

Towards Model Driven Safety and Security by Design

Miguel Campusano, Simon Hacks and Eun-Young Kang

SDU Software Engineering, Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense

Abstract

Software is getting more and more complex, while it gets more and more important to make it safe and secure. At the same time, the expectations towards the software developers increase and it is unrealistic that they are able to cope properly with all safety and security requirements. To enable developers to focus on the important parts of the system, model driven software development got widely adopted. Within this work, we extend this approach by proposing an architecture, which allows to automatize the analysis of the safety and security properties of the system under design. After the analysis of the system, feedback will be provided to the developers so that they are able to reason about the design decisions that they recently made. To discuss our approach, we rely on a model driven approach for drone mission planning and envision how the different components of the architecture would need to interact.

Keywords

Model Driven Software Engineering, Automatized Analysis, Safety, Security, Model Checking

1. Introduction

The complexity of software is increasing as the problems that software solves are getting more and more difficult [1]. This is particularly true when software is used to interact with the real world, by physical interaction using cyber-physical systems, or by providing remote interfaces to enable interaction from all over the world [2]. Due to this interaction, there are new demands toward the safety and security of such safety-critical systems: systems should not harm the people using them nor neglect access to the system from an unauthorized source. However, because of the complexity of these systems, it is unreasonable to expect developers to produce bug-free, safe and secure code. For this, models play a central role, as high-level abstraction allows developers to focus on the fundamental complexity of the systems (i.e., what the system is supposed to do) instead of their incidental complexity (e.g., safety and security) [3]. This high-level abstraction allows developers to build safe and secure code by design by taking care of the incidental complexity. As it is challenging to include proper safety and security measures into a system subsequently, it is preferable to follow a safety/security by design approach while developing a software system [4]. In each development phase, the respective measures can be included instead of cumbersome included at the end. This increases the safety and security of the system, while reducing the needed effort to introduce the needed measures.

There is a pressing need for methods and tools to verify and validate reliability of software systems, i.e., all software we build should be correct, robust, safe, and secure under certain circumstances. To prove safety and achieve error-free software systems, formal reasoning and methods are used by detecting when the system transitions into an unsafe state (i.e., one where it violates a critical safety requirement) [5, 6]. While testing can provide some reassurance that the systems being developed are bug-free, it is limited by the skills and expertise of the tester. It is not guaranteed that testing can find all errors or show their absence whereas formal verification can be employed exhaustive analysis [7]. Thus, the use of a combination of both approaches and software engineering techniques ensures potential errors are captured as early as possible. Our focus is on the use of formal methods alongside testing approaches, formal verification can be applied to establish functional correctness and can be combined with model-driven testing. This approach is integrated into a development workflow and provides correct configurations and practical considerations of design from an industrial perspective.

There are different approaches to achieve security by design for software systems [4]. One approach is to continuously perform penetration tests of the system under development [8]. However, this requires a large portion of resources to be permanently executed. Moreover, this requires already a system that can be tested. Another option is to perform attack simulations on a threat model that represents the system based on known vulnerabilities. This allows not only an easy security assessment of the actual system under development, but also it is possible to compare the security properties of different possible systems without having them already developed [9].

To enable software developers to assess the safety and security of their systems in almost real-time, it is nat-


QuASoQ 2022: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual

✉ mica@mmmi.sdu.dk (M. Campusano); shacks@mmmi.sdu.dk (S. Hacks); eyk@mmmi.sdu.dk (E. Kang)

🆔 0000-0002-7894-6635 (M. Campusano); 0000-0003-0478-9347

(S. Hacks); 0000-0002-4589-2378 (E. Kang)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

