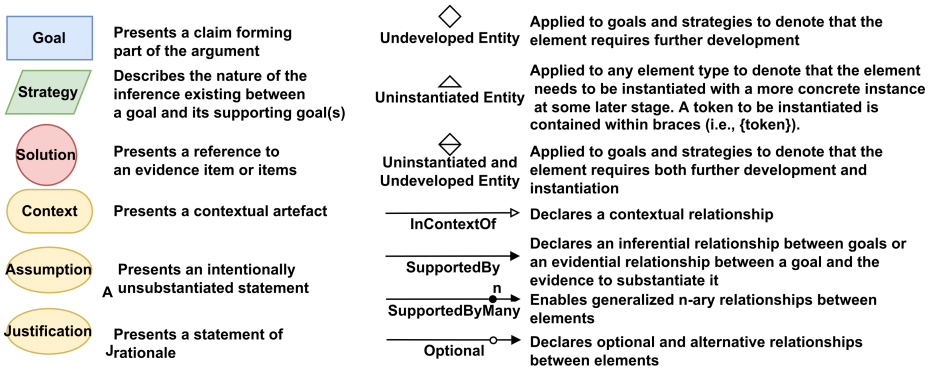

Fig. 1: Principal elements of the GSN Pattern Notation

2.3 Illustrative example

We illustrate our contributions using an example from ACAS Xu [8], a collision avoidance system for drones. The scenario involves two drones. One called the ”ownship.” The system is visualized in Fig. 2. The ownship's sensors collect data about other drones, which is then processed to determine an appropriate avoidance strategy. A planner generates a trajectory to navigate, and the actuator executes actions to follow the planned trajectory. The system's security is compromised if an attacker can modify messages sent to the processor, leading to decisions that result in a collision.

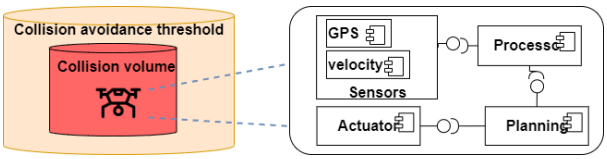

Fig. 2: Architecture of ACAS Xu

We extract from the work [1] a selection of SRs that impose requirements to design a secure ACAS Xu: $SR1$: The GPS messages are genuine and have not been intentionally altered, $SR2$: The processor must receive data only from valid sensors, and finally $SR3$: The system should employ mechanisms to mitigate unauthorized disclosure of the planning information.

3 General overview of the methodology

We propose constructing security cases from the SRs via their formal specification and verification in Alloy. Our approach runs in parallel to and is informed by the system development process. Each activity of our approach recovers artifacts from system

development and provides a security case to ensure and demonstrate the development and assurance activities.

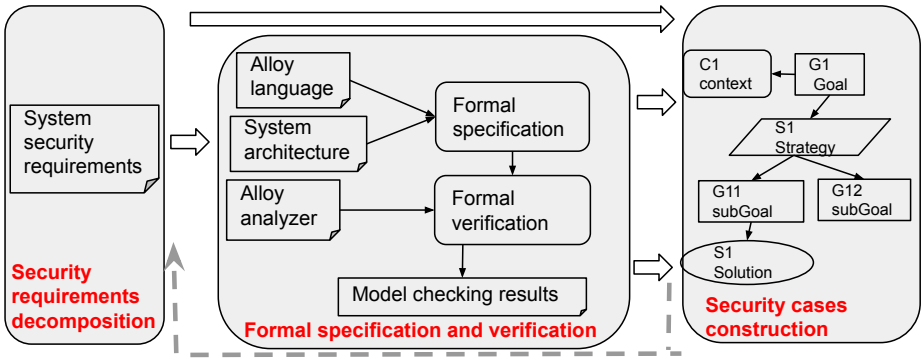

Fig. 3: Generating security cases from formal specification and analysis of SRs

As visualized in Fig. 3, we distinguish two main processes that influence the construction of security cases: 1 elicitation of security requirements, 2 representation of formalised system security requirements in formal models. Note that the construction of security cases is done by applying argument patterns reporting on the formal specification of the system and the SRs. If we can not build security cases, we revisit and improve the formalization of the requirements and the system. The grey shows this dependence dashed line in Fig. 3. The elicitation of the SRs is out of the scope of this paper.

