PADERBORN UNIVERSITY
The University for the Information Society
Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Software Engineering Research Group

Model-Driven Information Flow Security Engineering for Cyber-Physical Systems

Christopher Gerking

Dissertation

submitted in partial fulfillment
of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Referees:
Prof. Dr. Eric Bodden
Prof. Dr. Ralf H. Reussner

Paderborn, June 2020

Abstract

In cyber-physical systems, software is not only embedded into a physical context, but also used to control the communication between systems in the cyberspace. Due to the high degree of software-controlled communication, the security of the information handled by systems is at risk of being compromised. Hence, to secure the handling of information from the ground up, sound engineering principles must be applied during software development.

The theory of *information flow security* provides formal methods for security analyses of software systems, enabling engineers to detect unauthorized information flows in the design phase. However, due to specific characteristics of cyber-physical systems, these theoretical foundations are not directly applicable to the engineering in practice. For example, while cyber-physical systems involve multiple engineering disciplines, foundational works do not address such a discipline-spanning design. In addition, cyber-physical systems communicate asynchronously by message passing, whereas foundational works are based on a synchronous communication model. Furthermore, while the communication must conform to severe time restrictions imposed by the physical system context, fundamental works do not provide tool-supported techniques to analyze this real-time behavior for information flows.

In this thesis, we extend the scope of applicability by integrating formal methods for information flow security into a model-driven engineering approach for cyber-physical systems. Initially, we enhance a discipline-spanning specification technique for systems engineering by introducing a notion of security policies, which enable systems engineers to document and validate security requirements at an early stage of development. Next, we propose a body of rules for the refinement of security policies in the context of component-based software architectures, whose constituent components communicate asynchronously by message passing. Our rules enable software architects to draw conclusions about the security of a composite system from the security policies of its constituent components. On this basis, we provide software engineers with an automated technique to verify that the real-time communication behavior of individual components adheres to their refined security policies. By conducting a time-sensitive analysis, our verification technique even allows for the detection of subtle flows in which information is deducible from the timing of messages. Finally, since several of our contributions are based on transformations of models, we present an approach for reducing the verbosity of such transformations and the associated development effort. In particular, we enable developers to infer parts of a declarative transformation definition automatically and to refine the execution of the declared transformation by means of imperative instructions.

We assessed our discipline-spanning specification technique for security policies on the basis of a quality framework for security methodologies. Furthermore, we provided formal evidence for the soundness of our refinement rules applied to component-based security policies. In addition, we conducted case studies to evaluate the accuracy of the verification results and the inferred model transformations.

ZUSAMMENFASSUNG

In cyber-physischen Systemen ist Software nicht nur in einen physischen Kontext eingebettet, sondern steuert auch die Kommunikation zwischen den Systemen in der Cyberwelt. Die von den Systemen verarbeiteten Informationen sind durch das große Ausmaß an softwaregesteuerter Kommunikation einem hohen Sicherheitsrisiko ausgesetzt. Während der Softwareentwicklung müssen daher fundierte ingenieurwissenschaftliche Prinzipien zur Anwendung gebracht werden, um den Umgang mit Informationen von Grund auf abzusichern.

Formale Methoden zur Analyse der Informationssicherheit von Softwaresystemen werden durch die Theorie der *Information Flow Security* zur Verfügung gestellt, mit deren Hilfe sich unautorisierte Informationsflüsse bereits in der Entwurfsphase auffinden lassen. Aufgrund spezifischer Eigenschaften von cyber-physischen Systemen sind diese theoretischen Grundlagen auf die Entwicklung in der Praxis jedoch nicht direkt anwendbar. Während cyber-physische Systeme verschiedene Ingenieursdisziplinen einbeziehen, setzen die Grundlagenarbeiten nicht auf der disziplinübergreifenden Ebene des Systementwurfs an. Zudem kommunizieren cyber-physische Systeme asynchron durch den Austausch von Nachrichten, wohingegen die Grundlagenarbeiten auf einem synchronen Kommunikationsmodell basieren. Während der physische Systemkontext der Kommunikation außerdem strenge zeitliche Beschränkungen auferlegt, stellen Grundlagenarbeiten keine werkzeuggestützten Techniken zur Verfügung, um dieses Echtzeitverhalten auf Informationsflüsse hin zu analysieren.

Um den Umfang der Anwendbarkeit zu erhöhen, werden in dieser Arbeit formale Methoden der Information Flow Security in einen modellgetriebenen Entwicklungsansatz für cyber-physische Systeme integriert. Zunächst wird eine Spezifikationstechnik für das disziplinübergreifende Systems Engineering um Sicherheitsrichtlinien erweitert, durch die sich Sicherheitsanforderungen bereits in einer frühen Entwicklungsphase dokumentieren und validieren lassen. Weiterhin wird ein Regelwerk für die Verfeinerung der Sicherheitsrichtlinien im Kontext komponentenbasierter Softwarearchitekturen vorgestellt, deren Komponenten durch asynchronen Nachrichtenaustausch miteinander kommunizieren. Die aufgestellten Regeln befähigen Softwarearchitekten dazu, aus den verfeinerten Sicherheitsrichtlinien einzelner Systemkomponenten Schlüsse über die Sicherheit des Gesamtsystems zu ziehen. Auf dieser Grundlage wird eine Verifikationstechnik vorgestellt, mit deren Hilfe sich die Einhaltung von Sicherheitsrichtlinien durch das Echtzeitkommunikationsverhalten einzelner Komponenten automatisch prüfen lässt. Durch diese zeitsensitive Analyse lassen sich selbst subtile Flüsse auffinden, bei denen Informationen aus den Zeitpunkten der ausgetauschten Nachrichten abgeleitet werden könnten. Da mehrere der genannten Beiträge auf der Transformation von Modellen basieren, wird abschließend ein Ansatz zur Reduzierung des Entwicklungsaufwandes solcher Transformationen vorgestellt. Dadurch werden Entwickler befähigt, Teile einer deklarativen Transformationsdefinition automatisch zu inferieren sowie die Ausführung der deklarierten Transformation durch imperative Anweisungen zu verfeinern.

Im Zuge der Arbeit wurde die disziplinübergreifende Spezifikationstechnik für Sicherheitsrichtlinien auf Grundlage eines Rahmenwerks für die Qualitätsbewertung von Sicherheitsmethodiken untersucht. Zudem wurde die Korrektheit der Verfeinerungsregeln für komponentenbasierte Sicherheitsrichtlinien formal nachgewiesen. Es wurden darüber hinaus Fallstudien durchgeführt, um die Genauigkeit der Verifikationsergebnisse sowie der inferierten Modelltransformationen zu evaluieren.

Danksagung

Bei der Anfertigung dieser Arbeit wurde ich dankenswerterweise von vielen Menschen unterstützt. Ich danke in erster Linie meinem Doktorvater Eric Bodden für die vielen fachlichen Ratschläge und die moralische Unterstützung, sowie auch für die Möglichkeit, diese Arbeit unter seiner Betreuung fortführen zu können. Meine Dankbarkeit möchte ich darüber hinaus auch meinem ursprünglichen Doktorvater Wilhelm Schäfer ausdrücken, der die inhaltliche Ausrichtung und die methodische Herangehensweise dieser Arbeit maßgeblich geprägt hat. Bestens danke ich an dieser Stelle außerdem Ralf Reussner für die Begutachtung der Arbeit, sowie Roman Dumitrescu, Gregor Engels und Ben Hermann für die Bereitschaft zur Teilnahme an meiner Promotionskommission.

Vor und während der Anfertigung meiner Dissertation konnte ich mich immer auf die tatkräftige Unterstützung vieler Kolleginnen und Kollegen aus der Fachgruppe Softwaretechnik verlassen. Euch allen danke ich für den fachlichen und außerfachlichen Gedankenaustausch, der meinen Arbeitsalltag auf vielfältige Weise bereichert hat. Besonderer Dank gilt an dieser Stelle meinen langjährigen Bürokollegen David Schubert und Johannes Geismann nicht nur für ihre Hilfsbereitschaft und die sehr gute Zusammenarbeit an gemeinsamen Publikationen, sondern vor allem für die vielen anregenden und ermunternden Diskussionen zu den verschiedensten Themen. Euch bin ich zu großem Dank verpflichtet. Darüber hinaus danke ich auch allen ehemaligen Kolleginnen und Kollegen mit denen ich im Fortschrittskolleg „Gestaltung von flexiblen Arbeitswelten“ interdisziplinär zusammenarbeiten durfte.

Für die stets nette und zuverlässige Unterstützung in organisatorischen Angelegenheiten, bei Verwaltungsfragen oder technischen Problemen bedanke ich mich herzlich bei Jutta Haupt, Jürgen Maniera und Vera Meyer, sowie auch bei Astrid Canisius und Eckhard Steffen. Ihr seid mir immer eine große Hilfe gewesen.

Während meiner Promotionszeit habe ich außerdem mit vielen Studierenden zusammengearbeitet, die ich entweder als studentische Hilfskräfte oder während ihrer Bachelor- bzw. Masterarbeiten betreuen durfte. Ich bedanke mich für die dabei entstandenen Beiträge und das angenehme Miteinander. Besonders danken möchte ich Ingo Budde, der mich mit seinem technischen Fachwissen auch bei gemeinsamen Veröffentlichungen tatkräftig unterstützt hat.

Mein Dank gilt außerdem Ben Hermann, David Schubert, Johannes Geismann, Jörg Holtmann, Marie Christin Platenius-Mohr, Matthias Becker und Stefan Dziwok für ihre wertvollen Ratschläge, die sie mir als Korrekturleser dieser Arbeit gegeben haben.

Besonders bedanken möchte ich mich abschließend bei meiner Familie, durch die ich während der Promotion jederzeit bedingungslos unterstützt und immer darin bestärkt wurde, den eingeschlagenen Weg weiter zu verfolgen. Euch danke ich von ganzem Herzen.

Contents

Abstract III

Zusammenfassung V

Danksagung VII

Contents IX

1 Introduction 1

1.1 Information Flow Security . . . 2
1.2 Running Example . . . 3
1.3 Problem Statement . . . 4
1.4 Scientific Contribution . . . 7
1.5 Outline . . . 8

2 Foundations 9

2.1 Model-Driven Engineering . . . 9
2.1.1 Metamodeling . . . 10
2.1.2 Model Transformation . . . 12
2.2 Model-Based Systems Engineering with Consens . . . 14
2.2.1 Environment . . . 14
2.2.2 Active Structure . . . 16
2.3 Model-Driven Software Engineering with MechatronicUML . . . 18
2.3.1 Design Process . . . 18
2.3.2 Component Model . . . 20
2.3.3 Real-Time Behavior . . . 22
2.4 Information Flow Security . . . 28
2.4.1 Security Policies . . . 29
2.4.2 Definitions of Security . . . 31

3 Specification of Security Policies in Model-Based Systems Engineering 33

3.1 Scientific Contributions . . . 35
3.2 Quality Factors . . . 35
3.3 Overview . . . 36
3.4 Documentation of Policies . . . 37
3.5 Validation of Refined Policies . . . 39
3.5.1 Refinement . . . 40
3.5.2 Validity . . . 42

3.6 Quality Assessment 43
3.6.1 Situational Criteria Profile 43
3.6.2 Methodology Model 44
3.6.3 Results 46
3.7 Limitations 48
3.8 Related Work 50
3.8.1 Information Flow in Model-Driven Security Requirements Engineering 50
3.8.2 Security in Model-Based Systems Engineering 51
3.9 Summary 52

4 Architectural Refinement of Component-Based Security Policies 55
4.1 Scientific Contributions 57
4.2 Requirements 57
4.3 Overview 58
4.4 Component-Based Security Policies 59
4.4.1 Example Policy 61
4.4.2 Limiting Factors 61
4.5 Policy Derivation 62
4.5.1 Derivation Rules 63
4.5.2 Example Derivation 64
4.5.3 Generalization 65
4.6 Well-Formedness of Refinements 66
4.6.1 Delegation 66
4.6.2 Assembly 68
4.6.3 Best Practices 68
4.6.4 Example Refinement 69
4.7 Composability 70
4.7.1 Defining Security 71
4.7.2 Preserving Security 73
4.8 Limitations 78
4.9 Related Work 80
4.9.1 Security for Component Architectures of Cyber-Physical Systems . . 80
4.9.2 Information Flow Security in Component-Based Software Engineering 81
4.9.3 Compositional Information Flow Security for Cyber-Physical Systems 82
4.10 Summary 83

5 A Verification Technique for Real-Time Information Flow Security 85
5.1 Scientific Contributions 87
5.2 Requirements 87
5.3 Overview 88
5.4 General Verification Approach 89
5.5 Automata Construction 90
5.5.1 Perturbed Automaton 91

5.5.2 Test Automaton . . . 92
5.5.3 Adjusted Automaton . . . 97
5.6 Case Study . . . 99
5.6.1 Case Selection . . . 100
5.6.2 Data Collection . . . 101
5.6.3 Analysis . . . 103
5.6.4 Results . . . 103
5.6.5 Validity . . . 104
5.7 Limitations . . . 105
5.8 Related Work . . . 108
5.8.1 General Verification Techniques for Information Flow Security . . . 108
5.8.2 Timing-Sensitive Information Flow Security . . . 110
5.9 Summary . . . 111