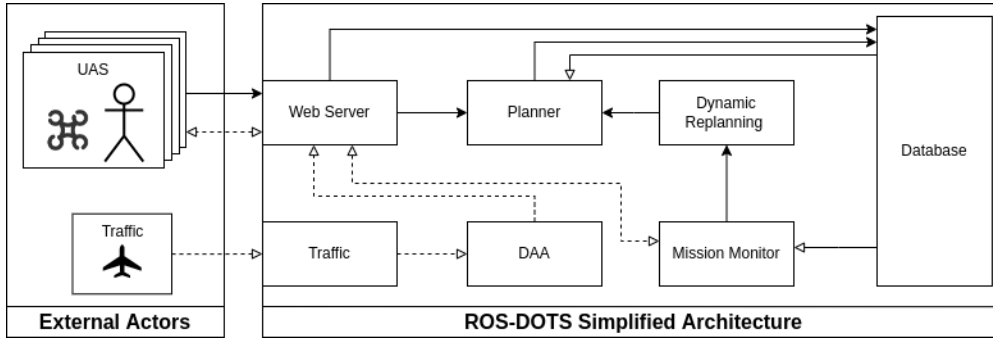


Figure 1: Simplified ROS-DOTS architecture. The different lines represent different types of interaction: black arrows are request/response calls, white arrows are callbacks, and dotted lines are constant data flow by publish/subscribe pattern.

ural to combine the beforehand presented approaches. Therefore, the models and design created by developers are not solely used for implementing and generating the system, but they are also transformed automatically into the respective representations that are needed for the assessment of the safety and security. In the background, the assessment will be executed and as soon as the results are available, the developer gets informed about potential safety or security issues in their actual software architecture thought these same models. In a second step, the developer also gets suggestions how to improve the safety and security based on the comparison of different possible evolutions of the actual architecture.

Within this work, we present the first step towards an automatized model driven safety and security assessment of software systems. To achieve this, we discuss first the background of Model Driven Software Development, Safety Assessment, and Security Assessment. Afterwards, we present our vision of achieving a model driven safety and security assessment before we discuss first insights and how we plan to continue our work.

2. Background and Related Work

2.1. Model Driven Software Development

Model Driven Software Development (MDS) refers to using software abstractions to separate fundamental and incidental complexity of systems. These abstractions are done by models, which are the representation of the essential aspects of the system. By using them, developers can define structures and behaviors on these systems efficiently, considering the domain-specific aspects of the systems. Then, developers can use this high-level abstraction of the designed system to generate executable source code by a sequence of model transformations [10, 11].

Any modeling approach is described using metamodels

[11]. Models are abstractions over similar programs while metamodels are abstractions over similar models from a particular domain. One specific way of designing systems using models is first designing a metamodel in the form of a Domain Specific Language (DSL). A DSL is a concise language that describes a solution of a particular domain. Developers can use the DSL to define programs that specify the behaviors of the different models of the system. Moreover, the domain abstractions allow the DSL to be used by developers and domain experts [12].

We have successfully used MDS to develop Unmanned Aerial Systems (UAS), commonly refers as drones. We built a DSL for the specification of multi-UAS missions, called Drone Operation Template Specification (DOTS) [13], which uses specific languages constructions to coordinate the use of several UAS in the airspace. The DOTS programs are then loaded to a service-oriented architecture called ROS-DOTS [14] (Robot Operation System (ROS)). This architecture provides several services for UAS, in particular multi-UAS mission planning (i.e., specification of UAS flight paths to fulfill a goal), dynamic replanning of missions (i.e., flight paths modification of ongoing UAS missions), and a detect-and-avoid system (i.e., local collision alerting of UAS). Figure 1 shows a simplified version of ROS-DOTS.

2.2. Safety Assessment

Safety assessment methods include preliminary and system hazard and risk analysis, fault tree generation and analysis, failure mode and effects analysis. Despite such well-established methods provide an efficient support for safety engineers, the methods could benefit from an integration with system modeling, verification, and validation (V&V) environments. Efforts have been put into investigation of safety assessment through the MDSE based on general purpose System Modeling Language (SysML) [15], Similar studies are also conducted in other

modeling language such as EAST-ADL [16, 17, 18, 19] or AADL [20] that support writing transformation rules towards formal languages to permit their analysis by formal tools. However, these languages are limited for robot design compared to our domain specific UAS and meta-attack languages. Systems theory process analysis (STPA) [21, 22, 23] has been studied in the context of unifying both safety and security assessment. However, their approaches lack formalism that limits formal V&V. As far as we know, our approach is the first to combine both safety and security assessment based on MDSE, STPA, attack simulations, and formal V&V to guarantee trustworthiness in cyber-physical systems, e.g., robot, UAS, manufacturing, and IoT systems, etc.

