Model driven design of secure properties for vision-based applications: A case study

Giuseppe Airò Farulla*, Marco Indaco
Looqui Srl and Politecnico di Torino, Turin, Italy
giuseppe@looquilabs.com, marco@looquilabs.com

Axel Legay
INRIA, Rennes, France
axel.legay@inria.fr

Tiziana Margaria
Univ. Limerick and Lero, Limerick, Ireland
tiziana.margaria@lero.ie

* corresponding author

Abstract—In this paper we discuss an approach to overcome difficulties and gaps which are typically encountered when dealing with security-oriented model-driven approaches. In particular, we state that state-of-the-art MDS approaches are not suitable for modern companies and industry in general, and address security only at a late stage of development, often causing big delays and reengineering costs due to extensive reworks. Instead, we propose to adopt in the SEcube platform an OTA-based XMDD approach to integrate security ab-initio. In addition, since our approach is based on a set of reusable SIBs organized within dedicated palettes in DIME, we decouple the issue of guaranteeing that the SIBs are correct and secure from the issue of analyzing the applications, which can be greatly simplified by knowing the characterization of each SIB in advance. We apply our approach to the concrete realm of computer vision steering robotics, present the safety and security properties elicited on the specific case study, and discuss the ways they can be enforced.

I. INTRODUCTION
Continuous Model-Driven Engineering (CMDE) [1] is a software and system development methodology which focuses on creating and exploiting domain-specific models, which are conceptual models of all the artefacts, concepts, actions, and properties related to a specific system under development. Hence it aims at creating abstract and possibly reusable representations of the knowledge and activities that govern a particular application domain, rather than focusing on the computing (i.e., algorithmic) concepts.

The CMDE approach is meant to increase productivity along the entire life-cycle by maximizing compatibility between systems (via reuse of standardized models), simplifying the design process (via models of recurring design patterns in the application domain), and promoting communication between individuals and teams working on the system (via standardization of the terminology and the best practices used in the application domain).

Model-Driven Architecture (MDA) is a model-based software design approach for the development of software systems. Traditional Model Driven Development (MDD) of software, as promoted e.g., by the OMG, shifts the attention and the design activities from the programming level to the modeling level, but still remains in the IT realm. Even at the platform independent (PIM) level, the typical UML-based MDD and MDA-approaches provide varieties of model structures that focus on different technical issues/aspects, that have separated life-cycles. Mastering this wealth is a special art requiring both IT knowledge and a good ability of dealing with abstraction.

The MDD paradigm relies on the use of Domain-Specific Modeling Languages that incorporate as core objects (also called “first-class citizens”) elements of the domain being modeled and their relationships, and model transformations that transform the models into platform-specific artefacts, such as code [2].

In this paper we focus on security aspects in CMDE, concerning modeling of secure properties in the concrete realm of Computer Vision (CV) and vision-based algorithms, used in many fields of modern IT. We also present and discuss a practical case study.

Model-Driven Security (MDS) has for more than a decade proposed methodologies for supporting the development of secure systems, or the development of systems leveraging on secure properties and actions to protect mission-critical tasks. Yet, according to studies and state-of-the-art papers such as [3], there is still a big gap between the current practice and what is need to make MDS more easily applicable and adoptable by companies and industry. Most current MDS approaches do not extensively deal with multiple security concerns but rather focus on a specific one, i.e., authorization or authentication.

Security patterns, based on domain-independent, time-proven security knowledge and expertise, could be considered as reusable security bricks upon which sound and secure systems can be built. They are however not actually applied as much as they could be. This is mainly due to two reasons:

- developers have problems in selecting them and applying the right pattern in the right places, especially at the design phase, because they are not used to this programming modality, and
- although the framework enables powerful and almost limitless security federation in principle, it requires users to have deep knowledge of security, and an understanding of the security infrastructures.