6 Imperative Refinement of Declarative Model Transformations 113
6.1 Scientific Contributions . . . 115
6.2 Overview . . . 115
6.2.1 Problem Definition . . . 117
6.2.2 Requirements . . . 119
6.2.3 Solution Approach . . . 119
6.3 Mapping Models . . . 120
6.3.1 Type Mappings . . . 121
6.3.2 Feature Mappings . . . 124
6.4 Inference Engine . . . 125
6.5 Execution Framework . . . 127
6.5.1 Execution Algorithm . . . 128
6.5.2 Imperative Refinement . . . 131
6.5.3 Language Facilities . . . 133
6.6 Case Studies . . . 135
6.6.1 Case Selection . . . 135
6.6.2 Data Collection . . . 136
6.6.3 Analysis . . . 137
6.6.4 Results . . . 137
6.6.5 Validity . . . 138
6.7 Limitations . . . 139
6.8 Related Work . . . 141
6.8.1 Mapping Models . . . 141
6.8.2 Model Transformation Generation . . . 142
6.9 Summary . . . 144

7 Conclusion 145
7.1 Summary of Contributions . . . 145
7.2 Future Perspectives . . . 147

A CoCoME Security Policies 151

B Implemented Execution Framework 155

Publications and Contributions 157

Bibliography 161

Published Works 161

Supervised Theses 164

Scientific Literature 165

Standards 215

1

Introduction

Software engineers are currently faced with a shift from traditional embedded systems to a new generation of so-called *cyber-physical systems (CPSs)* [FLV14; Lee10]. Previously, embedded software was characterized by an interplay between discrete *computation* on the one hand, and continuous *control* of a physical context on the other hand. On this basis, embedded systems have been applied to problem domains such as automotive engineering, industrial production, or health care. In contrast, next-generation CPSs introduce a new dimension besides computation and control, as they involve *communication* over networks [ZFR13]. This advanced degree of interconnection enables systems to exchange relevant information with each other [SW07]. Thereby, CPSs form a *collective intelligence* that represents the driving force behind technical innovations such as autonomous driving, smart grids, or smart manufacturing in the Industry 4.0.

The advanced interconnection of a CPS causes not only an increased communication with external actors, but also comes at a price of a larger *attack surface* [The+18] that may be exploited by malicious actors. Therefore, the *security* of CPSs is a quality property that has recently gained an increasing attention by the scientific literature [Lun+19; GK16; Gir+17; CPS17; AM17; AIS18; HLLL17]. Whereas engineering secure software is a traditional and ever-present challenge [DS00], this task is particularly challenging in the problem domain of CPSs [APN17]. The overall goal of this thesis is therefore to enable secure software engineering for CPSs.

Along with the rising complexity of technical innovations, it also becomes more challenging for software engineers to ensure that CPSs meet crucial quality requirements like security. *Model-driven engineering (MDE)* [Sch06; Sil15; BCW17] is a prominent approach to overcome the inherent complexity of CPSs [DLS12; Gra+18]. In case of MDE, models are the primary *artifacts* [Mén+19] created during development. The key feature of models is their *abstraction* [Kra07] from implementation details, enabling engineers to analyze security or other quality properties at the early development stage, prior to the deployment of a ready-to-use prototype. MDE advocates the use of *domain-specific languages (DSLs)* [DKV00; KBM16], which enable engineers to develop models by means of specially tailored notations that are commonly used in their problem domains. Studies suggest that software maintenance with DSLs is less error-prone than using general-purpose programming languages [Mel+16]. Accordingly, MDE was shown to have a beneficial effect on quality [VBT15], especially in the embedded systems domain [Lie+18; AGD18; ASSS13].

One benefit of MDE is *formal verification* [Gab+19; DKW08], analyzing models for quality characteristics with the help of *formal methods* [Pel19; WLBF09]. Formal verification techniques are frequently applied to the development of CPSs [DZGL18; ZJKK17]. On the basis of rigorously defined syntax and semantics of models, such techniques enable engineers to prove quality properties like security in an exhaustive way. As long as the verified properties of a model correspond to the real-world properties of the modeled system, the underlying formal rigor assures engineers that the system is correct with respect to security or other characteristics. This principle is known as *correctness by construction* [KW12]. In this thesis, we use formal verification as a means to underpin the secure construction of CPSs.

1.1 Information Flow Security

Formal methods for security must enable engineers to verify that no secret information is leaked to a malicious actor, which would be a violation of *confidentiality*. Furthermore, the verification must also rule out *integrity* violations, preventing malicious actors from manipulating any significant information. To verify the achievement of protection goals like confidentiality or integrity, this thesis is based on the theory of *information flow security* [Man11; HS12; Smi07], which helps detect unauthorized flows of information between specific actors. The most prominent security property in this field is *noninterference* as introduced by Goguen and Meseguer [GM82]. According to this property, security-critical information (e.g., secrets) must not *interfere* with other information considered observable by certain actors. Otherwise, the observations could enable these actors to draw conclusions about critical information. To prevent such information flows, the observable information processed by a system must not depend on critical information. Therefore, information flow is a matter of dependence [Man03, p. 5], and verification techniques must analyze the behavior of systems for unauthorized dependencies in the information processing.

In recent years, numerous information flow properties besides noninterference have been proposed [cf. Man00a; McL96; FG95]. These properties vary in their underlying *security policy* [ASL02; GM82; Jau12], which indicates what information is considered critical or observable. Thereby, a security policy defines the sources and sinks of information flows that must be prevented. Furthermore, properties differ widely in the definition of security, thereby varying the conditions under which an observation is regarded as an information flow or not. This degree of freedom leads to properties of different strength, enforcing different levels of security. Finally, properties also differ in their underlying *model of computation* that describes how information is processed during execution [MZ10; FRS05]. Trace-based approaches [Man00a; McL96] describe systems in terms of their possible execution traces. Similarly, state-based approaches [GM82] use stateful models such as automata to describe a system. In contrast, process-algebraic approaches [FG95; RS01] represent systems in terms of algebraic operations over concurrent processes. All of these models describe *reactive* systems [HP85], such as the CPSs addressed in this thesis. In contrast, we do not take into account language-based approaches [SM03], which analyze *transformational* systems using programming languages as the underlying model of computation.

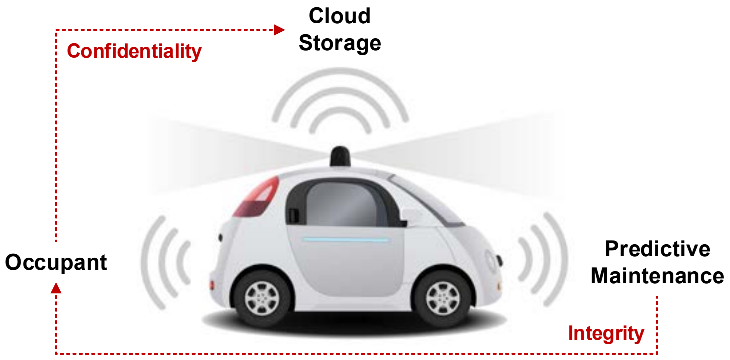


Figure 1.1: Confidentiality and integrity requirements of a self-driving car.

1.2 Running Example

To illustrate the application of information flow security to CPSs, we use a running example from the domain of autonomous driving. At present, modern cars already collect enormous amounts of data, including personally identifiable information about their occupants or owners [FPN20]. Thus, it is crucial to adopt protective measures against data theft or corruption. Consequently, the adoption of such measures is about to be regulated by the upcoming international standard for cybersecurity engineering of road vehicles [ISO20; SGM18].

As a concrete example, Fig. 1.1 shows a self-driving car that provides a user interface for the communication with occupants, enabling them to define the destination, to customize comfort functions, or to receive status information. Second, the car is equipped with an interface to a cloud storage on the Internet, which is used to store vehicle data collected while driving. Third, the car also operates an interface to a back-end for predictive maintenance, which enables the manufacturer to monitor the car's condition, diagnose faults, or tweak configuration settings using over-the-air programming.

The car's communication over these interfaces gives rise to the following security requirements. On the one hand, since the cloud provider is not deemed trustworty, data entered by occupants through the user interface must not be leaked to the cloud. Otherwise, malicious users of the same cloud storage could potentially draw conclusions about personally identifiable information. This requirement ensures confidentiality from the perspective of occupants (cf. Fig. 1.1). Thus, no information must flow from an occupant to the cloud storage. On the other hand, information displayed to occupants through the user interface must not be manipulated by the back-end for predictive maintenance. This requirement is a preventive measure, assuming that the maintenance interface might be accessed by malicious actors, who must not tamper with the information given to occupants. Thereby, the requirement ensures integrity of the information received by an occupant (cf. Fig. 1.1). Hence, information is not allowed to flow between predictive maintenance and occupants. During development, the challenge for engineers of the car is to specify such information flow requirements as part of a security policy and verify that the constructed software adheres to that policy.

1.3 Problem Statement

The paradigm of using MDE to engineer secure software systems is commonly referred to as *model-driven security* [NKKT15; MB19; Lúc+14; JJ11; KHN11; BCE11; BDL06]. On the basis of specific design notations [BSYJ17], this approach enables security properties to be analyzed and improved at an early design stage, and thereby helps engineers apply the principle of "*security by design*" [Bod18; WBM14]. Hence, model-driven security is a well-established practice in the security-critical field of CPSs [GB20; NAY17].

However, in contrast to other countermeasures adopted in the scope of model-driven security, information flow has failed to fully grow out of theory into practice. This problem of acceptance is also indicated by Mantel, who points out that "software engineering does not respect information flow security" [Man11]. The goal of this thesis is therefore to make information flow security applicable to the model-driven engineering of CPSs. To that end, we address the following subproblems created by the problem domain [*Ger16]:

Underspecification of Discipline-Spanning Security Policies. As an integration of cyberspace and physical world, CPSs combine software with artifacts from other disciplines like mechanical, electrical, or control engineering. Therefore, CPSs emerge from a multi-disciplinary engineering, which requires a close collaboration of software engineers with other specialists. Addressing this challenge by means of discipline-spanning models is known as *model-based systems engineering (MBSE)* [RFB12; MS18]. According to this approach, engineers agree on a dedicated modeling language that spans the involved disciplines. In the role of *systems engineers*, they use such a language to coordinate the disciplines at the beginning of the development. The resulting models serve as a starting point for the downstream engineering within the individual disciplines.

Information flow is an integral construct supported by modeling languages for MBSE like the standardized Systems Modeling Language (SysML) [Wol+20; OMG19]. In such languages, information flows represent data communication between systems or subsystems. However, the existing modeling languages for MBSE provide no means to restrict how the communicated data is to be processed and whether certain data is allowed to depend on other data. Thus, an open problem is that information flows are not clearly distinguished between authorized and unauthorized ones. The existing languages thereby prevent the early specification of a security policy. As a consequence, the *security requirements* [Tür17; TJM08] of a system remain underspecified at the discipline-spanning level of MBSE. Instead, the *security requirements engineering* [HLMN08; CILN02; Fab+10; MBSF10; EYLL11; SK12; KI16] is implicitly deferred to the downstream, discipline-specific phases of development. This underspecification reduces security to an *afterthought* [Ste+12] and therefore conflicts with the aforementioned principle of security by design. Accordingly, the first problem addressed in this thesis is that existing works do not provide systems engineers with sufficient means to specify information flow security policies at the early, discipline-spanning stage of development.

Composability of Component-Based Security Policies. After the initial, discipline-spanning phase of MBSE, software engineering is the discipline in which the discrete information processing of a system is designed and eventually implemented. Hence, it is the responsibility of software engineers to ensure that the predefined security policies are adhered to. Initially, engineers represent their coarse-grained design decisions in terms of a *software architecture* [Gar14]. In the role of *software architects*, they use dedicated *architecture description languages* [MT00] to describe "how the system is composed of interacting parts" [Gar14]. *Component-based software engineering (CBSE)* is a prominent principle for the decomposition of software systems during the architectural design [Val+16; Szy02]. Accordingly, systems are composed of individual software components, which interact over well-defined interfaces.

CBSE is frequently applied in the problem domain of CPSs [CMMS16]. Hence, the software of a CPS is decomposed into a set of interacting components, whereas the behavior of the overall system emerges from the component interaction. It is therefore desirable for software architects to reason about the global security of a composite system in a compositional way by analyzing local properties of the constituent components. To this end, architects must break down the coarse-grained security policy of a composite system into a set of finer-grained policies, one for each component. However, this *refinement* of security policies is a challenging task because information flow security is not generally preserved under composition [Man02]. Therefore, the security of a composite system might be violated, even if all constituent components adhere to their refined policies. To ensure security of composite systems, policies must be *composable* such that the composition of fine-grained policies restricts the information flow exactly as intended by a coarse-grained policy. However, as the second problem addressed in this thesis, existing works do not provide software architects of CPSs with guidelines for the refinement. As a consequence, the refined policies are not guaranteed to be composable and do not ensure the security of composite systems.