To reasoning about and analysis of a design model it is essential that the modeling language has a well-defined (informal or formal) semantics. For cyber-physical systems (which heavily rely on the real-time aspect) a promising approach to provide analysis of models is to formally specify systems in a modeling language such as Timed Automata (TA) [24]. A TA is a finite state machine extended with clocks, where a *clock* is a variable over the positive real numbers. All clocks in a TA start at zero, grow continuously at the same rate, and can be tested and reset to zero. Clocks are tested using constraints on clocks, called *guards*. A TA over actions A is defined as a tuple $\langle N, l_0, E, V_C, I \rangle$, where N and E are the locations and edges, $l_0 \in N$ is the initial location, V_C is the set of clock variables, and $I : N \mapsto G$ with guards $g \in G$, actions $a \in A$, and a set of clocks $r \subseteq V_C$ to be reset (an alternative notation is $x := 0$ for the rest of a clock x). Figure 3 shows an example of a TA (model in right side), which is a formal representation of the *RiskStatus* state diagram (in left side). The actions used in the TA are *risky* and *no_risky*. The location *No Alert* is marked initial, as indicated by an extra circle inside it. The edge from *No Alert* to *Alert* is labelled with the action *risky* with a guard status. The mode has a clock *Time*, which is used to measure the time elapsed since the action *no_risk*. The location *After Alert* is labelled with a clock invariant to ensure that the delay is less than five time units between the actions *no_risky* and *risky/reset* time. Edges are labelled with guards to ensure that the delay is more than five time units and the current status is not alert, i.e., $\text{status} \neq \text{alert}$.

2.3. Security Assessment

Designing secure and reliable systems is challenging and attackers constantly find opportunities to compromise systems. There are different countermeasures at the disposal of organizations to cope with this challenge, such as applying best practices (e.g., OWASP [25]), penetration testing [26], established frameworks (e.g., Process for Attack Simulation and Threat Analysis (PASTA) [27]),

or threat modeling [9].

Here, we facilitate threat modeling to analyze the security properties of the system under design. Via threat modeling one wants to reason about the complexity of a system, as well as identifying potential threats [28]. Usually, this is done by graphs, where each of the threats is modeled as a node and they are connected by edges [29]. Given a threat model, attack simulations allow to analyze attack scenarios on the described infrastructure [30, 31].

More concretely, we rely on the Meta Attack Language (MAL) as tool to perform our attack simulations. For a detailed overview of the MAL, we refer readers to the original paper [32]. First, a MAL DSL contains the main concepts of a domain under study, so called assets. An asset contains attack steps, which represent the actual attacks/threats that can be executed.

An attack step can be connected with n following steps creating an attack path, which is used for the attack simulation. Attack steps can be either OR or AND. Additionally, each attack step can be related with specific types of risks (i.e., confidentiality (C), integrity (I), and availability (A)). Furthermore, we have defenses at our disposal that do not allow connected attack steps to be performed. Finally, we can assign probability distributions to represent the effort to complete the related attack step. Assets have relations between them, so called associations. Moreover, we have inheritance between assets and each child asset inherits all the attack steps of the parent asset. Additionally, the assets can be organized into categories.

In List. 1, a short example of a MAL DSL is presented. In this example, we have four assets and their connections of attack steps from one asset to another. In the *Host* asset, the *connect* attack step is an OR attack step, while *access* is an AND attack step. The \rightarrow symbol denotes the connected next attack step. For example, if an attacker performs *phish* on the *User*, it is possible to reach *obtain* on the associated *Password* and as a result finally perform *authenticate* on the associated *Host*. In the last lines of the example the associations between the assets are defined.

3. Automated Safety and Security Assessment

3.1. Architecture Framework

The MDSD approach allows developers to design systems using high-level abstractions (i.e., models). Then, these abstractions generate executable source code corresponding to the system's behavior. Our objective is to use an MDSD approach to model and build systems considering safety and security in their design. To do this, we plan to reuse the same abstractions that define programs to generate safety and security assessment artifacts. This conceptual model and the interaction between the sys-