To become truly pervasive, as required in today’s interconnected and des-intermediated world, security should be unified with the software engineering process, and thus security engineering is of great importance [4].
II. Modeling Security with the One Thing Approach

Developing systems with the eXtreme Model-Driven Development (XMDD) paradigm [6], [7] involves the user/application expert continuously throughout the whole systems’ life-cycle, according to a user-in-the-loop and expert-in-the-loop philosophy [8]. It is model-driven because it is based on the One-Thing Approach (OTA) [5], [9], which works by successively enriching and refining single artefact that is a rich multi-aspect and multi-faceted model. We use the DIME [10] tool, a Cinco [11] product that is adequate for modeling the aspects of concern here. As illustrated in [10], it is possible to model an application-level security aspect through Role Based Access Control (RBAC). In addition, as shown in [12] and [13], within DIME it is possible to use Service Independent Building Blocks (SIBs) that provide a service-oriented library of basic security communication for Data at rest and Data in motion, as well as a choice of higher level primitives and protocols that can be embedded in the application under design [14]. The entire SEmcubë™ platform [15] is therefore prepared to support users in adding security aspects to the model, by leveraging predefined abstract security primitives, which they might theoretically not even know nor understand in detail.

In this paradigm, our application models are at the center of the design activity and the first class entities of the global system design process. In this approach:

- domain specific libraries are established at the model level: our building blocks are (elementary) models rather than software components;
- systems are specified by model assembly. Here we use orchestration, hierarchy, and configuration as composition techniques;
- knowledge and requirements are expressed by means of properties, via constraints that are formulated in an automatically verifiable fashion. Actually, some of the constraints happen to be domain-independent, and to be already taken care of at design time of DIME;
- security is layered inside SIBs, at the SLG level, and at the global level including the run-time environment. We will see this briefly described on the case study;
- system changes (e.g., upgrades, customer-specific adaptations, new versions) occur only, or at least primarily, at

the model level, with a subsequent global re-verification, and re-compilation (or re-synthesis, in the future):

- optimizations are kept distinct from design issues, in order to maintain information on the structure and the design decisions independently of the considerations that lead to a particular optimized implementation.

For these reasons, DIME includes by design the support of properties and model manipulations that are foundational for the OTA-based XMDD. DIME focuses on application experts, who are typically non programmers, and its versatility is one of its key characteristics. Although there has been a lot of research on security issues concerning technologies, we want to address business-level security intent at a level that is easy to understand even for business users.

III. Integration of Security Aspects

The current state-of-the-art in developing security-critical software and systems in practice is far from being satisfactory. New security vulnerabilities are discovered on an almost daily basis. Integration of security into the overall development process is problematic and suffers from two gaps [16]. First, it is possible to identify a gap as security models and system design models are typically disjoint and expressed in different ways. Second, although security requirements and threats are often considered during the early development phases (requirements analysis), there is another gap with security mechanisms which are later employed and implemented in the final development phases (system integration and validation). As a result, security is typically integrated into systems in a post-hoc manner, this way degrading the security and maintainability of the resulting systems.

To address this problem, a significant amount of work over the last decades has provided model-based development approaches [16]. State-of-the-art approaches which aim to raise the trustworthiness and intrinsic security of mission-critical systems are typically based on the Unified Modeling Language (UML) [17], consisting of a collection of formalisms intended to collectively provide a standard way to define and visualize the architecture and behavior of a system. However, we believe that UML is neither formal nor intuitive enough to be easily understood by application experts, and advocate the need of a XMDD-based approach for security to gain better formality, consistency, and at the same time full notation-independency.

The aim is to integrate security aspects ab initio, from the high-level definition of processes all the way through the whole development of embedded/cyber-physical systems. This will enable true system-immanent security modeling. By integrating security and XMDD, it is possible to model and generate security aware applications that only present options to the user that are consistent with the formalized security policy. Users can be also provided with short checklists (in reality palettes of constraints) summarizing established rules of prudent secure engineering. Our approach towards model-based security engineering combines the following strategies:
• definition of security primitives;
• automatic analysis of models against security requirements;
• automatic generation of code (or tests) from models.

Before presenting our approach it is interesting to define what could be an ideal primitive, which varies in its traits between different application domains. In our domain, security primitives should be composed of a hierarchy of controllers and a data path. We are interested in implementing through such a primitive security controllers and to enforce integrity and privacy within our resources, intended both as data and as processes. The main idea is to define security modeling languages to form a general-purpose extensible DSL, in the sense that they leave open the nature of the protected resources, that belong to some vertical application domain, and the nature of the rules of the game.