Verification of Security Policies in the Presence of Real Time. After the architectural decomposition, software engineers design the system behavior, which emerges from the interaction of its constituent components. To interact with each other, we expect components to communicate asynchronously by message passing [CHQ16]. The communication behavior is driven by a stateful model, which describes how a component switches from one state to another when messages are sent or received. On the basis of formal semantics, such a model defines how information is processed. Accordingly, to formally verify that a component adheres to its security policy, software engineers require techniques to analyze this model for unauthorized information flows.

Since CPSs must react to events within restricted time frames imposed by the physical context, they are *real-time systems* [Lee18]. Thus, the communication between components must satisfy hard real-time requirements. To impose such time restrictions, DSLs for CPSs consider timing as a first-class citizen [BDE13]. For example, behavioral models are frequently based on the theory of *timed automata* [WDR13; AD94].

In the context of information flow security, verification techniques must be sensitive to the imposed real-time restrictions because timing might be exploited as an implicit communication channel. By observing the point in time at which a message is passed, certain actors might draw unauthorized conclusions about critical information. To identify the presence of such *timing channels* [BGN17], verification techniques for timed automata must detect cases in which the timing of observable messages depends on critical messages. Dedicated techniques for the verification of real-time behavior exist [Wan04] and are implemented by off-the-shelf verification tools for timed automata like Uppaal [LLN18; Ben+96]. However, whereas such tools enable software engineers to verify properties like safety or liveness [AS85], they do not support the off-the-shelf verification of information flow security properties. Therefore, as the third problem addressed in this thesis, engineers are currently required to implement verification techniques for the security of timed automata from scratch, which is tedious and error-prone due to the infinite, real-valued state space that must be analyzed.

Verbosity of Model Transformation Definitions. In the scope of MDE, *model transformation* [Men13; MG06; CH06; Met05; SK03] is of vital importance to translate models from one development stage to the next, or to convert models into a form that enables them to be analyzed for quality properties like security. The execution of model transformations is carried out by means of dedicated tools [Kah+19], which are based on various model transformation languages [Heb+18; KG17]. Developers of model transformations use these languages to establish a set of transformation rules. Thereby, they give a *transformation definition* [KWB03], which is executed by the tool in use. Such languages are typically distinguished between *declarative* and *imperative* [MG06]. Using a declarative language, developers declare relations between source and target patterns of model elements, however, without having to define precisely how the relations between the actual elements are established during execution. In contrast, imperative languages are based on the procedural programming paradigm and therefore require developers to give a precise, algorithmic definition of the execution.

An open problem is that declarative and imperative languages are equally ineffective when it comes to the concise definition of model transformations. The reason is that the underlying *logic* [CH06] of real-world transformations often involves simple relations between source and target elements on the one hand, and more complex parts on the other hand. Hence, whereas declarative transformations enable a concise definition of the simple relations, they imply a verbose blowup of the set of transformation rules in order to encode more complex parts of the transformation logic. Conversely, imperative transformations enable a direct definition of these complex parts, but require a verbose encoding of recurrent instructions that are needed to establish the simple relations between elements. Such recurrent instructions are also referred to as *boilerplate* code. Thus, the fourth unresolved problem addressed in this thesis is that current approaches for the definition of model transformations require developers to cope with verbosity either way, regardless of whether they use declarative or imperative languages.

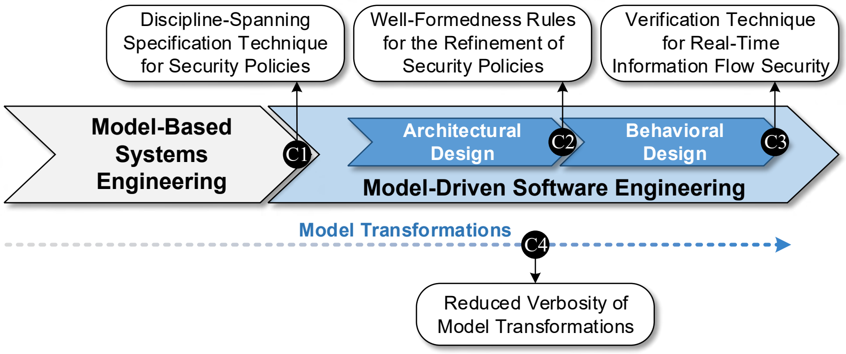

Discipline-Spanning Specification Technique for Security Policies
Well-Formedness Rules for the Refinement of Security Policies
Verification Technique for Real-Time Information Flow Security
Model-Based Systems Engineering
C1
Architectural Design
C2
Behavioral Design
C3
Model-Driven Software Engineering
Model Transformations
C4
Reduced Verbosity of Model Transformations

Figure 1.2: Overview of the contributions made in this thesis.

1.4 Scientific Contribution

This thesis proposes a model-driven approach for the information flow security engineering of CPSs. To this end, our approach provides engineers with model-driven techniques for the specification, refinement, and verification of information flow security policies. In particular, as depicted in Fig. 1.2, this thesis comprises the following contributions:

C1 At the level of MBSE, we provide systems engineers with a **discipline-spanning specification technique for security policies** [*Ger18]. To avoid an underspecification of security requirements, we integrate the concept of *flow policies* [Man03] from the theory of information flow into structural models used in systems engineering practice. This integration enables authorized flows to be distinguished from unauthorized flows and thereby facilitates the early security requirements engineering. We also address the decomposition of structural models, which requires engineers to refine the specified security policies from the system level to the subsystem level. We conceptualize an automatable validity check for these refinements, which ensures that the security restrictions of a system-level policy are enforced by the subsystem-level policy.

C2 We guide software architects during the architectural design by establishing **well-formedness rules for the refinement of security policies** [*GS19; *GS18]. To enable the specification of security policies in the context of CBSE, we classify the interfaces of a component according to the sensitivity of the information they exchange. The classification separates security-critical from observable information and thereby restricts the information flow between certain interfaces. We enable architects to partially derive such component-based policies from the flow policies used in MBSE. Furthermore, when architects decompose a component, our rules help refine its security policy into a well-formed set of finer-grained policies, one for each resulting subcomponent. A well-formed refinement ensures that the fine-grained policies are composable, i.e., they restrict the information flow as requested by the coarse-grained policy being refined. Hence, following the established rules ensures security of composite systems.

C3 Building upon a well-formed refinement of component-based security policies, we enhance the behavioral design of components with a **verification technique for real-time information flow security** [*GSB18]. After software engineers have designed the communication behavior of components using timed automata, the proposed technique checks the information processing of the automata against unauthorized information flows violating the component's security policy. In particular, our contribution is sensitive to the real-time behavior of timed automata. Thereby, it enables software engineers to detect timing channels, which are exploitable by observing the instant of time at which messages are passed. Our contribution reduces the verification to an automated *refinement check* [HBDS15], which is based on existing model-checking techniques for real-time systems. Thereby, we enable software engineers to reuse the off-the-shelf model checker Uppaal for the verification of the information flow security.

C4 At the base level of MDE, an auxiliary contribution of this thesis is a **reduced verbosity of model transformations** [*GB19; *GSB17]. For example, such transformations are used by our prior contributions to translate flow policies into component-based security policies, or to convert timed automata into a form that enables off-the-shelf verification of security properties. To reduce the verbosity of transformation definitions, we apply the principle of MDE to the transformation development itself. In particular, we enable developers to declare simple relations between source and target elements using a *transformation model* [Béz+06], which abstracts from the detailed execution of a transformation. On this basis, our contribution includes an automated engine to infer recurrent declarations from a given set of non-recurrent declarations. Hence, large parts of the transformation logic do not have to be declared manually, saving developers from the verbose definition of boilerplate instructions. Furthermore, we contribute a framework for the automated execution of transformation models. To define complex parts of the transformation logic that cannot be declared, our framework enables developers to refine the execution of a transformation model using imperative instructions. As a benefit, neither simple relations between source and target elements, nor complex parts of the transformation logic need to be encoded verbosely.

1.5 Outline

The remainder of this thesis is structured as follows. In Chapter 2, we lay the foundations by giving background information on MDE, the engineering of CPSs, and the theory of information flow security. Subsequently, we address the level of MBSE and describe the discipline-spanning specification of security policies in Chapter 3. Next, in the context of CBSE, we explain the architectural refinement of well-formed security policies in Chapter 4. On this basis, Chapter 5 introduces our verification technique for the information flow security of real-time systems. In Chapter 6, we focus on the level of MDE and present our approach for reducing the verbosity of model transformations. Finally, Chapter 7 concludes this thesis and also takes possible future extensions into account.

2

Foundations

In this chapter, we lay the foundations for the remainder of the thesis. To this end, Section 2.1 first elaborates on MDE in general. Subsequently, we focus more specifically on the use of models for systems engineering in Section 2.2, and for software engineering in Section 2.3. Finally, in Section 2.4, we address the research field of information flow security.

2.1 Model-Driven Engineering

MDE [Sch06; Sil15; BCW17] is the central development paradigm underlying this thesis, emphasizing models as the primary *artifact* [Mén+19] used to engineer software-intensive systems such as CPSs. As stated by Stachowiak [Sta73], a crucial feature of a model is that it describes a corresponding original. Thus, the concept of models is intrinsically linked to a modeling language in which this description is given. Accordingly, we define models as follows:

Definition 2.1 (Model)**.** "A model is a description of (part of) a system written in a well-defined language" [KWB03].

Note that the original system described by a model must not necessarily pre-exist, but can also be a system that is still under development. Whereas the purpose of a *descriptive* model is to document existing systems, a model that is used to develop new systems is referred to as *prescriptive* [Hel+16]. According to the constructive focus of this thesis, models are used for prescriptive purposes, thereby adopting the approach of Model-Driven Software Development (MDSD) [SV06].

Another core feature of models is abstraction from their corresponding originals [Sta73]. Thus, a model does not describe all aspects of the original system in full detail, but reduces the description to a certain extent. In the scope of MDSD, this abstraction applies to the computing *platform* [AK05] on which the final implementation of a system is eventually based. A platform represents the solution space for the problem of developing a software system [SV06]. In this context, models act as the corresponding problem description because they abstract from the platform-specific implementation of a system under development. Such a model is termed *platform-independent*, ensuring portability to different target platforms. In the remainder of this thesis, we will focus on platform-independent models.

Despite the abstraction, another feature of models is their pragmatics [Sta73]. For example, the development of executable software still requires a proper implementation on the basis of a specific platform. Thus, a platform-independent model must be pragmatic by enabling the derivation of platform-specific artifacts. A basic principle of approaches called *model-driven* is to automate this derivation [SV06], generating platform-specific artifacts such as code from platform-independent models automatically. This approach is also referred to as *generative software development* [RSVW10]. In contrast, approaches termed *model-based* reduce this degree of automation because they require a higher manual effort to develop a system, using models only as a preliminary draft. As a prerequisite for the automation in model-driven approaches, models must be based on a formal scheme that defines precisely how they are processed by a generator. Accordingly, as described in Definition 2.1, the underlying modeling languages must be well-defined using appropriate specification techniques [cf. BKP20]. In Section 2.1.1, we therefore focus on the use of metamodels to define modeling languages formally. Subsequently, in Section 2.1.2 we address model transformation as an enabling technology for the automated generation of artifacts from models.

2.1.1 Metamodeling

Unlike traditional modeling languages for MDSD like the standardized Unified Modeling Language (UML) [OMG17], current languages used in MDE tend to focus on a specific application *domain* [Sch06], which we define as follows:

Definition 2.2 (Domain)**.** A domain is “a bounded field of interest or knowledge” [SV06].

A modeling language for a specific domain is known as a DSL [DKV00; KBM16]. The domain that we address in this thesis is the field of CPSs. Thus, the modeling languages introduced in Section 2.2 and Section 2.3 are DSLs focusing on specific characteristics of CPSs. In general, a model given in a DSL must therefore be well-formed (cf. Definition 2.1) with respect to the constraints and rules of the underlying domain. Accordingly, such constraints and rules must be accounted for by the definition of a DSL. Whenever the definition of a modeling language is itself given as a model, one refers to *metamodeling* [Küh06; SRVK11]. We define such metamodels as follows:

Definition 2.3 (Metamodel)**.** “A metamodel is a model used to specify a language” [Kle08].

A model given in a modeling language is therefore an instance of its associated metamodel. In this context, a metamodel defines the *abstract syntax* of the language [SV06], thereby imposing structural restrictions on models. By contrast, a metamodel is not concerned with the notation in which models are given, which is determined by the *concrete syntax* of a language. In particular, a metamodel leaves open whether a language is textual or visual.

Throughout this thesis, we rely on the Eclipse Modeling Framework (EMF)[1] as our underlying environment for metamodeling. To enable the creation of metamodels for specific target domains, EMF provides a dedicated meta-metamodel named *Ecore*. In Fig. 2.1, we show a simplified excerpt from this metamodel in the form of a class diagram.