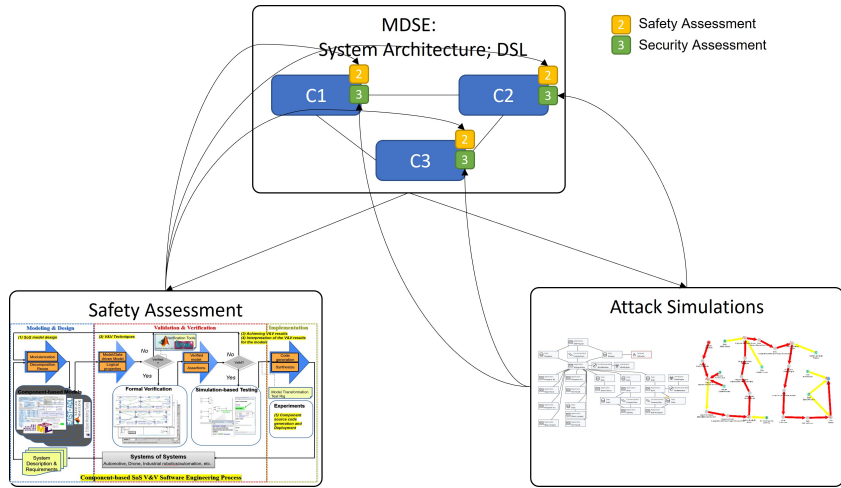


Figure 2: Methodology Roadmap

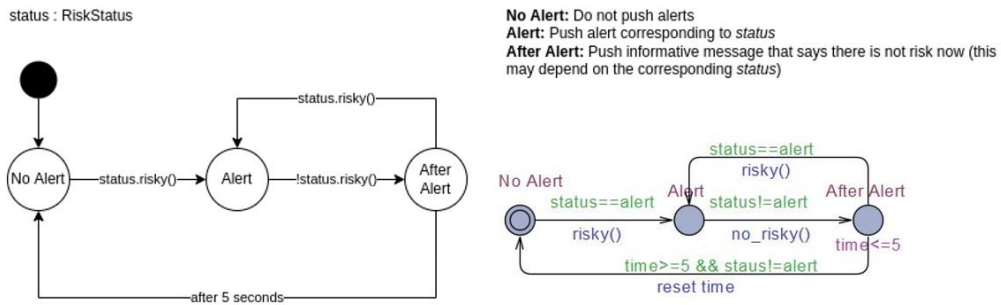


Figure 3: ros-state diagram for the risk status behavior of Detect and Avoid (DAA) service (left side) and its corresponding TA (right side)

```

operation carry_medicine:
import action Move<navigation_point>
import property Transport
import property Camera
nursing_home: navigation_point
camera_uas: drone<Transport, Camera>
implementation:
camera_uas += Move(nursing_home)
camera_uas should not be able to carry a camera when carrying payload to nursing_home

```

Figure 4: Conceptual feedback of a privacy issue of a UAS moving to a location given directly in the DOTS DSL.

models derived over the services and their interactions in the ROS-DOTS architecture.

First, the assessment of the safety and security properties of the models generated by the DSL can be explicitly shown to developers. To allow this level of feedback, a bi-directional connection between the models and the generated artifacts that validate security and safety properties should be available. In other words, models should generate artifacts in a way that the artifacts can relate to the models that generated them. Then, like in every modern Integrated Development Environment (IDE), the system can mark the problematic lines of code in the DSL program with a meaningful message for developers to fix the problems. For example, we can consider the case of a UAS transporting a package from point A to B. An operator can use any UAS capable of carrying a payload for this action. However, the system may restrict the use

tem architecture, safety assessment, and security attack simulations can be seen in Figure 2. In the context of this work, we use the example of defining multi-UAS missions. In this system, we have two different high-level definitions: models derived over the DOTS DSL and

```

1  category System {
2    asset Network {
3      | access
4      -> hosts.connect
5    }
6
7    asset Host {
8      | connect
9      -> access
10     | authenticate
11     -> access
12     | guessPwd
13     -> guessedPwd
14     | guessedPwd [Exp(0.02)]
15     -> authenticate
16     & access {C,I,A}
17   }
18
19   asset User {
20     | attemptPhishing
21     -> phish
22     | phish [Exp(0.1)]
23     -> passwords.obtain
24   }
25
26   asset Password {
27     | obtain {C}
28     -> host.authenticate
29   }
30 }
31
32 associations {
33   Network [networks] *
34   <-- NetworkAccess --> * [hosts] Host
35   Host [host] 1
36   <-- Credentials --> * [passwords] Password
37   User [user] 1
38   <-- Credentials --> * [passwords] Password
39 }