Starting from this consideration, our SIBs and Service Logic Graph (SLGs, graphs representing the control flow of an application) are created by modeling primitives and generation rules for integrating security into the development process, i.e., including the resources the security primitive protects. Graphically, they will be visualized within colored boxes, indicating that they are executed under surveillance of the security controller.

Once designed, the SIBs managing critical accesses or operations will be easily reusable in other applications of the same, or similar, domain, to create new secure applications. Secure SIBs will be easily reusable also by developers not specialized in secure systems, as they do not need to be aware of the enforced security properties: they can simply include these SIBs in their designs, and rely on the safety controller for guaranteeing the security of the application models. We are designing a methodology as a general schema that allows designers to specify system models along with their security requirements and then use DIME to automatically generate the actual system architectures from the own models plus the security aspect once available as sketched in [13], including complete, configured access control infrastructures. This way, we pave the way towards an integrated system which closes the above mentioned gaps: the gap between security models and system design models, and the gap between design and implementation. We accomplish this by a model-driven development process where security is explicitly integrated in all the phases of the design process. To make a high-level SLG security aware, we combine it with low-level SIBs implementing security primitives, intended for guaranteeing privacy, integrity and authenticity of all its composing SIBs (or of the most critical ones). The security controller checks at design time the correctness against static rules about, e.g., allowed roles, users, and permissions, and at run-time evaluates dynamic code to cover monitoring and run-time protection aspects.

In our approach, we have defined three basic levels of security for the domains we consider.

No security The first level, representing the lowest level in our classification, is reserved for SIBs which do not require to be protected, either because they do not perform mission-critical operations, or because they do not interact with other machines nor hardware, so there is no communication to secure. Still, an attacker could exploit vulnerabilities in the code of these SIBs, but could not gain advantages to compromise the execution of the whole SLG. SIBs belonging to this level of classification do not have a graphical marker.

Medium security: secure SIBs The second level in our classification is reserved for SIBs which are still not mission critical, although they perform inter-machines actions, for instance exchanging messages over the network, thus requiring the establishment of a secure and confidential transmission line, and the verification that the user executing the SIB has the right permission to do so. Secure SIBs belonging to this level of classification are graphically identified by a surrounding yellow box in our DIME models. They can use the Data at rest/Data in motion libraries of [12], or the design and protection mechanisms introduced in [10] and [13].

High security: critical SIBs The third level, finally, is reserved for mission critical SIBs. These SIBs execute operations which, if changed or disturbed from a malicious attacker, could compromise the execution of the whole SLG. In addition, their operations require both inter-machines communication and communication to/from hardware devices, that need special protection to ensure that attacker cannot change or interfere with them. For instance, these SIBs check both the user executing them, to evaluate if it belongs to a group of authorized subjects thus having the right permissions, and the hardware, verifying its integrity and the device characteristics (e.g., ID, provider) against a list of authorized devices. Critical SIBs belonging to this level of classification are graphically identified by a surrounding red box in our DIME models.

In our designs the layout of secure and critical SIBs is not different from the layout of the insecure SIBs within DIME. This is so as our aim is to make the task of switching between insecure and secure SIBs the easiest possible even for users which are not at ease with principles behind the XMDD. However, secure and critical SIBs are clearly contained in dedicated palettes, and so are easily distinguishable. As a rule of security, we impose that each SLG must be identified with an overall security level which cannot be lower than the one of any of its constituting SIBs. For instance, a SLG containing a second level SIB cannot be seen as a SIB of first level from any of the higher hierarchy level SLGs.

IV. Secure remote control of anthropomorphic actuators

The Human-Machine Interaction (HMI) paradigm is traditionally dominated by direct manipulation and physical interfaces (e.g., gloves, keyboard, joystick). These interfaces are often cumbersome and inadequate, and require dedicated training phases and calibration procedures. In addition, they
are often not suitable for people with disabilities. A valid alternative comes from modern motion tracking technologies, which instead offer many advantages in terms of usability, reduced costs and learning time, and do not require calibration procedures. In the past, we have already investigated vision-based interfaces for remote control of machine and robotic actuators in assistive [18] and rehabilitative [19] applications.