3.1 Formal specification and analysis of SRs in Alloy

The formal method Alloy provides a structured way to specify the system model and its SRs. Subsequently, the formal verification provides strong evidence that a system is secure and meets its SRs. We formalize security objectives, as an implementation of SRs at the architecture level. The output of this process includes two main elements: the formal specification results (which consist of the system architecture model and the formalization of security objectives) and the formal verification results. These two elements correspond, respectively, to the first and second arrows from the formalization to the security cases construction top to bottom. To do so, we rely on works from [10] where, first, we adopt a metamodel to describe the architecture and capture 1 the functional architecture in terms of components and connectors, and 2 the behavioral aspects of the architectural elements. The modeling is done in the context of component-based architecture (CBA) while adopting the message passing paradigm for the communication. Formally speaking, a subset of these elements is specified in Listing 1.1.

```
1 sig Component {  uses: set Port}
2 sig Msg extends CommunicationStyle {   sent: one Tick,
3         received: Component -> lone Tick,  payload: one Payload   } ........
```

Listing 1.1: Declarations of some architecture elements

The metamodel enables us to describe the objectives: each objective is associated with a representative property that is defined as a Predicate constraint. After that, these properties are used in the realization of the objectives through model checking: we define assertions to check that the properties are not violated. Alloy Analyzer detects the violation of an objective due to the violation of the assertion by finding a counterexample. For example, the predicate *payloadIntegrity* showed in Listing 1.2 is a representative property associated with the payload integrity objective and the assertion *integrityNotHold* is used to detect the violation of this objective. The objective is defined as follows: "if some component c_2 is able to get the payload p of m then p is the accurate payload of m (has not been altered)." Formally speaking:

```
1  pred payloadIntegrity { all m:Msg, c:Component, p:Payload |
2                          E_get_pld[c,m,p]  implies once sent_with[m,p] }
3  assert integrityNotHold {
4    all c1,c2: Component, m:Msg| payloadIntegrity[c1,c2,m] }
```

Listing 1.2: Payload Integrity property

Alloy Analyzer finds a counterexample related to the violation of the *payloadIntegrity* property. Consequently, an appropriate security requirement is added to codify a security mechanism to satisfy the payload integrity objective. The *intg* requirement is defined as a predicate on the connector that guarantees that the sender's payload is the same payload received by the receiver through this connector.

```
1  pred Connector.intg {
2       all m:Msg, t:Tick, c:Component, d:Payload |
3       m in this.buffer.t  implies E_set_pld[c,m,d,t]
4       implies some a1:AllowedSetPld | a1.msg = m and a1.comp = c
5       all c:Component, t:Tick | E_inject[c,m,t]
6       implies some a1:AllowedSetPld | a1.msg = m and a1.comp = c }
7  assert integrityHold {  (all c:Connector | c.intg)
8       implies all c1, c2 : Component ,   d:Payload | payloadIntegrity[c1,c2,d] }
```

Listing 1.3: Payload Integrity solution

Finally, we check if the added requirement *intg* implies the satisfaction of the assertion violated previously. According to Alloy, no counterexample was found. The satisfaction of *intg* property allows the fulfillment of the corresponding security requirement to realize the payload integrity, as shown in Listing 1.3.

3.2 Construction of security cases

We build the security cases by applying argument patterns. We derive these patterns from the previous approach processes. First, the formal model development requires an argument about the well-definedness of the system model upon which we applied the formal method. It is the role of (P1). Building this argument is based on Alloy semantics provided in 2.1 and which shows the necessary conditions for a model to be well-defined.

Moreover, the formal methods involve building security cases by providing evidence elements and guiding the arguing process according to the formal language elements used for the property specification as shown in (P2). The strategies used are mainly inspired by the formal specification and verification of the requirements. Regarding the evidence elements, we use the results of the formal verification. In summary, the outputs

of the approach provide reusable security argument patterns: an argument pattern for the well-definedness of the system model (P1), and an argument pattern for the satisfaction of SR (P2). The following section will present these three argument patterns in detail.

4 Security argument patterns

This section presents the argument patterns derived from the approach presented in section 3 and their application on the case study presented in section 2.3.