```

Listing 1: Exemplary MAL Code

of a UAS with extra properties, which can entail security issues, for example, a UAS with a camera attached. An attacker can intercept the link between the UAS and the operator, accessing the camera images, which can bring privacy issues for the people living around the path of the UAS. While a UAS with a camera is essential for other use cases (e.g., monitoring a geographical area), we want to limit the use of the right UAS for the right job, to reduce security and safety issues. Figure 4 shows a concept of how this feedback can be displayed in DOTS.

Second, the safety and security properties of the architecture itself should also be checked and informed to developers. To do this, we can use the same idea of a bi-directional connection between the executable architecture and the generated artifacts that check security and safety properties. The architecture can generate artifacts to test properties over single services and the communication between multiple services. One example of a single service test is to check how the detect-and-avoid service works (DAA in Figure 1). This service alerts operators when a UAS is in a direct collision path with an external agent in the airspace. It is vital to check the safety properties of this service to ensure a safe interaction of agents in the airspace.

As a key example of a multi-service architecture tester, we present the dynamic replanning feature. This feature replans the path of every UAS involved in a mission when new constraints that affect the original plan are added into the airspace. This feature uses three services: Planner, Mission Monitor, and Dynamic Replanning. Suppose the dynamic replan service returns a plan with safety problems, such as 2 UAS in a direct path against each other. In that case, our safety checker can inform this issue directly to developers.

3.2. Formal framework for Safety and Security Assessment

To be trustworthy, systems need to remain safe and secure while being resilient to unpredictable changes, functional/operational failures and cyber-security threats. Rigorous V&V is essential to ensure trustworthiness of systems and clear definition of requirements is an important prerequisite for V&V.

Most engineering practice highly relies on V&V test-and-fix of system nature, which is time-consuming, expensive, and limiting the possibilities for exploration of alternatives in system design. Thus, we provide a correct-by-construction approach based on a combination of analysis techniques such as Systems Theoretic Process Analysis (STPA) [21] and formal verification such as model checking to generate critical requirements, remove ambiguities in the requirements, and specify formal safety and security properties that should be satisfied by the system. We also suggest a modularized/compositional approach for formal modeling to enhance re-usability and to reduce the complexity of formal modeling.

To facilitate the formal verification, the system architecture (illustrated in Figure 2) consisting of a set of function/service blocks and its operational behaviors are translated into a formal modeling description, UPPAAL¹ TA. To explicitly annotate and reason about the functional or operational behavior of each block (e.g., DAA in Figure 1) at the architecture level, we first adopt a state diagram (*RiskStatus* ros-state diagram in Figure 3) and extend it with a UML profile which integrates relevant concepts from our ROS-DOTS. Such a profiled model is then translated into a formal modeling description, UPPAAL TA. The translation process is supported in fully automatic by using our DSL. The communications between different blocks are also transformed into synchronization channels among other TAs in UPPAAL. Indeed, each asset and its behavioral logic in Listing 1 are visualized in a TA and the associations are represented in synchronization channels among different TAs.

¹<https://uppaal.org>. An integrated tool environment for formal modeling, validation and verification of real-time systems modeled as networks of TA

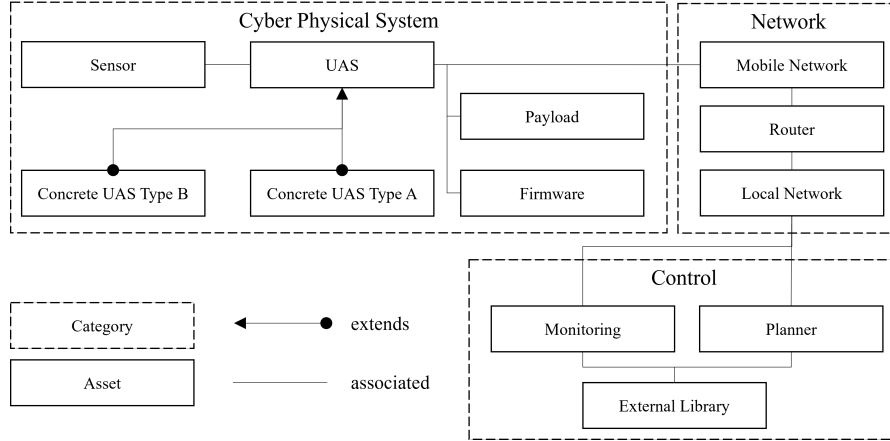


Figure 5: Sketch of a possible uasLang for security analysis

Similarly, a path-planning TA can be generated based on the Planner block and its *ros-state* diagram.