A. The Models

We present a case study based on a gesture-based communication pipeline to remotely control robotic actuators (e.g., an exoskeleton) leveraging security primitives and secure SIBs with the aim of guaranteeing confidentiality of the exchanged data and security, meaning that robotic actuators can receive and accept only valid inputs coming from authorized devices. The secure pipeline has been thought for applications in the field of remote communication among deaf-blind [18], but it could also be employed in similar domains or in other fields such as tele-rehabilitation.

An example of another possible usage of the secure communication pipeline is to enforce a remote communication system for deaf-blind people. Such “tele-signers” require signs from tactile Sign Languages to be recognized on one side resorting to vision applications and to be reproduced on the other side resorting to anthropomorphic actuators that are intelligible to the deaf-blind recipient. The pipeline is represented in Fig. 1. The reader may refer to [18] for further details.

For each operation which needs security, and especially for mission critical operations, two versions of the corresponding SIB have been designed: the “regular” one (i.e., without particular attention to security aspects) and the secure one. Each SLG in every pair is easily interchangeable in DIME by simple replacement via drag and drop; acting like this, it is possible to secure even complex SLGs and models in a few minutes by simply replacing critical sub-parts with their secure version. SLGs defining the models are organized in a hierarchical way, in order to improve their readability and especially their reusability. In fact, SLGs consist of an assembly of orchestrated SIBs as explained in [10], but can also be used themselves as higher level building blocks within higher level SLGs. This refinement by hierarchy concept is very powerful. It helps in particular to manage distributed development (when certain SIB palettes belong to a certain team or provenance) and distributed management and maintenance concerns, supported by rich and semantic interface definitions that are a form of contracts and Service Logic Agreements (SLAs) expressed by means of properties.

In our concrete application, the communication we wish to secure is based on the Robot Operating System (ROS). ROS [20] is an open-source, meta-operating system for robot software development that provides a collection of packages, software building tools, and an architecture for distributed inter-process and inter-machine communication. The building blocks of ROS-based applications are called nodes. A node is a piece of code which implements a specific functionality, described in a proper SLG within the XMDD framework. Nodes interact with each other by subscribing or publishing messages on specific ROS topics. The communication between nodes is based on the TCP network protocol, thus native ROS does not guarantee any security.

The lowest level in the hierarchy is represented by the ROS node which has access to the input device (Microsoft Kinect or Leap Motion) to extract data that will be processed later on. Since data are represented from joints identifying human hand skeleton, the node is named SkeletonExtractorNode. The DIME model of such a node is shown in Fig. 2. It is possible to notice the clarity of such a design in comparison with code. This makes the model understandable also for ROS experts with minimal, or none, expertise of the actual application. Each ROS node has to be initialized with a minimal set of parameters including a string identifying human hand skeleton, the node is named SkeletonExtractorNode. The DIME model of such a node is shown in Fig. 2. It is possible to notice the clarity of such a design in comparison with code. This makes the model understandable also for ROS experts with minimal, or none, expertise of the actual application. Each ROS node has to be initialized with a minimal set of parameters including a string identifying its name, a string identifying the ROS package name and a Boolean node indicating whether the node itself presents a unique name or it must be anonymized. Initializing a ROS...
node is done by publishing its existence to the network of intra- and inter-machine communications, to let other nodes communicate with it via message exchanging on topics.

The first SIB right after the start SIB is named ROSInit, it is in charge of such an initialization, and represents also the first example of SIB for which we have provided a secure version.

This SIB is defined in DIME as follows:

```plaintext
sib ROSInit : communication_pipeline.ROS#rosInit
    packageName : text
    nodeName : text
    anonymous : boolean
    -> success
    nodeHandle : integer
    -> failure
    errorMessage : text
```

meaning that this SIB is defined in the class ROS within the Java package communication_pipeline.

This SIB accepts three input parameters ("packageName", "nodeName" and "anonymous") and provides two output branches: when errors or exceptions occur (e.g., it was not possible to communicate with the ROS network, maybe because ROS environment is not present in the machine executing the node) the "failure" branch is executed, returning "errorMessage" as output; otherwise, the node has been correctly instantiated and a handle will be returned as outcome of the success branch. Note that SLGs are orchestrations, meaning that (unless there is fork/join parallelism) there is a single threaded execution. For this reason, upon execution SIBs activate only one of their outgoing branches, namely the one corresponding to the right continuation in the control flow according to their execution’s outcome.