4.1 Pattern for the well-definedness of the system model (P1)

The goal of this pattern in Fig. 6 is to claim the well-definedness of the system model according to Alloy language semantics. A consistent model has at least one instance that resolves all the facts and the declarations. The system model describes assumptions about the world in which a system operates, requirements that the system is to achieve, and a design to meet those requirements. The root claim in (G0) resumes the main goal of the pattern in the context (C0, C1). According to Alloy language rules (C2), a model is inconsistent if it does not have any core instances (S0,J0,J1). The goal (G1) claims that the model has an instance that resolves constraints formed by the conjunction of the facts (C3) and the declarations (C4). The analysis results (Sn0) form the evidence to support the claim. On one side, the set of declarations regroups all implicit constraints in

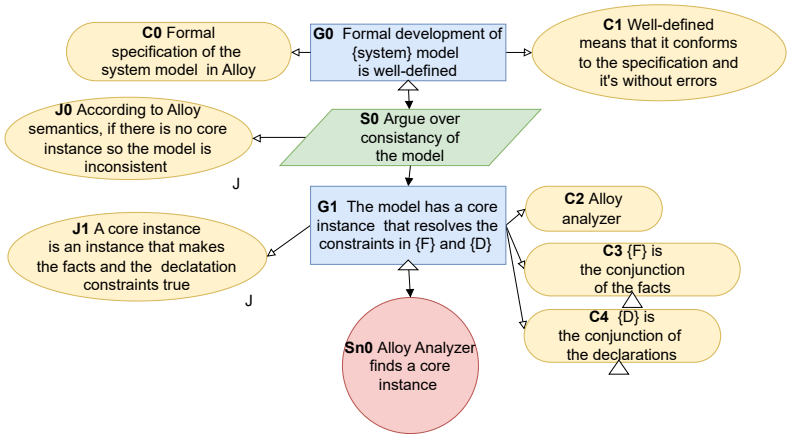

Fig. 4: Pattern for the well-definedness of the system model (P1)

the signatures, mainly type and multiplicity ones. Listing 1.1 depicts some examples of declarations constraints used for the specification of the CBA metamodel from [11]. On the other side, a subset of the conjunction of the facts used to build the CBA metamodel from [11] is presented in Listing 1.4.

```
fact ComponentsConnectorFacts {
  no disj m,m':Msg | some m.payload and m'.payload
  all m: Msg | origin_sender != receiver  ... }
```

Listing 1.4: Examples of facts

We illustrate the application of this pattern (see Fig. 5). First, we define the components and the connectors connecting them based on the metamodel shown in Listing 1.1. Then, we have to formally show that the ACAS Xu model is consistent. We use particularly the facts from 1.4 as the context in the node (C3). The metamodel is built with all the constraints, we will not add new facts to describe the ACAS Xu architecture. However, we add new declarations as shown in Listing 1.5 to instantiate the context node (C4).

```
sig Processor extends Component {} {uses = PortPlanOut + PortSenIn}
sig Sensors extends Component {} { uses = PortSenOut } ....
```

Listing 1.5: Some declarations used to describe ACAS Xu

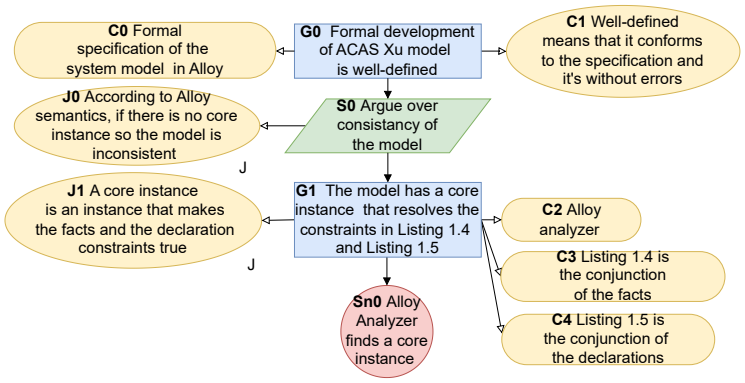

Fig. 5: P1 instantiation example

4.2 Argument pattern for the satisfaction of SRs (P2)

The goal of (P2) is to provide a convincing argument about using Alloy as a formalization language to formalize and verify the SRs. The structure of (P2) is shown in Fig. 6. The root claim (G2) of the pattern is about the satisfaction of the requirement {security requirement}. We refer to the requirement in the node (C5). Context node (C6) provides extra details about the system model and describes the operating environment in which the system operates securely. Since the pattern claims that the system model realizes the SR, it is wise to assume that the system model is well-defined in Alloy (A0) and reflects the true system (A1). The pattern (P1) represents an argument to develop this assumption (A0). The proposed strategy (S1) involves arguing through formal specification and verification of the security requirement presented as model property. The

argument implies that if the sub-claims (G3, G4) are satisfied, then the system satisfies the requirement. Sub-claim (G3) claims that the proposed formalization *property* (C7) is the proper formalization of the requirement. We rely on domain and formalization experts' inspection (Sn1) as evidence that the formulated expression is the proper specification of the requirement and is well-defined according to restrictions imposed in Alloy. According to claim (G4), the model satisfies the property {property}. This claim is supported by the results of model checking on *property*: Alloy Analyzer (C8) doesn't find a counterexample. We assume that the tool support (Alloy Analyzer) is correct (A2) (i.e., analysis parameters (e.g., scope analysis) are well defined).