The auto-generated TA model is then amenable to formal verification using a UPPAAL model checker against safety and security requirements. Furthermore, additional safety and security constraints are identified based on STPA and feedback from model checking and attack simulation (which will be further explained in the following section). The system design can be refined by adding the new constraints which inhibit any hazards. Finally, we prove correctness and consistency of the safety and security constraints through formal verification and improve the system’s design accordingly. In addition to formal verification, testing can be performed to validate the implementation (code as written/automatically generated by our DSL) actually runs correctly w.r.t. to the safe and secure design/specification.

The security of a system is affected by a broad variety of aspects. Formal verification of a system is an important step to assure to its security. However, to get a complete verification of the system, every detail about the system must be known. Moreover, depending on the system’s size, a complete verification is time intensive and, thus, usually only conducted for extremely sensitive system like rockets transporting humans to space. Alternatively, a verification on an abstracted view of the system can be conducted, which is less time intensive, but accompanied by the cost of a not complete verification.

To address this shortcoming, we foresee in our architecture to not solely rely on verification, but also on threat modeling and attack simulations as they are also able to cope with incomplete information (i.e., “known unknowns” [33]). We facilitate MAL [32] as vehicle to perform out attack simulations. More concretely, we imagine a uasLang (cf. Figure 5) that might reuse certain parts

of existing MAL languages [34], such as coreLang [35] for the fundamental IT parts or icsLang [36] for the operation technology (OT) similar parts (e.g., sensors or actuators).

In the following, we will shortly elaborate on the different assets that we envision in a possible uasLang. Firstly, we have the assets often referred to as cyber physical system. There is the *UAS* itself, which is the hardware platform carrying the *Payload*. Moreover, a *UAS* has *Sensors*, which help it to orientate itself in the real-world. The entire platform is controlled by the *Firmware*, which processes the incoming data from the *Sensors* and communicates with the controlling assets *Planner* and *Monitoring*.

The *Planner* is the central unit coordinating the different *UAS* and the routes they are taking to reach to their waypoints. The *Monitoring* receives the actual state of the *UAS* and provides an interface to external systems that might further process this data. Further, both assets can incorporate *External Libraries*, e.g., to perform better routing or providing additional reporting capabilities.

The communication between the *UAS* and the controlling units takes place in classical *Local Network*, like Ethernet, in which the IT parts are hosted and a *Mobile Network*, like 5G, that covers the operation area of the *UAS*. These two networks are usually separated by some kind of *Router* restricting the network access.

Given uasLang and the models created during the model driven software development, we are now able to create a threat model that represents the actual infrastructure and perform attack simulations in securiCAD [30]. The simulations will tell us potential threats in the architecture and in which time an attacker might be able to exploit them. This information is then played back to the developer, so that they are able to determine

possible countermeasures in their system's architecture to improve the security.

4. Challenges & Future Directions

The distinctive features of cyber-physical systems with regard to model checking are their complexity, modularity and the need to comply with safety and security requirements, which means that every failure result should be thoroughly analyzed and fixed. Despite being one of the most reliable approaches for ensuring system correctness, model checking requires additional knowledge about a system as a whole and efforts aimed to localize an error in the model of the system. A tool or framework that supports user-friendly model checking which focuses on explanation of negative verification results and performs an analysis that the refined system design is still consistent would be highly beneficial. For example, in the system verification process, once a violation of a safety or security constraint/requirement is detected, a counterexample (failure trace) generated by a UPPAAL model checker that can be visualized in the architecture model in order to pinpoint on which time step/block has caused the property failure. Seemingly including such aspects into our framework and tool is crucial for making the formal verification techniques more approachable to engineers.