In the Data model view, when failures occur, the corresponding error message is put in an appropriate variable (as shown in the top data container in Fig. 3) and the failure outgoing branch of the SLG is executed, causing in this case the error message to be printed and the control flow to return to the upper level in the hierarchy from the failure condition. We see here how easy it is to model the data flow in DIME, as data are connected and shared as any other resource. Note also
the independence of this representation from any particular paradigm of communication (equivalent to a PIM/CIM in traditional MDA): it can be a shared memory variable, a file store, or a messaging mechanism. The precise nature can be designed by refining it to a specific technological option.

Otherwise, the control flow proceeds by executing the subsequent SIB, named ROSGetParams, which identifies and defines proper parameters of ROS to ensure the correct execution of the SLG, i.e., of the node itself. The ROSInit node is not a mission critical node and has not been represented with a secure SIB in Fig. 2. Even if it failed due to an attack, the whole ROS communication pipeline would not start, thus the attacker could not spoil information from the input devices nor command maliciously the output robotic interfaces. Nevertheless, this node requires inter-machine communication, since data from the node itself (e.g., its name) and from the user launching it have to be propagated through the ROS network, which is not intrinsically secure. For this reason, we have prepared a secure version of this SIB. Even if the layout of the SIB does not change, so that the two ROSInit SIBs are easily interchangeable, their implementation is different: in fact, the secure SIB will rely on a trusted network for data exchange, and will check whether the ROS environment is consistent and coherent (by checking that its version number is correct) and whether the user is authorized to initialize ROS nodes on that particular ROS environment.

The authorization is defined in a fashion similar to the example and discussion in [10]. Similarly to the SIB organization and management explained in [13] on the cryptography example, the secure SIB is defined within a different package (i.e., we have defined a secure package to substitute the communication_pipeline one) and a dedicated palette, so that it is immediate for users to distinguish them, and to choose between insecure and secure versions of critical SIBs.

B. Dealing with Security Properties

In this section, we focus on properties that have been identified by the designers of the application. We first list the properties and then discuss techniques that can be used to enforce their correctness.

1) Property 1: any data variable within the data flow must have a unique data type, which has to be coherent and fixed through all the life of the variable itself;
2) Property 2: any SIB and any SLG which expect a set of input parameters must be executed with a set of parameters that fit the expected one for its size, and each of these parameter must be actually linked to a data variable of a proper type;
3) Property 3: any SIB and any SLG must provide at least one outgoing branch, and can terminate with the execution of only one outgoing branch;
4) Property 4: any SIB and any SLG which provide a set of output parameters must return a set of parameters that fit the expected one for its size, and each of these parameter must be actually linkable to a data variable of a proper type;
5) Property 5: it is not possible to initialize ROS nodes if a proper version of the ROS environment is not installed and active;
6) Property 6: it is not possible to execute ROS nodes that have not been initialized;
7) Property 7: it is not possible to subscribe to unadvertised ROS topics;
8) Property 8: it is not possible to read from ROS topics if publishers have not written data;
9) Property 9: it is not possible to read from input devices information in a format different from the one they support;
10) Property 10: it is not possible to send commands to output interfaces if no data has been read from input;
11) Property 11: it is not possible to receive input data at a frame rate higher than the one supported from the input device; interfaces;
12) Property 12: it is not possible to send commands to output interfaces starting from data not acquired from ROS and the input devices;
13) Property 13: it is not possible to send to output interfaces configurations that they cannot reproduce;
14) Property 14: access to hardware interfaces must be realized through secure primitives;
15) Property 15: hardware interfaces must be checked against a list of valid supported devices.

C. On enforcing properties

Formal verification and software quality assurance offer a rich variety of techniques to enforce security/safety properties. Examples of such techniques include model checking, testing, runtime verification techniques, and synthesis or correct by construction techniques.