[1] https://www.eclipse.org/modeling/emf

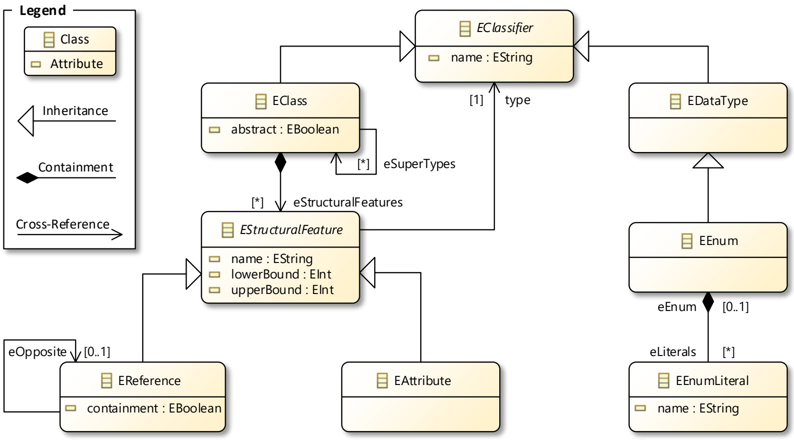


Figure 2.1: Simplified excerpt from the *Ecore* metamodel as part of the EMF.

Please note that Ecore is self-explanatory in the sense that it serves as its own metamodel. Accordingly, it enables a form of bootstrapping because the metamodel shown in Fig. 2.1 is based on the syntax defined by itself. In the following, we will therefore use the Ecore metamodel itself to exemplify its own constituent elements.

As can be seen, an Ecore-based metamodel comprises a number of *classifiers* represented by the class EClassifier, each one using a name to denote a relevant type from the target domain. Ecore supports two basic kinds of classifiers. One kind is a *data type*, which is represented by the class EDataType. Whereas primitive data types like EInt, EString, or EBoolean are already built-in by EMF, it is also possible to define additional domain-specific types. For example, an *enumeration* is a special kind of data type that is represented by the class EEnum and comprises a fixed, enumerable set of *literals*. Each of these eLiterals is represented by an EEnumLiteral, which has a name and may refer back to its comprising eEnum.

Another kind of classifier is a *class*, which is represented by the class EClass and acts as the type of one or more constituent elements of a model. Thus, each model element is an instance of a specific class from the underlying metamodel. For example, each of the constituent classes of Ecore depicted in Fig. 2.1 is actually an EClass itself. Classes can be equipped with *features* to represent characteristics of individual model elements, enabling them to feature specific values. To that end, each EClass comprises a number of eStructuralFeatures. Each EStructuralFeature is typed by a classifier, thereby defining the type of the featured values. In addition to a name, each feature is also associated with a lowerBound and an upperBound. The integer interval represented by these two bounds is the *multiplicity* of a feature, defining the minimum and maximum number of values that can be featured. If the maximum number restricts the feature to at most one value, we refer to it as a *single* feature. Otherwise, if the maximum number allows a feature to indicate more than one value, it is known as a *many* feature. For example, in Fig. 2.1, both eLiterals and eStructuralFeatures are *many* features. In contrast, the type is a *single* feature limited to one. There are two basic kinds of features, differing in the kind of EClassifier that acts as the type of an EStructuralFeature.

First, a *reference* is a feature that is represented by the class EReference, using another EClass as its type. Thus, a reference represents a relationship between classes, thereby enabling a model element to refer to other elements. A reference can be a containment such that referred elements are treated as contained by the referring element. As a result, a contained element may not exist without its container element. For example, in Fig. 2.1, the eLiterals of an EEnum and the eStructuralFeatures of an EClass are both containments. In contrast, a reference that is not a containment is called *cross-reference* and enables a referred element to exist independently of the referring element. As an example, the type of an EClassifier corresponds to a cross-reference. References are generally unidirectional, such that a referred element is navigable from the referring element only. However, a reference may use another reference as its eOpposite, which inverts the relationship between both classes. Thereby, it enables navigation in the opposite direction as well, merging both opposite references into a single, bidirectional reference. For example, the reference between EEnum and EEnumLiteral is the only bidirectional one in Fig. 2.1. Second, another kind of feature is an *attribute* represented by the class EAttribute. In this case, the type of the feature is an EDataType. In Fig. 2.1, each depicted attribute of a class corresponds to an EAttribute.

Finally, metamodels also support *subtyping* [LW94] in the form of *multiple inheritance* between classes. If an EClass refers to one or more eSuperTypes, it acts as a subclass that implicitly inherits all features from these superclasses. For example, inheritance is used by the subclasses of EClassifier and EStructuralFeature. In Fig. 2.1, both are italicized to indicate that they are abstract, i.e., elements can only be created by instantiating their subclasses.

Besides the abstract syntax, a metamodel can also impose additional conditions on the syntactical context of model elements [SV06], thereby restricting an element's features to specific, context-dependent values. Such context conditions[2] define the circumstances under which the syntax of a model is well-formed. In the remainder of this thesis, we assume that context conditions are given in the Object Constraint Language (OCL) [OMG14].

2.1.2 Model Transformation

The transformation of models is an essential part of MDE, generating development artifacts from models automatically [Men13; MG06; CH06; Met05; SK03]. In this thesis, we restrict ourselves to *model-to-model* transformations, in which the generated artifacts are again models given in terms of their abstract syntax [CH06, p. 634]. Thereby, we exclude *model-to-text* transformations [RMKP12], in which platform-specific artifacts such as code are synthesized in the form of a concrete, textual syntax. We define model transformations as follows:

Definition 2.4 (Transformation)**.** "A transformation is the automatic generation of a target model from a source model, according to a transformation definition" [KWB03].

According to Definition 2.4, we use the terms *source model* and *target model* to describe the input and output of a transformation. As described below, transformations are defined at the meta level of the respective languages in which source and target models are given:

[2] We avoid the common term *static semantics* to account for the fact that context conditions do not assign any semantics to models at all [HR04, p. 65].

Definition 2.5 (Transformation Definition)**.** "A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language" [KWB03].

A transformation definition is usually given in a *model transformation language* [Heb+18; KG17], which is "a language that provides a set of constructs for explicitly expressing, composing, and applying transformations" [SK03]. Definitions given in a particular language are executed with the help of a *transformation engine* [CH06], as provided by dedicated transformation tools [Kah+19]. We refer to the set of transformation rules from Definition 2.5 as *rule set*. Depending on whether a transformation language provides a *modularity* mechanism [CH06], the rule set can be split across multiple *transformation modules*. Transformation rules encode the *logic* of a transformation, which "expresses computations and constraints on model elements" [CH06]. We define transformation rules as follows:

Definition 2.6 (Transformation Rule)**.** "A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language" [KWB03].

Transformation languages can be fundamentally distinguished according to whether the encoded rules are *declarative* or *imperative* [MG06]. In the imperative case, transformation rules define precisely *how* the source constructs are transformed into target constructs. By contrast, declarative rules only declare *what* the source and target constructs are. Thus, a description of *how* the transformation is carried out, as demanded by Definition 2.6, requires the underlying transformation engine to apply the transformation rules automatically. For example, Query/View/Transformation (QVT) refers to a standardized family of transformation languages comprising both declarative and imperative approaches [OMG16].

Model transformations can have various *intents*, which are cataloged in [Lúc+16]. This variety implies a fundamental distinction between two types of transformations, differing with respect to the uniformity of source and target metamodels. On the one hand, a transformation is called *endogenous* if the source and target metamodel are the same [MG06], such that the respective models are given in one uniform language. For example, in the scope of this thesis, we use such endogenous transformations with an *analysis* intent [Lúc+16] to analyze the source models for security properties.[3] In the endogenous case, another distinction is made between *in-place* transformations (manipulating a single model that acts as both source and target) and *out-place* transformations (separating source and target into multiple distinct models) [MG06]. In this thesis, we generally focus on out-place transformations.

On the other hand, a transformation from one metamodel to a second, distinct metamodel is called *exogenous* [MG06], involving non-uniform languages for the description of source and target models. As an example from this thesis, we use exogenous transformations with a *translation* intent [Lúc+16] to translate the meaning of source models into a target language. Note that, due to the non-uniformity of source and target metamodels, exogenous transformations are out-place by definition.

[3] According to Lúcio et al., the *analysis* intent is restricted to exogenous transformations [Lúc+16, p. 667]. However, endogenous transformation can be regarded as a special case of exogenous transformation. In general, transformations with an analysis intent can therefore be endogenous as well.

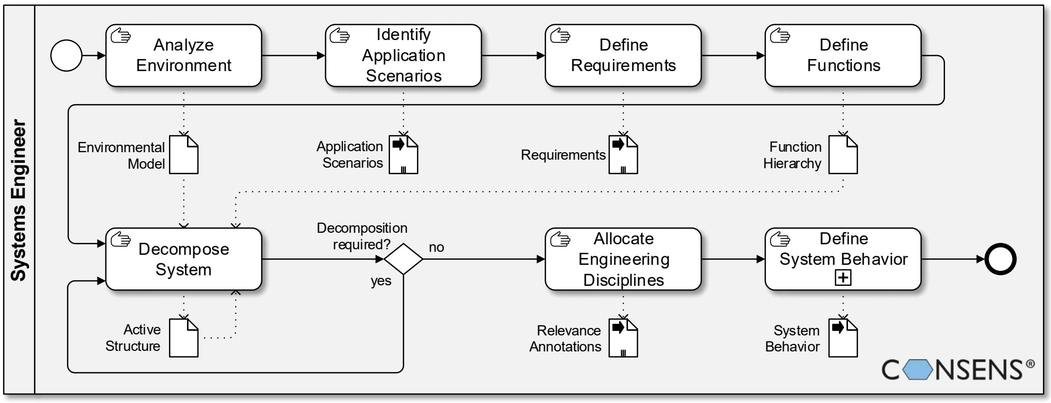


Figure 2.2: High-level overview of the Consens process for MBSE [cf. Hol+16a].

2.2 Model-Based Systems Engineering with Consens

This thesis uses Consens [Ana+14a; GFDK09] as a specification technique for MBSE. Consens provides systems engineers with a DSL for the design of *intelligent* technical systems, which manage and optimize their own operations autonomously depending on their situational context. The software underlying such systems is therefore referred to as *self-adaptive* [Wey19; Lem+13; Che+09]. Although this thesis does not focus on self-adaptation, we use the DSL provided by Consens as an enabler for MBSE of CPSs. Furthermore, we also build upon an associated process that stipulates how engineers apply this DSL systematically. The outcome of the process is a number of *partial models*, each one giving a particular *view* [BBCW19; CCP19] on the system under development. In combination, the partial models constitute the discipline-spanning design model, which is used as a starting point for the discipline-specific engineering. In Fig. 2.2, we give a high-level overview of the Consens process [cf. Hol+16a; Gau+14; GFDK09]. To describe this and other processes in this thesis, we use the Business Process Model and Notation (BPMN) [OMG13]. By focusing on the role of a systems engineer, Fig. 2.2 abstracts from the diversity of systems engineering roles [She96]. We refer the reader to [Hol+16a] for a more diverse reflection of the roles involved in the Consens process. In the following Sections 2.2.1 and 2.2.2, we introduce the process by addressing two partial models in detail.

2.2.1 Environment

Initially, in the activity Analyze Environment, systems engineers identify relevant interfaces between the system and physical or virtual elements in its environment. These environmental elements, including the corresponding interfaces, are represented in a logical *environmental model*. By focusing on the environment only, this model provides a black-box view of the system under development. In Consens, various types of interfaces can be defined by means of the following relations:

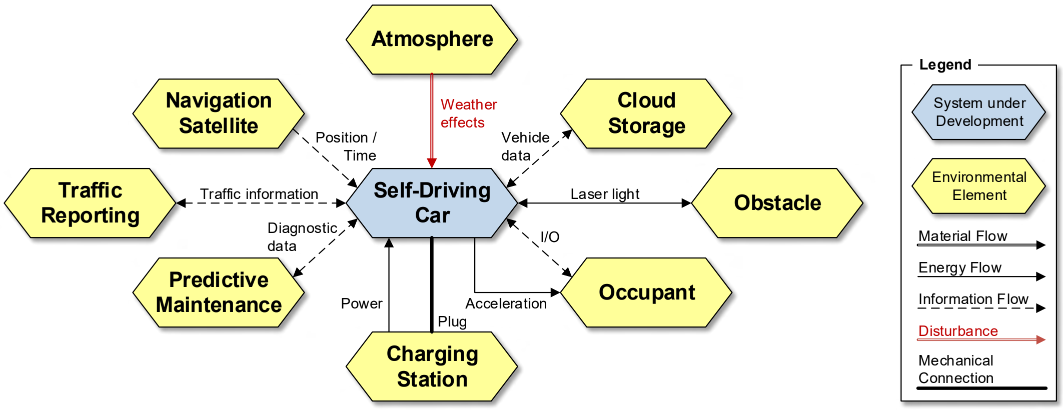


Figure 2.3: Environmental model of a self-driving car.

Material flow describes the exchange of tangible physical substance like "gases, fluids, solids, dust and also raw products, objects being tested or objects being treated" [Del+14].

Energy flow applies to intangible physical values like "mechanical, thermal, electrical, optical energy, or force and electric current" [Del+14].

Information flow refers to the exchange of knowledge about "measured variables, control pulses, or data" [Del+14] over virtual channels of communication.

Mechanical connection comprises undirected connection types that are used in connection technology [KBD16], including form-fit, force-fit, and firmly bonded connections.