For the security assessment, we recognize that not all parts of the architecture (cf. Figure 1) are reflected one-to-one in uasLang. However, this is no issue due to two reasons. Firstly, the security assessment takes the point of view of an attacker. Thus, we are not solely interested in the system's architecture, but to all parts that are exposed to the environment. Consequently, uasLang contains further information (e.g., on concrete UAS deployed), that might be provided from outside of our presented solution. Other parts of the architecture might not be of greater relevance for the security assessment (e.g., the web server), as they do not change in our setting (and even do not have a representation in the used model driven software development approach) and thus do not have any influence on the outcome of the attack simulation results.

Secondly, it is recommended to base a newly developed MAL DSL on another existing MAL DSL [34, 37]. Consequently, we would base uasLang at least on coreLang [35]. In other words, we would have all assets available that are available in the base language (i.e., coreLang) and, thus, would be able to model all IT related assets presented in Figure 1.

Moreover, we have more consideration on the modeling of the architecture, when we relate this modeling to users and developers. For example, a DSL is built to be concise, to have the necessary features to describe the so-

lution using models, and no more. However, developers may need to define extra properties for assessing security and safety properties, that are not needed to solve the problem itself. In other words, they may need to define aspects of the incidental complexity of the problem they are solving. While this may be an issue, we envision an architecture where developers should declare as few as possible incidental properties. In addition, even when they need to declare these properties, the system may allow them to define them in other parts of the system, such as configuration files, environment variables, directly into the architecture, etc.

Finally, we are aware that checking safety and security properties may take some time, making it inappropriate to give feedback to developers when they are writing their programs. Even when certain properties can be checked fast enough to give them while developers write their programs, we need to be aware of the properties that can take more time. For the properties that take a considerable amount of time, we envision a system that gives feedback to developers when the program is running (similar to debugging), or when the program finishes its execution (similar to the case of automatic testing or continuous integration/delivery systems).

References

- [1] T. Mens, On the complexity of software systems, *Computer* 45 (2012) 79–81.
- [2] K. Zhang, D. Han, H. Feng, Research on the complexity in internet of things, in: 2010 International Conference on Advanced Intelligence and Awareness Internet (AIAI 2010), IET, 2010, pp. 395–398.
- [3] O. Pastor, S. España, J. I. Panach, N. Aquino, Model-driven development, *Informatik-Spektrum* 31 (2008) 394–407.
- [4] M. Waidner, M. Backes, J. Müller-Quade, Development of secure software with security by design, *Fraunhofer-Verlag*, 2014.
- [5] E. M. Clarke, J. M. Wing, Formal methods: State of the art and future directions, *ACM Comput. Surv.* 28 (1996) 626–643.
- [6] F. Benaben, M. Larnac, J. Pignon, J. Magnier, A process for improving multi-technology system high level design: Modeling, verification and validation of complex optronic systems, in: 2000 International Conference On Systems, Man & Cybernetics, volume 1–5, IEEE, 2000, pp. 1036–1040.
- [7] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, Cambridge, 2007.
- [8] B. Arkin, S. Stender, G. McGraw, Software penetration testing, *IEEE Security & Privacy* 3 (2005) 84–87.