1) Model Checking [21] allows us to verify a model of the system and all its executions with respect to some properties expressed in a logic, frequently resorting to temporal logics. Those logics allow the expression of predicate-based assumptions on states combined with temporal ones on the sequence of states (e.g., eventually, the state q will satisfy the predicate). Those techniques are used at design time, and use the knowledge of the entire model that they are able to see. For example, software model checking checks properties of models of the code (this is useful e.g., to verify for example the correctness or security of SIB implementations), while behavioral or system-level model checking is more abstract, and sees behavioral elements (components, services, or in our case SIBs) as uninterpreted propositions, thus can check very efficiently their correct compositions. Model checking is successful and efficient for boolean properties (yes/no questions), posed to a large variety of system model types with a wide range of expressiveness, but has difficulties with other types of variables, albeit SMT-solvers [22] and systems with integrated decision procedures (e.g., for arithmetics) have helped to make progresses there. Examples of successful classical model checkers include
3) **Runtime techniques** [30], [31] allow us to monitor an execution of the system and eventually take decisions in case some red light is crossed. One of the best example of the usefulness of such techniques is in anti-virus and malware detection, where the execution of the code is monitored until completion, or until a series of flags have been passed (in which case an exception is thrown). Those techniques are of interest when we only have a partial view of the system and we have no prior knowledge on the environment that is used. Their require that we can instrument the system so that the necessary information can be monitored. This can be done by modifying the code in a white or grey box fashion, in case we have the code and thus can augment it with runtime assertions or checks, for example in JML for Java programs, or if we are able to overlay the monitoring as a woven aspect in an aspect oriented development paradigm. If this is not possible, a black box approach includes having a model of the behaviors one wish to monitor run in parallel with the actual system under consideration, and use the (usually state-transition) model as an abstract indicator/predictor of the health of the system, stopping the execution when the model foresees some “safety” threshold trespassing.

Of course, in this case one learns by failure, as those models can be only refined and improved after having experienced their inadequacy.

4) The **correct by construction** paradigm is attractive because instead of leading to repairing errors, be it in some system models at design time or in the system itself at runtime, it helps prevent the insurgence of non-conform behaviour at all. It assumes that there are strong hypotheses on the “knowledge” these approaches have of the system under design and its elements, either in terms of the system itself (code if software, the device if hardware), or of properties guaranteed by a trusted third party. For example, depending on what one has and what one trusts, termination properties of a SIB can be ensured by code inspection, software model checking, by testing, or by experience of the community (if it is a widely used library and has been debugged by myriad users). It also assumes that there is total control on the (quality of the) tools that are used to develop the system. These hypotheses are there to guarantee that the system will satisfy certain properties at design time because it has been built from certain conforming entities and using techniques that are known to enforce or preserve those properties of the product. As an example, the BIP toolset guarantees that if two components are secured, then this will also be the case of their composition, this being achieved by restricting the way components can interact together. Other examples are on restricting the high-level languages to those whose type can be controlled. For example, this typically excludes programming languages typed at runtime as unsuitable, but also prohibits constructs that may give rise to some risks, which is the motivation behind the existence of MISRA C [32]. Successful examples of correct-by-construction are high-level hardware synthesis [33], as well as workflow and process synthesis [34], [35]: they have in common that they do not strive to create the circuit or the code from scratch, but (like in our approach) they correctly choose, configure, assemble and wire their respective products starting from libraries of elements with characterized properties, that have been established independently.

With this in mind, the choice of which techniques to adopt and where (the scope) and when (in which phase of the product construction) to enforce them largely depends on various criteria that include (but are not limited to) 1. the nature of the property (behavioral, structural, involving variables...), 2. the best moment to verify the property (at design time, at deployment time...), 3. our knowledge of the environment (do we have an estimate of all the variables, do we have knowledge on the average user), and 4. the restrictions we permit on the design process (e.g., is this reasonable to forbid the user to write in python).

We now briefly examine our properties and discuss which technique could be applied.

We first observe that we have full control on Property 1 to Property 4: indeed, they are environment-independent, and
concern things (SLGs and SIBs) that live within DIME, which we control ourselves. We can thus enforce these properties by construction, acting on DIME. We distinguish two categories of properties: correct typing, and morphology.

1) **Properties 1, 2, and 4** concern correct typing. They are independent of the particular application and of the application domain: they should apply to every SIB and every SLG. As we describe in [11], through the use of Cinco and of DyWA, they are ensured by construction on any application built using the framework. This is an example of the advantage for designers of using frameworks with rich formal semantics.