As an example, Fig. 2.3 shows an environmental model of the Self-Driving Car introduced in Section 1.2. The car represents the system under development, whereas the aforementioned types of relations reflect the interfaces between the system and its environment. First, the car is exposed to weather effects, which give rise to a material flow induced by the Atmosphere. Second, the car is assumed to be an electric vehicle. To charge its battery, it may be plugged into a Charging Station, thereby giving rise to a mechanical connection that represents the plug. As depicted, unlike the aforementioned flow relations, mechanical connections are not directed to any of the connected elements. The connection with the Charging Station represents the car's power supply, which corresponds to an energy flow. Furthermore, since the car is equipped with a rangefinder, it also uses energy flows to illuminate an Obstacle with laser light and measure the light reflection. Another energy flow corresponds to the acceleration of an Occupant when carried by the car. Third, the communication paths of the car introduced in Section 1.2 correspond to information flows. In particular, traffic information is exchanged with a Traffic Reporting system, whereas the Cloud Storage is used to store and retrieve vehicle data. Another information flow represents the exchange of information with an Occupant. Moreover, the car has access to a back-end for Predictive Maintenance, which is used by maintenance engineers to request diagnostic data. In addition, by transmitting its current position and time, a Navigation Satellite enables the car to determine its geographic location.

In general, the above types of flows represent intended interfaces between the system and its environment. However, a flow may also correspond to an unintended interface that causes disturbance, but must nevertheless be handled by the system. In Fig. 2.3, this is the case for the weather effects induced by the Atmosphere. In order to distinguish intended from unintended interfaces, a flow may be marked as a *disturbance* relation [Ana+14a]. As shown in Fig. 2.3, such a disturbing flow is depicted in red inside the environmental model.

In Fig. 2.2, the subsequent activity Identify Application Scenarios leads to the informal, scenario-based specification of the system's behavior. Each scenario indicates a designated trigger and describes informally how the system behaves whenever the triggering situation occurs. In the following activity Define Requirements, the identified scenarios are further refined by specifying informal, textual requirements in tabular form. In particular, whereas the application scenarios are of purely functional nature, the specified requirements may involve non-functional aspects as well. Afterwards, engineers specify the function range of the system. In this context, a function is defined as a "relationship between input and output parameters, with the aim to fulfill a task" [Ana+14a]. As a result of the activity Define Functions, such tasks are represented in terms of a *function hierarchy*, which recursively subdivides functions into subfunctions.

2.2.2 Active Structure

The function hierarchy provides an initial indication of how a system must be modularized to fulfill particular functions. On the basis of this modularization, engineers use the activity Decompose System to describe the decomposition of the overall, coarse-grained system into fine-grained subsystems. The environmental model serves as an input to the decomposition, which results in a logical model referred to as *active structure*. Unlike the environmental model, the active structure describes not only the interfaces between system and environment, but also interfaces between subsystems. Thereby, it provides a white-box view on the system under development. As depicted in Fig. 2.2, the activity may be executed recursively to provide for the case that the resulting subsystems require a further decomposition. This recursion enables the active structure to be decomposed up to a desired level of granularity.

We depict an excerpt from of the active structure of the self-driving car in Fig. 2.4. In this case, only one level of subsystems has been established. Due to the focus on software engineering of this thesis, we restrict ourselves to nested elements that are developed with the involvement of software engineers. One of these elements is a User Interface, which handles the input and output of information from or to occupants. The Body Control Module monitors and operates multiple electronic devices inside the car and also provides the diagnostic data analyzed during predictive maintenance. Among other functions, the Infotainment System enables the car to navigate autonomously. To that end, it uses a Positioning System to determine its own location. Furthermore, to enable *crowdsourcing* of traffic information, the Infotainment System does not only receive such information from the traffic reporting, but also reports the car's own position. Another nested element represents the Electric Engine of the car. Finally, a Storage Gateway enables other elements of the system to store and retrieve vehicle data, acting as an interface to the cloud.

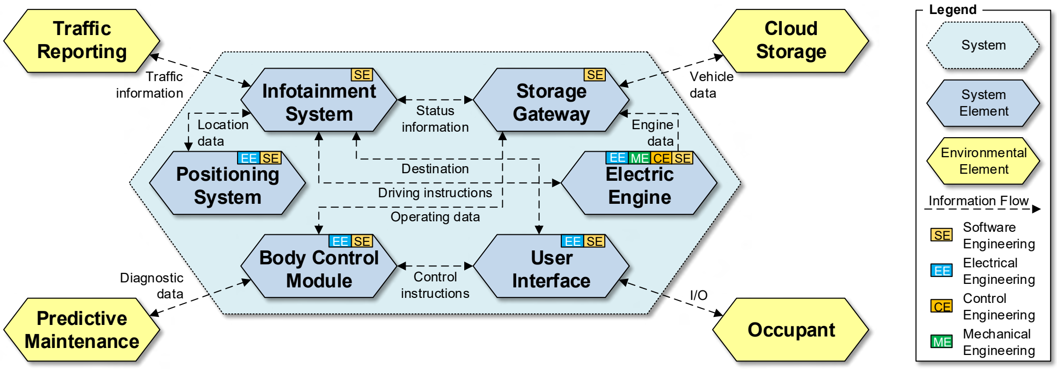


Figure 2.4: Active structure of a self-driving car.

According to the focus of this thesis, we omit material and energy flows, which are irrelevant from the perspective of a software engineer. Since such physical flows might still carry information, they are vulnerable to security breaches as well. Nevertheless, we focus on the communication between software systems or subsystems by means of information flows. For example, in Fig. 2.4, the User Interface exchanges information with both Infotainment System and Body Control Module, thereby enabling occupants to enter the destination or issue control instructions for electronic devices. Since the car drives autonomously, the Infotainment System provides the Electric Engine with the necessary driving instructions. To this end, it also retrieves location data from the Positioning System. The Storage Gateway receives engine data to be stored in the cloud from the Electric Engine. Furthermore, it is also used by the Body Control Module to store and retrieve operating data. Finally, the Infotainment System uses the Storage Gateway to retrieve status information extracted from the stored vehicle data.

In the following activity Allocate Engineering Discipline, systems engineers prepare the downstream development of the system within the individual disciplines. To that end, they annotate system elements inside the active structure with the relevant engineering disciplines that need to be involved in their development. Accordingly, the result of this activity are *relevance annotations* [Rie15] for the individual system elements. As mentioned before, all nested system elements depicted in Fig. 2.4 are relevant to the discipline of software engineering. In addition, all elements except the Infotainment System and the Storage Gateway involve electrical engineers. Finally, the Electric Engine also requires substantial contributions from the disciplines of mechanical and control engineering.

In the final activity Define System Behavior, systems engineers use various types of behavioral descriptions to specify how the system behaves in order to realize the predefined application scenarios. For example, the behavior can be described by sequences of interactions between system or environmental elements, or by means of a stateful model defining transitions between different states of the system. In this thesis, we abstract from the detailed specification of system behavior at the level of MBSE. Accordingly, in Fig. 2.2, the activity is depicted as a collapsed subprocess [OMG13], hiding the detailed specification of behavioral descriptions.

2.3 Model-Driven Software Engineering with MechatronicUML

MechatronicUML [*Bec+14; *Dzi+16; *Dzi+14; GS13; GTBF03] is a method for MDSD of CPSs, providing engineers with a design process and an accompanying DSL. It emphasizes the interplay between software and control engineering because both disciplines develop software in a broader sense. As a difference, software engineers are responsible for the discrete, stateful *coordination behavior* of a system that uses message passing communication [CHQ16] to coordinate different systems or subsystems with each other. Hence, this coordination software is located at the *cyber* level of a CPS. In contrast, control engineers develop the continuous feedback controllers that define the *control behavior* of a system, e.g., in terms of differential equations. By controlling the signal exchange over sensors and actuators, this control software represents the interface to the *physical* layer comprising the mechanical and electrical elements of a CPS. Whereas the provided DSL is dedicated to the development of the coordination software, it also addresses the integration of the control software, which is developed outside the scope of MechatronicUML. After introducing the design process in Section 2.3.1, we elaborate on the architectural design of the coordination software in Section 2.3.2 and on its behavioral design in Section 2.3.3.

2.3.1 Design Process

In Fig. 2.5, we give an overview on the MechatronicUML process, restricted to the design of the coordination software by software engineers. We refer the reader to [HSST13] for a detailed consideration of the interaction between software and control engineers, which is beyond the scope of this thesis. We also abstract from the *requirements engineering* as a subdiscipline of software engineering. In this phase, requirements engineers use the initial subprocess Specify Formal Scenarios to formally specify the intended coordination behavior in terms of scenarios, which are derived from the system behavior specified at the level of MBSE (cf. Section 2.2.2). In this thesis, we omit the details of this subprocess. Instead, we refer the reader to [Hol+16b] for the software requirements engineering in the scope of MechatronicUML and to [Hol+16a; Hol19, Chapter 3] for its integration with MBSE.

In this thesis, we focus on the platform-independent design phases of the MechatronicUML process. Initially, during the *architectural design* phase, engineers operate as software architects and design a *software architecture* [Gar14] of the system under development. Since MechatronicUML is based on the paradigm of CBSE [Val+16; Szy02], it reflects the view of architects by providing them with a *component model* [CSVC11; LW07; Lau14] for the design of component-based architectures. On the basis of the MechatronicUML component model [Hei15, Chapter 3], the outcome of the activity Derive Component Architecture is a composition of components, which is derived from the *active structure* developed in MBSE (cf. Section 2.2.2). We refer to [Hol19; Rie15; Ana+14b; Gau+09] for details on this derivation. To separate different concerns into distinct and reusable components, every component might be decomposed into subcomponents during the Decompose Component activity. This decomposition is a recursive process that terminates when none of the components need to be decomposed further. We describe the component model in Section 2.3.2.

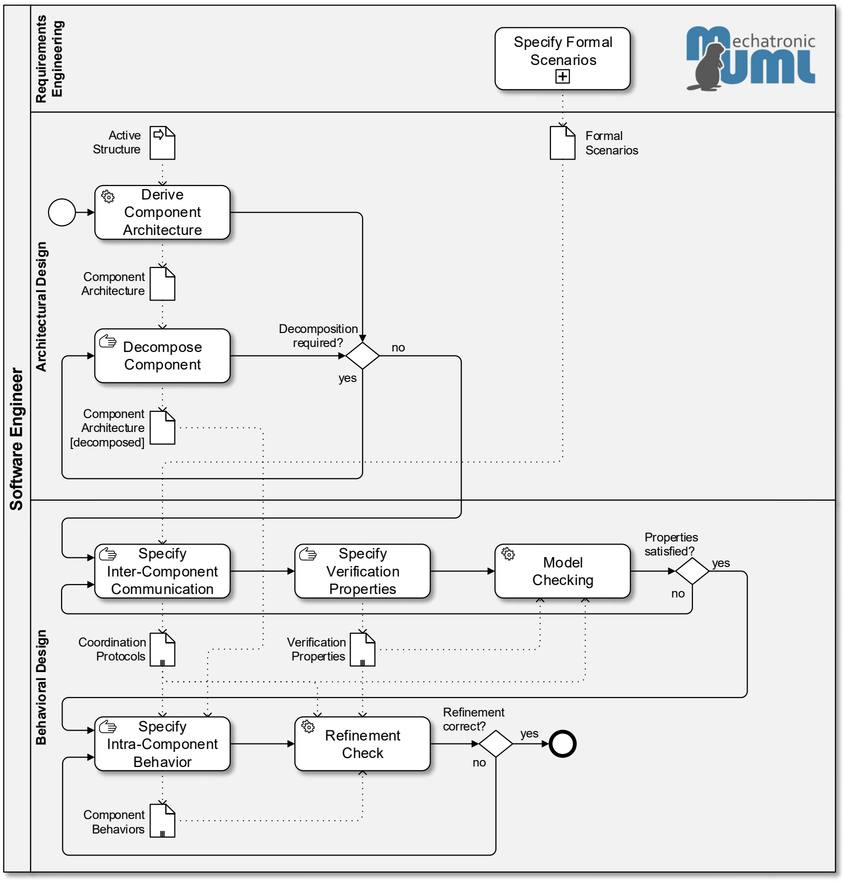


Figure 2.5: High-level overview of the MechatronicUML design process [cf. HSST13].

The subsequent phase is the *behavioral design* of the predefined software components, using a behavior-oriented view that we introduce in Section 2.3.3. Designing the coordination behavior is a two-stage process that decouples the external communication between components from their internal behavior, thereby enabling compositional verification [GTBF03]. As part of the first stage, software engineers define the application-level communication between components in the activity Specify Inter-Component Communication, thereby realizing the formal scenarios specified during the requirements engineering phase. The result is a set of *coordination protocols* [Dzi17, Chapter 3], acting as behavioral *contracts* [Mey92] for components passing messages to each other. The subsequent activity Specify Verification Properties leads to a set of properties like *safety* or *liveness* [AS85; Lam77], given in terms of temporal logic formulas [Pnu77]. These properties act as functional requirements for the coordination protocols [Dzi17, Chapter 4] and are formally verified during the subsequent Model Checking activity [*GSDH15; Dzi17, Chapter 5]. If this first step of the compositional verification fails because any of the properties is violated, software engineers need to adjust the protocols accordingly until all properties are satisfied.