- [9] W. Xiong, R. Lagerström, Threat modeling—a systematic literature review, *Computers & security* 84 (2019) 53–69.
- [10] S. Beydeda, M. Book, V. Gruhn, et al., *Model-driven software development*, volume 15, Springer, 2005.
- [11] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, *Model-driven software development: technology, engineering, management*, John Wiley & Sons, 2013.
- [12] M. Fowler, R. Parsons, *Domain-specific languages*, Addison-Wesley Professional, 2010.
- [13] M. Campusano, N. Heltner, N. Mølby, K. Jensen, U. P. Schultz, Towards declarative specification of multi-drone bvlos missions for utm, in: *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, IEEE, 2020, pp. 430–431.
- [14] M. Campusano, K. Jensen, U. P. Schultz, Towards a service-oriented u-space architecture for autonomous drone operations, in: *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, IEEE, 2021, pp. 63–66.
- [15] P. David, V. Idasiak, F. Kratz, Reliability study of complex physical systems using sysml, *Reliability Engineering & System Safety* 95 (2010) 431–450.
- [16] EAST-ADL, last accessed on 20.10.2022. <http://maenad.eu/index.htm>.
- [17] E. Kang, P. Schobbens, Schedulability analysis support for automotive systems: from requirement to implementation, in: *Symposium on Applied Computing*, ACM, 2014, pp. 1080–1085.
- [18] L. Huang, T. Liang, E. Kang, Tool-supported analysis of dynamic and stochastic behaviors in cyber-physical systems, in: *19th International Conference on Software Quality, Reliability and Security*, IEEE, 2019, pp. 228–239.
- [19] E. Kang, G. Perrouin, P. Schobbens, Model-based verification of energy-aware real-time automotive systems, in: *18th International Conference on Engineering of Complex Computer Systems*, IEEE, 2013, pp. 135–144.
- [20] P. Feiler, D. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, Addison-Wesley Professional, 2012.
- [21] N. Leveson, J. Thomas, *STPA Handbook*, Cambridge, 2018.
- [22] I. Friedberg, K. McLaughlin, P. Smith, D. Laverty, S. Sezer, STPA-SafeSec: Safety and security analysis for cyber-physical systems, *Journal of Information Security and Applications* 34 (2017) 183–196.
- [23] W. Young, N. G. Leveson, An integrated approach to safety and security based on systems theory, *Commun. ACM* 57 (2014) 31–35. doi:10.1145/2556938.
- [24] R. Alur, D. L. Dill, A theory of timed automata, *Theoretical Computer Science* 126 (1994) 183–235.
- [25] M. Bach-Nutman, Understanding the top 10 owasp vulnerabilities, *arXiv preprint arXiv:2012.09960* (2020).
- [26] M. Bishop, About penetration testing, *IEEE Security & Privacy* 5 (2007) 84–87.
- [27] M. M. Morana, T. Uceda Vélez, *Risk centric threat modeling: Process for attack simulation and threat analysis*, John Wiley & Sons, Hoboken, New Jersey, 2015.
- [28] A. Shostack, *Threat modeling: Designing for security*, Wiley, Indianapolis, IN, USA, 2014.
- [29] S. Myagmar, A. J. Lee, W. Yurcik, Threat modeling as a basis for security requirements, in: *SREIS*, volume 2005, Citeseer, 2005, pp. 1–8.
- [30] M. Ekstedt, P. Johnson, R. Lagerström, D. Gorton, J. Nydrén, K. Shahzad, securiCAD by foreseeti: A CAD tool for enterprise cyber security management, in: *19th International EDOC Workshop*, IEEE, 2015, pp. 152–155.
- [31] H. Holm, K. Shahzad, M. Buschle, M. Ekstedt, P²Cy-SeMoL: Predictive, probabilistic cyber security modeling language, *IEEE Trans Dependable Secure Comput* 12 (2015) 626–639.
- [32] P. Johnson, R. Lagerström, M. Ekstedt, A meta language for threat modeling and attack simulations, in: *13th ARES Conference*, 2018, pp. 1–8.
- [33] S. Hacks, M. Kaczmarek-Heß, S. de Kinderen, D. Töpel, A multi-level cyber-security reference model in support of vulnerability analysis, in: *International Conference on Enterprise Design, Operations, and Computing*, Springer, 2022, pp. 19–35.
- [34] S. Hacks, S. Katsikeas, Towards an ecosystem of domain specific languages for threat modeling, in: *International Conference on Advanced Information Systems Engineering*, Springer, 2021, pp. 3–18.
- [35] S. Katsikeas, S. Hacks, P. Johnson, M. Ekstedt, R. Lagerström, J. Jacobsson, M. Wällstedt, P. Eliasson, An attack simulation language for the it domain, in: *International Workshop on Graphical Models for Security*, Springer, 2020, pp. 67–86.
- [36] S. Hacks, S. Katsikeas, E. Ling, R. Lagerström, M. Ekstedt, powerlang: a probabilistic attack simulation language for the power domain, *Energy Informatics* 3 (2020) 1–17.
- [37] S. Hacks, S. Katsikeas, E. Rencelj Ling, W. Xiong, J. Pfeiffer, A. Wortmann, Towards a systematic method for developing meta attack language instances, in: *International Conference on Business Process Modeling, Development and Support*, International Conference on Evaluation and Modeling Methods for Systems Analysis and Development, Springer, 2022, pp. 139–154.