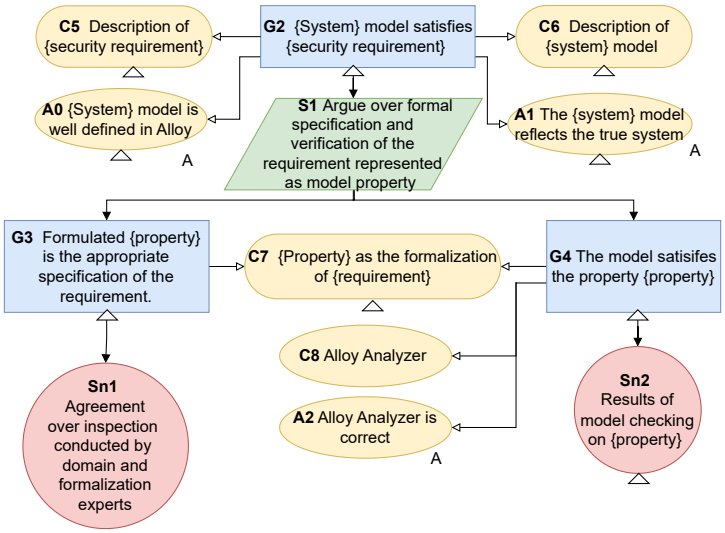

Fig. 6: Argument pattern for the satisfaction of security requirement (P2)

We illustrate the application of this pattern (Fig. 7). We refer to the system in nodes (G2, C5, A0, A1). For the sake of simplicity, we only showcase one instance of (P1) for the satisfaction of the security requirement (SR1). First, we refer to the requirement (SR1) in the context node (C5). SR1 is prescribing the need for a mechanism to realize the message integrity objective. Recall that Listing 1.2 shows the formalization of the integrity objective of any message's payload exchanged between all the components. Particularly, for messages exchanged between GPS sensors and the processor. Consequently, the property *GPSMsgIntegrity* shown in Listing 1.6 is the formal specification of the requirement and is referred to in nodes (C5, G3, G4, Sn2). We follow the same steps to instantiate (P1) for the realization of SR2 and SR3.

```
1 assert GPSMsgIntegrity{ payloadIntegrity [Sensors,Processor]}
2 check GPSMsgIntegrity for 3
```