The second stage of the compositional verification is initiated by the Specify Intra-Component Behavior activity. Software engineers use this activity to specify the stateful behavior that drives the message passing of each component, thereby implementing the predefined coordination protocols. However, whether these protocols have been implemented correctly depends on the result of the subsequent Refinement Check [HBDS15; Hei15, Chapter 5]. This second step of the verification indicates whether the component behavior might violate any of the predefined properties and therefore represents an incorrect refinement of the protocol behavior. In this case, software engineers need to adjust the component behavior and recheck the refinement. In case of a correct refinement, the compositionality of the verification ensures that the coordination behavior of the system satisfies the given properties [GTBF03].

2.3.2 Component Model

In general, a software component model defines "what components are, how they can be constructed, how they can be composed or assembled, and how they can be deployed" [LW07]. To support these operations, MechatronicUML provides software architects with a component model for the architectural design of CPSs [Hei15, Chapter 3]. This model enables software architectures to be hierarchically structured since components may be composed of nested *subcomponents*. Thereby, MechatronicUML stipulates a top-down decomposition of high-level components into subcomponents at the next lower level. In the scope of this thesis, a component is referred to as a *composite* component if it is decomposed into subcomponents, each forming a *component part*. The bottom-level components of an architecture, which are not further decomposed, implement the coordination behavior of the system. To this end, they encapsulate behavioral specifications introduced in the upcoming Section 2.3.3. In contrast, the behavior of composite components emerges from the recursive composition of subcomponents, enabling their behaviors to interact with each other.

To enable this interaction, components are equipped with a communication interface comprising a set of dedicated interaction points, called *ports*. For two components to communicate, their ports must be bound by means of a *connector*. MechatronicUML supports two crucial binding mechanisms. First, a connector between a port of a composite component and another port of a subcomponent is known as a *delegation* connector, enabling the composite component to delegate the interaction over its port to a specific subcomponent. The underlying binding mechanism is known as *vertical binding* [CSVC11]. Second, a connector is referred to as *assembly* connector if it connects two ports of components at the same hierarchical level, and thereby enables interaction between their behaviors. This binding mechanism is referred to as *horizontal binding* [CSVC11].

Figure 2.6 shows an example architecture for the self-driving car. The top-level composite component named Car is composed of six component parts. These parts refer to subcomponents that reflect the software-relevant elements of the active structure from Fig. 2.4. The Car uses delegation connectors to delegate the interaction with the environment to specific subcomponents. Furthermore, assembly connectors enable the interaction between subcomponents. Accordingly, the depicted connectors reflect the information flows from Fig. 2.4. In the following, we introduce the various types of ports and components separately.

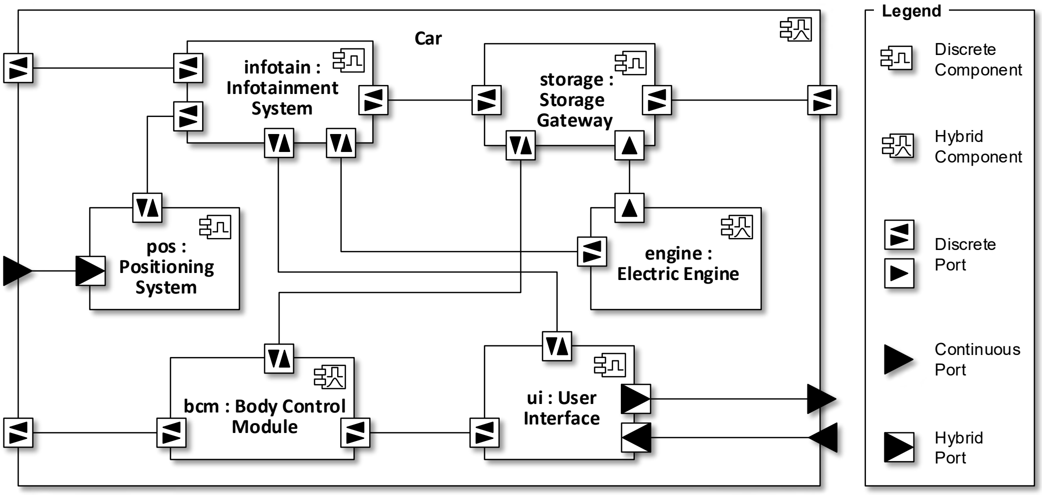


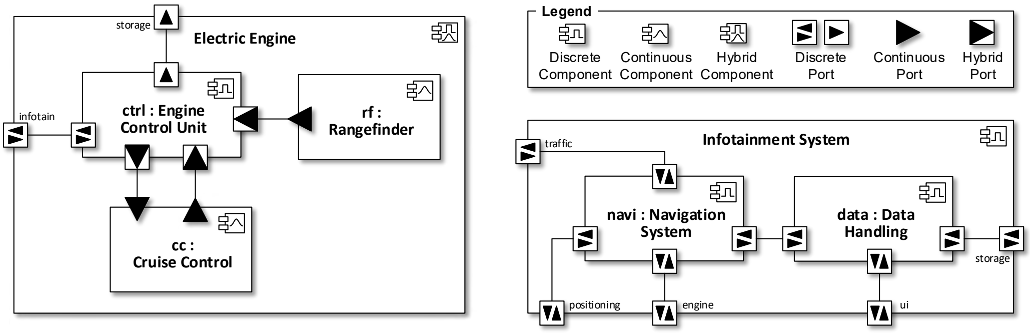
Figure 2.6: MechatronicUML component architecture of the self-driving Car.

Port Types

By providing specific types of ports, MechatronicUML supports different *interaction styles* [CSVC11] between components. First, a *discrete* port enables a component to interact by message passing, thereby implementing the coordination behavior of a CPS on the basis of the *request-response* interaction style [CSVC11]. Second, a *continuous* port represents a signal exchange between components, which realizes the control behavior of a CPS. Third, a *hybrid* port uses a sampling interval to turn such a continuous signal into a discrete value or vice versa, thereby representing the interface between control and coordination behavior. Both continuous and hybrid ports rely on the *pipe & filter* interaction style [CSVC11].

In Fig. 2.6, the communication over assembly connectors is realized by means of discrete ports, enabling subcomponents to coordinate each other by message passing. In contrast, three of the depicted delegation connectors correspond to a signal exchange. For example, the continuous signals received from a navigation satellite are sampled by a hybrid port of the Positioning System. Similarly, the User Interface is equipped with hybrid ports to sample the continuous I/O signals exchanged with an occupant through input or output devices.

As depicted in Fig. 2.6, the interaction over ports in MechatronicUML is directed. Since signal exchange is unidirectional, both continuous and hybrid ports correspond to *in* or *out* ports, supporting either the receiving or sending of signals. Similarly, message passing can be used to realize a one-way communication as well. In this case, the respective discrete ports are unidirectional *in* or *out* ports, which are either used to receive or send messages. This is illustrated by the connector between Electric Engine and Storage Gateway in Fig. 2.6, which transmits engine values to be stored in the cloud. However, a discrete port may also be declared as a bidirectional *in/out* port, which enables messages to be both received and sent. This is the case for all other discrete ports in Fig. 2.6.



(a) Electric Engine

(b) Infotainment System

Figure 2.7: Decomposition of composite components.

Component Types

Similar to ports, also components can be distinguished between discrete, continuous, and hybrid. A component is *discrete* if all of its recursively contained ports are either discrete or hybrid. Thereby, a discrete component implements coordination behavior of a system. In contrast, a component that contains exclusively continuous ports is a *continuous* component. This type of component implements control behavior of a system and thereby represents an interface to the discipline of control engineering. Finally, a *hybrid* component may recursively contain all kinds of ports. However, in MechatronicUML, the bottom-level components inside an architecture need to be clearly defined as either discrete or continuous, thereby forcing a precise distinction between coordination and control software at the lowermost hierarchical level. Accordingly, only composite components can be hybrid.

For example, Fig. 2.7a shows a further decomposition of the Electric Engine into nested subcomponents. On the one hand, both Cruise Control and Rangefinder are continuous subcomponents, which are implemented by control engineers outside of MechatronicUML, therefore using continuous ports only. On the other hand, the Engine Control Unit is a discrete subcomponent, combining discrete and hybrid ports in order to control the speed of the car. Hence, by recursively containing all kinds of ports, the Electric Engine corresponds to a hybrid composite component. In contrast, Fig. 2.7b introduces a Navigation System and a Data Handling subcomponent. Both are discrete components implemented by software engineers, thereby turning the Infotainment System into a discrete composite component.

2.3.3 Real-Time Behavior

Due to their physical environment, embedded systems like CPSs operate under hard real-time constraints, which also impact their software [Lee09]. The communication between software components is therefore restricted to specific time frames, whereas untimely communication represents a serious safety hazard. Hence, MechatronicUML uses *real-time statecharts* [*Dzi+16; GB03; Bur02] to impose time restrictions on the coordination behavior of components and thereby restrict the timing of the communication with other components.

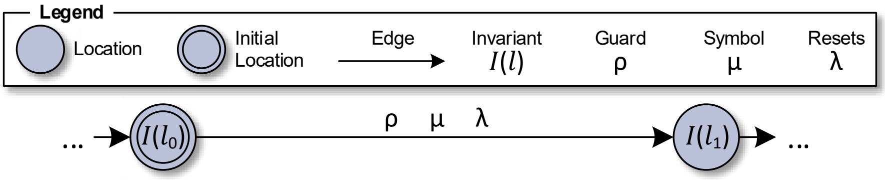


Figure 2.8: Schematic representation of the visual notation used for timed automata.

Whereas real-time statecharts combine UML statemachines [OMG17] and timed automata [AD94], their semantics is also expressible using the core syntax of timed automata [Hir04; *Ger13]. For simplicity, we therefore assume that the real-time coordination behavior of MechatronicUML is given in the form of timed automata. In the following, we will first introduce the theory of timed automata. Thereafter, we will illustrate its practical application to the coordination protocols that describe the inter-component communication, and to the intra-component behavior that implements the predefined protocols.

Timed Automata

Essentially, a timed automaton is a labeled, directed graph of locations L representing discrete states of a system, and edges E describing state transitions between locations. We introduce our visual notation of timed automata in Fig. 2.8. A dedicated *initial* location $l_0 \in L$ represents the starting state, whereas an edge that switches the system from one location to another is said to *fire*. To measure the progression of time, the state space of a timed automaton also comprises a set $\mathcal{C}$ of real-valued *clocks*. In l_0, all clocks start to run at zero. From then on, they will increase continuously at the same rate while the automaton delays in some location.

Clocks can be referred to by the labels of locations and edges (cf. Fig. 2.8), thereby enabling the specification of time-dependent behavior. First, edges can be labeled with clock *resets* as a set λ from the power set $2^{\mathcal{C}}$. Thereby, the time measurement of a set of clocks is restarted from zero when the edge fires. Second, both edges and locations may be labeled with a clock constraint from the set $\mathcal{B}(\mathcal{C})$, restricting their activity to fixed time intervals. A clock constraint is a Boolean expression that compares the current time values of specific clocks against time limits defined by constant natural numbers. Constraints may use the comparison operators $\leq$, $<$, $=$, $>$, and $\geq$ [BY04]. When assigned to an edge, a clock constraint ρ acts as a *guard* that defines a time interval in which the state transition is enabled. When assigned to a location l, a clock constraint $I(l)$ is referred to as an *invariant* that forces the automaton to not delay in l any longer than the specified time limit. A location is *urgent* if its invariant allows no time delay at all and therefore must be left immediately upon entry.

Furthermore, an edge may be labeled with a symbol μ from an alphabet Σ, denoting the occurrence of an event associated with the state transition. Whereas events are generally considered external to an automaton, internal events are denoted by the symbol $\tau \in \Sigma$. An automaton that fires an edge labeled with τ is said to make a τ transition. An automaton is *total* with respect to $S \subseteq \Sigma$ if, for each $\mu \in S$, it can always fire some edge labeled with μ, regardless of its current state. Referring to [BY04], we define timed automata as follows:

Definition 2.7. A timed automaton A is a tuple $\langle L, l_0, E, I \rangle$ where

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times L$ is the set of edges where $\rho \in \mathcal{B}(\mathcal{C})$ is the guard, $\mu \in \Sigma$ is the action, and $\lambda \in 2^{\mathcal{C}}$ is the set of clock resets,
- $I : L \rightarrow \mathcal{B}(\mathcal{C})$ assigns invariants to locations.

Auxiliary Operations. We will make frequent use of two auxiliary operations over timed automata. First, the *restriction* operator \ is used to disable all edges associated with specific events. Thus, for $S \subseteq \Sigma$, the automaton $A \backslash S$ will only keep those edges labeled with $\mu \notin S$. This is equivalent to relabeling the respective edges with a guard of *false* such that they can never fire. Second, using the *hiding* operator $/$, edges will be disassociated from specific events. Thus, in A/S, all edges labeled with $\mu \in S$ will be relabeled with τ. The respective edges will be allowed to fire even without the occurrence of specific external events.