2) **Property 3** is a well-formedness property on the morphology of SIBs and SLGs at the graph level (indepedently of any meaning of its elements). This is taken care of by the design in Cinco of the DIME framework, as described in [10] and [11]. Here, we assume that the code used in each SIB terminates, and we only focus on the flow of operation at the SLG level, inter-SIBs.

Properties 5 to 15 on the contrary are application domain specific, dealing with ROS. Depending on our knowledge and control of the ROS, we will have to use different approaches.

We first observe that all those properties can be encoded with temporal logic. Indeed, they refer to sequences of operations and predicates on given states. However there are differences in the nature of information they manipulate. As an example, Property 6 makes hypothesis on the sequence of operations, while Property 5 looks similar, but also make assumption on the ROS version, which is not captured in the model. Accordingly, we are more likely to be able to verify Property 6 with Model Checking, while we can only do this for Property 5 if we have this knowledge on the environment. In case we lack this knowledge, we can only enforce the property at runtime, or at construction time by imposing that we only communicate with ROS of some specific version.

With this distinction in mind, Properties 7, 8, and 10 are similar to 6 and Properties 9 and 11 are similar to 5. The similarity between 6, 7, 8 and 10 is obvious, and we propose to model check them with our GEAR tool. The similarity between 5, 9 and 11 is more intriguing. In fact, Property 9 is a domain specific instance and refinement of property 2 and 4: information formats can be described by means of types. This information would be present in a (possibly ontological) description of the devices, making it possible to determine the format compatibility in a static way. Property 11 is strictly dependent on the input device used within the application. It can be verified by deriving a limit from the technical specifications of the input device (e.g., we know that a camera like Microsoft Kinect cannot provide more than 30 frames per second) and then checking that no one of the SIBs managing input data works at an higher frequency than this limit.

The last four properties are concerned with communication primitives and interfaces. This means that we will mostly enforce them by construction. As an example, Property 12 is concerned with the way ROS uses to communicate among nodes and the control flow within our models. By construction, it should not possible in our models to admit data which is not coming from authorized input devices and ROS nodes; external data may derive from attackers or unauthorized nodes or processes. On the other hand, Property 13 derives from a security request by the hardware output interface(s), that could be damaged (or cause problems or incidents) when forced to reproduce configurations that lay beyond a safe working area, defined within the output driver. Property 14 is actually a security request specification, and needs to be dealt with by overlaying a security layer on top of those primitives dealing with hardware, as explained in [13]. Also, there we have a need to identify and then deal with Data at rest/Data in motion aspects, along the lines of [12]. An easy way to enable this is to identify the “Hardware” SIBs with a hardware label (an atomic proposition, for the use in LTL or CTL properties), so that we can use this and other similar information in possibly layered properties. Finally, Property 15 seems to request a pre-processing that does this check: it would be either an extension to the SLG that does this, if it is performed every time the SLG is executed, or a distinct process e.g. in case the application is configuring a laboratory with a number of devices, and then these devices stay fixed for a while until a new reconfiguration happens.

V. CONCLUSION

In this paper we have discussed an approach to overcome difficulties and gaps which are typically encountered when dealing with security-oriented model-driven approaches. As state-of-the-art MDS approaches address security only at a late stage of development, they are not suitable for the needs of modern companies and industry in general, requiring a fast turnaround at low incremental costs. Instead, we showed how an OTA-based XMDD approach can systematically help integrate security ab-initio in applications created within the SEcube platform. Since our approach is based on a set of reusable SIBs organized within dedicated palettes in DIME, it decouples the issue of guaranteeing that the SIBs are correct and secure from the issue of analyzing the applications, which can be greatly simplified by knowing the characterization of each SIB in advance. We applied our approach to the concrete realm of computer vision steering robotics, presented the safety and security properties elicited on the specific case study, and discussed the ways they can be enforced.

We think that this work paves the way to a novel user-centered programming paradigm, allowing an easy, and almost hidden, integration of security domain-dependent, but application-independent, properties in almost any field of modern IT, that could be of great interest for companies and industry.

ACKNOWLEDGMENT

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094, ICT COST Action IC1204: Trustworthy Manufacturing and Utilization of Secure Devices, and co-funded under the European Regional Development
Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

REFERENCES