Listing 1.6: Formal specification of SR1

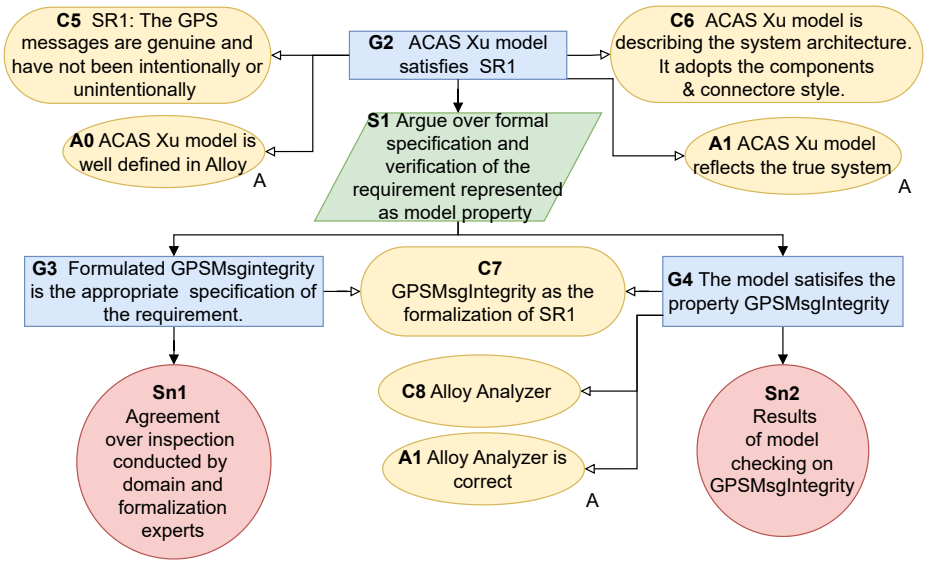

Fig. 7: P2 instantiation example

5 Related Works

Several approaches suggest mapping the activities from the security cases construction process to those of the secure system development process. Activities from different stages of the secure system development process help create either of the two parts (viz., argumentation strategies and evidence) of security cases or both. The work in [5] proposes to use Architecture Analysis and Design Language (AADL) and annex it with Resolute language to specify the system architecture, safety rules, and security claims. The analysis leads to the generation of fragments of assurance cases. This approach, like ours, avoids inconsistencies between a design and its assurance cases, thanks to automated model transformation. However, it needs to show how to assemble these fragments since each fragment is separately arguing about one component from the AADL metamodel. In [9], the authors introduce security assurance cases and present an expansion to the agile development process that entails the construction of these documents. They extract from the security procedures execution the evidence and argumentation needed to support the assurance case. The approach only applies to modular software, where security claims are associated with specific components. Unlike this, we formalized security requirements involving more than one component.Another work in [4] proposes embedding the Isabelle proof assistant and SACM (Structured Assurance Case Model) to generate safety cases. The requirements are expressed using High Order Logic, which allows for specifying more advanced requirements. However, the paper needs more argumentation about model correctness and well-definedness, which is mandatory for model-based development in general, and development using formal methods in particular.

6 Conclusion

In this paper, we propounded a formal model-based approach for the rigorous generation of a security case for critical systems. The approach takes a set of complete and consistent SRs as input and produces security cases for realizing these requirements, along with a well-defined system model. Our work in this paper is security-oriented and Alloy-aware, however, the approach remains valid for other properties (safety, reliability, etc.) and uses other formal methods. However, using the approach in different domains requires basic knowledge of formal modeling, requirements engineering, and assurance cases construction. The argument patterns generated by the approach can be expressed in various notations and have the potential for re-usability. In future work, we aim to develop a tool to facilitate the use of these patterns and instantiate them for constructing error-free security assurance cases. Furthermore, future work includes developing additional argument patterns to support consistency and completeness assessments of identified security requirements, aiming to enhance the confidence in the security case.

References

1. Altawy, R., Youssef, A.M.: Security, privacy, and safety aspects of civilian drones: A survey **1**(2) (2016), https://doi.org/10.1145/3001836
2. Batra, M.: Formal methods: Benefits, challenges and future direction. Journal of Global Research in Computer Science **4**(5), 21–25 (2013)
3. ter Beek, M.H., Larsen, K.G., Ničković, D., Willemse, T.A.: Formal methods and tools for industrial critical systems. International Journal on Software Tools for Technology Transfer **24**(3), 325–330 (2022)
4. Foster, S., Nemouchi, Y., Gleirscher, M., Wei, R., Kelly, T.: Integration of formal proof into unified assurance cases with isabelle/sacm. Formal Aspects of Computing **33**(6), 855–884 (2021)
5. Gacek, A., Backes, J., Cofer, D., Slind, K., Whalen, M.: Resolute: an assurance case language for architecture models. ACM SIGAda Ada Letters **34**(3), 19–28 (2014)
6. Habli, I., Kelly, T.: A safety case approach to assuring configurable architectures of safety-critical product lines. In: International Symposium on Architecting Critical Systems. pp. 142–160. Springer (2010)
7. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT press (2012)
8. Manfredi, G., Jestin, Y.: An introduction to acas xu and the challenges ahead. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). pp. 1–9. IEEE (2016)
9. ben Othmane, L., Angin, P., Weffers, H., Bhargava, B.: Extending the agile development process to develop acceptably secure software. IEEE Transactions on dependable and secure computing **11**(6), 497–509 (2014)
10. Rouland, Q., Hamid, B., Bodeveix, J.P., Filali, M.: A formal methods approach to security requirements specification and verification. In: 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 236–241. IEEE (2019)
11. Rouland, Q., Hamid, B., Jaskolka, J.: Specification, detection, and treatment of stride threats for software components: Modeling, formal methods, and tool support. Journal of Systems Architecture **117**, 102073 (2021)
12. Weinstock, C.B., Lipson, H.F., Goodenough, J.: Arguing security-creating security assurance cases. Tech. rep., Carnegie Mellon University (2007)