Parallel Composition. A composite system can be composed of multiple timed automata using a parallel composition operator that we denote by || [JLS00]. The result of the composition is a *network* of timed automata, which communicate with each other by *handshake* synchronization [BY04]. Thus, an edge labeled with $\mu \neq \tau$ may only fire when synchronizing with a counterpart edge that carries a complementary symbol $\overline{\mu}$. In the following, we will therefore regard all symbols except τ as synchronizations over dedicated synchronization channels. In order to fire two edges from different automata synchronously, they must be labeled with a complementary pair of symbols consisting of a provided synchronization (denoted by *!*) and a required synchronization (denoted by *?*) over the same channel. Whenever two automata synchronize, the network will merge the synchronizing edges into a τ transition. We will also consider required synchronizations as *inputs* from a set I, whereas provided synchronizations correspond to *outputs* from a set O. We generally assume that the alphabet is a disjoint union $\Sigma = I \uplus O \uplus \{\tau\}$. An automaton that is total with respect to I is said to be *input-total*. Thus, for each possible input, the automaton can always fire an edge labeled with a corresponding required synchronization. Note that, since handshake is mandatory, the || operator differs from the standard parallel composition operator | from process algebras like CCS [Mil80]. In our case, edges may only fire in isolation if they are labeled with τ. Accordingly, the network $A_1 \parallel A_2$ is equivalent to a parallel composition $A_1 \mid A_2 \setminus I_1 \cup O_1 \cup I_2 \cup O_2$, which uses the restriction operator to disable all unsynchronized edges labeled with $\mu \neq \tau$ [Ben+96]. In contrast, synchronized edges are not affected by the above restriction because they are merged into a τ transition by the composition $A_1 \mid A_2$.

Automata may declare urgent locations as *committed* to take precedence over other automata. The entering and leaving of a committed location can neither be interrupted by time delays, nor by firing an edge inside another automaton (unless being in a committed location itself). Such locations are therefore useful to encode atomic sequences of state transitions.

Simulation & Bisimulation. To address the question whether two timed automata behave similarly from the perspective of an observer, we refer to the notion of *simulation*, which is a binary relation between the state spaces of two systems [Mil71]. For each pair of related states, a simulation requires any observable state transition that is possible in the first state to be possible in the second state as well, and that the pair of states reached by these transitions is again included in the relation. In our case, observable state transitions comprise both inputs and outputs of firing edges, as well as time delays. Thus, two timed automata are *similar* if their start states are included in a simulation. If so, whenever the first automaton can fire an edge with an input or output, or can delay by some real-valued amount of time, then the second automaton can do the same. Hence, any observations exhibited by the first automaton are also exhibited by the second one. We denote the timed *similarity* of two automata by $\lesssim$.

Moreover, to consider the question whether two timed automata behave equivalently, we may further restrict the aforementioned relation such that its converse relation is also a simulation. If so, the relation is a *bisimulation* [Sti98]. If two timed automata are *bisimilar*, then each observable state transition of an automaton is also possible for the other, and vice versa. Thus, both automata will execute in lockstep and thereby exhibit equivalent observations. We refer to this observational equivalence as timed *bisimilarity*, denoted by $\approx$.

Unlike inputs and outputs, edges labeled with τ are *not* considered observable in the above definitions. Thereby, we restrict ourselves to *weak* variants of timed simulation and bisimulation [Yi91]. Observable state transitions may thus be prepended, appended, or (in case of time delays) even interrupted by an arbitrary number of τ transitions. Hence, automata can be weakly similar or bisimilar, even if they differ in the occurrence of internal events.

Coordination Protocols

Coordination protocols impose real-time restrictions on the asynchronous message passing between two components over assembly connectors (cf. Section 2.3.2). In this context, two communicating components each fill a *role* of the protocol, whereas we assume each role to be described by means of a timed automaton. In Fig. 2.9, we illustrate an example protocol named Speed Adjustment with roles filled by the Navigation System and the Engine Control Unit. The coordination behavior implemented by the automata of both roles is based on a recurring pattern named *failsafe delegation* [DBHT12; Dzi17, Chapter 6]. Here, the pattern is used to delegate the task of adjusting the car's speed depending on its geographic position.

Since the asynchronous communication in MechatronicUML can be encoded by means of synchronous communication [Hir04; *Ger13], we denote the sending of messages by outputs (!) and the receiving of messages by inputs (?). However, according to the asynchronous communication, both automata will never synchronize with each other directly when passing a message. Instead, as shown in Fig. 2.9, the protocol assumes that sent messages are delayed up to 5 units of time before they can be received. For this assumption to hold, a certain *quality of service* (QoS) must be guaranteed by the assembly connector over which messages are conveyed. In Fig. 2.9, further QoS assumptions indicate that the connector may be unreliable, which gives rise to messages getting lost in transit, and that messages will nevertheless be received in the same chronological order in which they were sent.

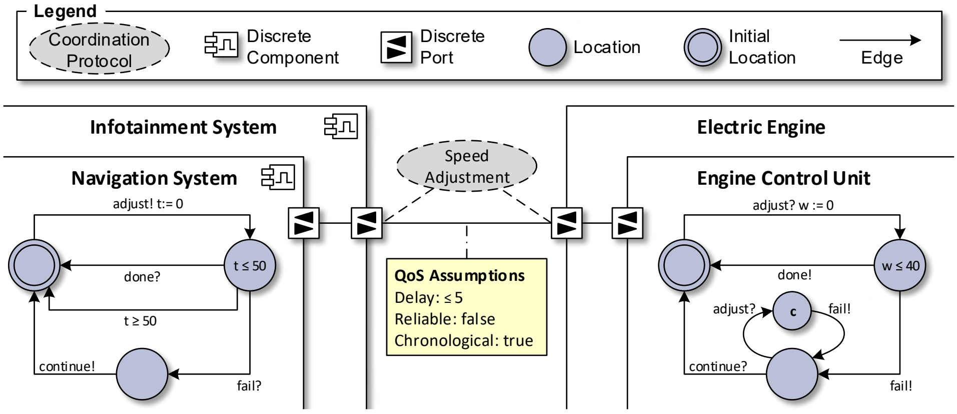


Figure 2.9: Communication behavior of the Speed Adjustment protocol.

In its initial location, the Navigation System initiates the coordination by sending an adjust message to the Engine Control Unit, thereby indicating a speed limit that must not be exceeded at the car's current position. At the same time, the Navigation System will reset a clock t, which acts as a timeout in its follow-up location with the invariant $t \leq 50$. Similarly, when receiving adjust, the Engine Control Unit resets a clock w and switches to a new location with an invariant of $w \leq 40$. Accordingly, the speed adjustment should be confirmed by means of a done message within 40 units of time. Alternatively, a fail message may be sent to decline the adjustment, e.g., because the car's pathway is blocked by a nearby obstacle detected through rangefinding. If done is delivered, both automata switch back to their initial locations. However, to deal with the case that any of the prior messages got lost in transit, the Navigation System will also switch back to its initial location when a timeout occurs because neither fail nor done was received until $t = 50$. If fail is delivered, both automata enter a failsafe location such that the Navigation System may react to the failure, e.g., by taking an alternative route. However, if fail gets lost in transit, the Navigation System may retry to adjust the speed from its initial location. In this case, a new adjust message might be received by the Engine Control Unit in its failsafe location, which it will immediately decline by sending another fail message. A *committed* location is used to combine adjust and fail into an atomic sequence. Finally, the failsafe locations of both automata are left when a continue message is sent from the Navigation System to the Engine Control Unit, thereby setting the protocol back to its original state.

According to the MechatronicUML design process from Fig. 2.5, coordination protocols are equipped with a number of temporal logic formulas, which are used to express functional requirements that must be satisfied by a protocol [Dzi17, Chapter 4]. For example, the Speed Adjustment protocol depicted in Fig. 2.9 must avoid a *deadlock*, which is defined as a state in which no future edges may ever fire, regardless of how long the underlying automata will delay [BDL04]. Under consideration of the specified real-time restrictions, such properties of protocols can be formally verified by means of model-checking techniques [*GSDH15; Dzi17, Chapter 5]. Checking the coordination protocols against these properties is the first step of MechatronicUML's compositional verification approach (cf. Section 2.3.1).

Component Behavior

On the basis of the formally verified coordination protocols, software engineers specify the real-time behavior of discrete or hybrid components, thereby filling the predefined roles of each protocol. In contrast to the behavior of a role, the component behavior may refer to continuous signals, which are sampled by the component's hybrid ports. Moreover, if a component comprises multiple discrete ports with different coordination protocols, the component behavior may also interrelate these protocols and thereby introduce intra-component dependencies between the individual role behaviors [DGB14; EH10]. In both cases, by introducing dependencies between the interaction behaviors of distinct ports, software engineers may resolve potential nondeterminism inside the predefined coordination protocols.

As an example, Fig. 2.10 shows the component behavior of the Engine Control Unit. The component comprises two discrete ports and therefore fills two distinct roles of different coordination protocols. First, the depicted timed automaton implements the Speed Adjustment protocol known from Fig. 2.9. Thus, the adjust, done, fail, and continue messages will be passed over the infotain port. As an addition to the role behavior from Fig. 2.9, the signal received over the hybrid distance port is used to resolve the nondeterminism between done and fail. Therefore, the signal sent over the hybrid speed port is only adjusted successfully when $\mathsf{distance} > 10$.

Second, the automaton implements another protocol referred to as Engine Data Transmission, which is based on a recurring pattern for real-time coordination named *periodic transmission* [DBHT12; Dzi17, Chapter 6]. The task accomplished by this protocol is to periodically store engine data in the cloud. To this end, the store message is sent over the storage port every five time units. This period is measured by a clock p, whereas the invariants of all non-committed locations of the automaton shown in Fig. 2.10 are restricted by $\mathsf{p} \leq 5$. Thereby, independent of the current location, one of the looping edges of the automaton is triggered at $\mathsf{p} = 5$ to send the store message and reset the clock p. We assume that the engine data to be stored is received by the Engine Control Unit as a signal over the hybrid diagnosis port. However, whereas the sampled signal value needs to be attached to the store message as a parameter, we abstract from this parametrization in Fig. 2.10.

Subsequent to the design of the component behavior, the MechatronicUML process includes a *refinement check* [HBDS15; Hei15, Chapter 5] to verify that the predefined coordination protocols are implemented correctly by a component. To this end, different refinement definitions including timed simulation and timed bisimulation may be used to verify that the protocol behavior preserves the verification properties guaranteed by the implemented protocols. Among other criteria, the nature of these properties will also determine which refinement definition must be selected. For example, in order to preserve the deadlock freedom of the Speed Adjustment protocol, it is sufficient to check for the existence of a timed simulation between protocol and component behavior [HBDS15]. The refinement check represents the second step of MechatronicUML's compositional verification approach (cf. Section 2.3.1). Thus, a correct refinement guarantees that the coordination behavior implemented by the components will satisfy all specified verification properties.

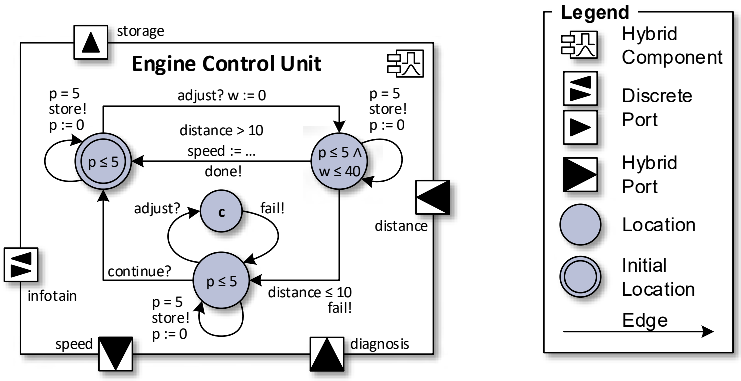


Figure 2.10: Component behavior of the Engine Control Unit.

2.4 Information Flow Security

The theory of *information flow security* [Man11; HS12; Smi07] serves as the general basis of this thesis as it provides our underlying *security model* [MSB02]. As such, it is used to define the security needs of software systems precisely. Information flow security assumes that a system exchanges information with multiple actors (which can also be other subsystems). On this basis, it restricts the information processing inside a system such that security-critical information provided by certain actors must not flow to other actors, which are only authorized to obtain noncritical information. Thus, such noncritical information must not depend on critical information, otherwise enabling unauthorized actors to draw security-critical conclusions. Information flow security addresses the protection goals of *confidentiality* and *integrity*. In the first case, confidential information such as a secret is security-critical because it must not be disclosed to malicious actors. In the second case, information is security-critical if manipulating that information must not enable malicious actors to tamper with other information whose integrity needs to be preserved. As an example, information provided by occupants is confidential and must not be leaked to the cloud (cf. Section 1.2). Thus, in Fig. 2.10, the store message is noncritical and must not depend on the critical adjust or continue messages.

A security breach is possible through both *explicit* and *implicit* information flows [DD77]. In case of explicit flows, information is made accessible to certain actors directly. In contrast, implicit flows enable actors to draw indirect conclusions about information from other, directly accessible information [KHHJ08]. Both forms of information flow establish communication channels between actors through a software system. In this context, implicit flows are strongly linked to the notion of a *covert channel* [JK11; JKZ12]. Originally introduced by Lampson [Lam73], we use this term to denote any unauthorized channel that is parasitic because it exploits communication over other, authorized channels referred to as *overt channels*. By abusing explicit flows as overt channels, an implicit flow is a means to establish a covert channel [SM03, p. 6]. We distinguish covert channels from *side channels* [Cad11]. This term is frequently used in the area of cryptography to describe information leakage over physical channels, which are unauthorized as well but not intended for communication at all.

Another kind of covert channel is a *timing channel* [BGN17], enabling unauthorized actors to draw critical conclusions from the response times of a system. Such channels exploit the fact that the timing of noncritical information might depend on critical information, even if the information content does not. Hence, since MechatronicUML involves time-dependent behavior (cf. Section 2.3.3), we must take the respective time restrictions into account when reasoning about the presence or absence of information flows. In the example of Fig. 2.10, the timing of the noncritical store message must not depend on the critical adjust or continue messages. Otherwise, it gives rise to a timing channel from the infotain to the storage port.

To avoid information flows like timing channels, critical information must not *interfere* with noncritical information, thereby justifying the notion of *noninterference* as the most prominent information flow property [GM82].[4] To overcome the restriction of noninterference to deterministic systems, multiple alternative properties have been proposed in literature [cf. Man00a; McL96; FG95]. These properties are referred to as *possibilistic* because they leave probabilistic system behavior and the probabilities of information flows out of consideration. According to Mantel, "a security property is defined by a security policy together with a definition of security" [Man00a, p. 186]. Whereas the former specifies which information is critical or noncritical, the latter defines to what degree critical and noncritical information must be independent. In the following, we first elaborate on security policies in Section 2.4.1, before addressing definitions of security in Section 2.4.2.

2.4.1 Security Policies

In general, a *security policy* [ASL02; GM82; Jau12] is a means to specify the security requirements of a particular system. Thus, in the scope of this thesis, such policies are used to impose restrictions on the information flow through systems, thereby specifying concrete confidentiality or integrity requirements. In accordance with Mantel [Man03], the policies used in this thesis are based on three different levels of *information sensitivity*.[5] In particular, these levels separate security-critical from noncritical information. We describe the underlying sensitivity levels in the following:

Critical information must not be communicated to certain actors. Thus, such actors must neither access the information directly, nor draw any indirect conclusions about that information from other accessible information.

Neutral information may be communicated to certain actors indirectly. Thus, such actors cannot access neutral information directly, but may still draw indirect conclusions about that information from other accessible information.

Observable information is communicated to certain actors directly. Thus, such actors can access the information, but must not be able to draw any indirect conclusions from that information about other, critical information.

[4] Noninterference is predated by Cohen's *strong dependency* [Coh77] as the seminal information flow property.

[5] Deviating from Mantel [Man03], we identify the three levels using alternative terms. In particular, we use the term *critical* instead of *confidential* to emphasize that integrity requirements can be specified as well.

In the example from Fig. 2.10, the store message must not depend on the critical adjust and continue messages, and therefore represents observable information. Finally, the done and fail messages, which are neither critical nor observable, represent neutral information. Note that the neutral level implies a form of carelessness with respect to the sensitivity of information [Man03, p. 31]. Accordingly, in literature, neutral information is also labeled as "*don't care*" [HMSS07]. Unlike critical information, neutral information allows conclusions about itself to be drawn from other information. In addition, unlike observable information, neutral information also allows conclusions to be drawn from itself about other, critical information. Thus, neutral information enables a security policy to authorize certain indirect information flows explicitly. Adopting the notation of Mantel [Man03], we use C, N, and V to abbreviate the crictical, neutral, and observable levels of sensitivity.

Compared to the above sensitivity levels, a *flow policy* [Man03] is a form of security policy that is used to specify information flow requirements in a more abstract fashion. A flow policy is based on a number of *security domains*, which are used to categorize the set of actors and the information handled by these actors, respectively. Thus, a security domain encapsulates specific information that must be handled with equivalent permissions by different actors. Our example from Fig. 2.10 involves the following three security domains: (i) the observable outputs sent over the storage port, (ii) the critical inputs received over the infotain port, and (iii) the neutral outputs sent over the infotain port. On this basis, a flow policy interrelates each pair of security domains by one of three relations:

- $\rightarrow$ is an *interference relation*, representing authorized communication that makes information from one domain directly accessible to another domain.

- $\rightsquigarrow$ is a neutral relation, representing authorized communication that does *not* make information from one domain accessible to another domain directly, but authorizes another domain to draw conclusions about that information indirectly.

- $\not\rightsquigarrow$ is a *noninterference relation*, representing unauthorized communication that *neither* makes information from one domain accessible to another domain directly, *nor* authorizes another domain to draw conclusions about that information indirectly.

Regarding the security domains as nodes and the above relations as edges, a flow policy forms a *complete* graph connecting every two security domains exactly once. Thereby, the disjoint union of the three relations is the Cartesian product of the set of security domains [Man03]. The confidentiality requirement of our example can be specified using a single noninterference between the inputs of the infotain port and the outputs of the storage port, whereas the residual domains are all related neutrally. As a natural restriction, flow policies require the interference relation $\rightarrow$ to be reflexive, such that each domain interferes with itself. Accordingly, information from one domain is always directly accessible by the same domain. We define flow policies formally as follows:

Definition 2.8 (Flow Policy)**.** "A flow policy [. . .] is a tuple $(D, \rightarrow, \rightsquigarrow, \not\rightsquigarrow)$ with a set D of security domains and relations $\rightarrow, \rightsquigarrow, \not\rightsquigarrow \subseteq D \times D$ [. . .]. The relation $\rightarrow$ is reflexive, i.e., $d \rightarrow d$ holds for all $d \in D$" [*Ger18].

2.4.2 Definitions of Security

In terms of information flow security, a system is deemed secure if the observations made by certain actors during an arbitrary execution do not depend on the amount of critical information processed during that execution. However, this definition still lacks a deep understanding of the term *dependence*. In the following, we provide such an understanding by building upon the concept of perturbations and corrections [Man03]. Thereafter, we consider information flow as a so-called hyperproperty [CS10].

Perturbation & Correction

We consider a *perturbation* as a scenario in which the execution of a system is modified by reducing or extending the amount of critical information processed during that execution. Typical modifications are the situational insertion or deletion of single pieces of critical information, or the complete removal of all critical information from an execution. For example, consider an execution in which the automaton from Fig. 2.10 resides in its initial location, repeatedly sending the store message every five time units. A possible perturbation of this execution is the insertion of an adjust message at some time, enabling the automaton to reset the clock w and switch to another location. Obviously, perturbations like this must have no effect on the observable information processed during an execution, otherwise constituting an unauthorized information flow. Thus, from the viewpoint of an observer, a perturbed execution must be indistinguishable from the corresponding unperturbed execution. In the given example, the observable store messages that are sent subsequent to the above perturbation must be indistinguishable from those messages sent without this perturbation.

To ensure indistinguishability, a perturbation may be compensated by a *correction*. A correction is a follow-up scenario in which the information processed during a perturbed execution is readjusted, thereby restoring the same observations that can be made during the unperturbed execution. Corrections are insertions or deletions of information, differing in the point of time at which they are allowed to be made (e.g., only after the perturbation). Obviously, observable information cannot be used for corrections because this would enable an observer to distinguish the perturbed from the unperturbed execution. Critical information can neither be used for corrections because it might undergo perturbation itself. Hence, only neutral information can be used for corrections. For example, in Fig. 2.10, a possible correction is the insertion of a neutral done message into the perturbed execution, switching the automaton back to its initial location. As described in Section 2.4.1, such a correction does not constitute an unauthorized information flow because neutral information allows conclusions about critical information to be drawn. Thus, the perturbed and unperturbed executions are allowed to be distinguishable with respect to neutral messages. However, note that the above correction is not even required in order to compensate for the prior perturbation. As can be seen from Fig. 2.10, the automaton ensures indistinguishability of observable messages even without switching back to its initial location. In summary, the degree of dependence imposed by concrete information flow properties is defined (i) by the specific perturbations that are required and (ii) by the specific corrections that are allowed to be made [Man03, p. 36].

Hyperproperties

As described above, information flow security demands that for every perturbed execution of a system, there must be another corrected execution of the same system that is indistinguishable for an observer. Thus, an inherent feature of information flow properties is that they interrelate multiple executions of a system, thereby referring to a set of execution traces. Thus, properties like information flow are also known as *hyperproperties* [CS10] because they are properties of sets of traces. Hyperproperties generalize standard properties like safety or liveness [Lam77], which are properties of traces. As such, they consider individual executions in isolation without interrelating them.

Consequently, as pointed out by McLean [McL96], hyperproperties like information flow security fall outside the classification of safety and liveness properties by Alpern and Schneider [AS85]. This poses several challenges, two of which we face in the remainder of this thesis. First, standard techniques for the *verification* of safety and liveness properties are not directly applicable to information flow properties. Thus, alternative techniques are needed to verify that a system is secure. Second, principles for the preservation of safety and liveness properties under *composition* of multiple systems, as proposed by Abadi and Lamport [AL93], are not applicable to information flow properties. Instead, information flow properties do not generally ensure composability [Man02]. The reason is that a composite system gives rise to communication between the subsystems it is composed of. However, a constituent subsystem might not be communicative. For example, it might not send the information that is expected by another subsystem, or might even refuse to receive information from another subsystem. As investigated by Sabelfeld [Sab01], subsystems may thereby block the communication behavior of other subsystems, such that certain executions of the individual subsystems might get disabled. Accordingly, “composition reduces the set of possible behaviors of a system” [CM15], restricting its executions to a subset. In particular, the composition might disable a specific execution that is needed to correct a perturbation. If a perturbation is not correctable, observers can distinguish the perturbed execution from the unperturbed execution. Thus, information flow properties of individual subsystems might no longer hold inside a composite system because they are not preserved by subsets of traces [McL96]. Composing multiple secure systems might therefore lead to an overall system that is insecure.

3

Specification of Security Policies in Model-Based Systems Engineering

As described in Section 2.2, information flow is one of the integral forms of interaction considered in MBSE. In general, an information flow represents intended, authorized communication between systems or subsystems. Accordingly, information flow is of vital importance to satisfy the advanced communication needs of CPSs and thereby meet the associated functional requirements. However, this communication might also enable systems or subsystems to circulate information in an unintended, unauthorized way, thereby compromising security as a non-functional requirement. Hence, systems engineers frequently need to balance the functional requirements of a system with its non-functional security requirements.

MBSE with Consens enables engineers not only to represent information flows between systems or subsystems, but also supports the specification of the associated functional requirements in terms of the designated system behavior (cf. Section 2.2). However, at present, there is no solution for the specification of non-functional security requirements that would enable systems engineers to restrict the information flow with respect to confidentiality or integrity. As a result, security requirements are frequently underspecified at the early stage of MBSE, implicitly deferring their specification to the downstream, discipline-specific software engineering. Thereby, security is put at risk of degenerating into an *afterthought* [Ste+12]. Handling security requirements late is likely to create the following problems:

- Without an explicit indication of security requirements, software engineers may easily disregard their implicit responsibility for taking the required security measures. In the worst case, vulnerabilities may go unrecognized until after the deployment, leading to so-called *zero-day* exploits that take place during operation without being recognized.
- Even if software engineers accept their responsibility for confidentiality or integrity requirements, they still lack an explicit indication of the security needs that the system under development must actually meet. As a consequence, software engineers might take inappropriate security measures that do not satisfy the actual requirements.
- A late consideration of security issues may have repercussions on the upstream phases of MBSE, forcing systems engineers to revoke earlier decisions that were made without having taken security into account.

State of Research. In the past, numerous MDE approaches for security have been proposed and surveyed in [NKKT15; OD15; UFF12]. At the same time, studies suggest that security requirements engineering is still inadequately represented because the majority of approaches address the integration of security into other, downstream development phases [MNAM17]. Nevertheless, approaches like UMLsec [Jür05] or IFlow [KSBR15] do support a model-driven security requirements engineering, enabling the specification of unauthorized information flows. However, these works are based on the UML [OMG17] and thereby tailored to the discipline of software engineering. Accordingly, they do not offer a solution to the underspecification of security requirements at the early, discipline-spanning level of MBSE. In contrast, the field of MBSE has recently put emphasis on security as well. In particular, approaches like SysML-Sec [AR16] address a systematic derivation of secure systems from predefined security requirements. However, the existing works for secure MBSE do not explicitly enable the specification of confidentiality or integrity requirements in terms of unauthorized information flows. Hence, in summary, current approaches enable information flow to be considered only at the level of software engineering, whereas an early specification of unauthorized flows during the upstream MBSE is an unresolved problem.

Contributions. In response to the aforementioned shortcomings, the contribution of this chapter is a novel integration of MBSE and information flow security. In particular, we turn the environmental models provided by Consens (cf. Section 2.2.1) into flow policies (cf. Section 2.4.1). Thereby, we enable the specification of unauthorized information flows between the environmental elements of a system under development. The resulting specification clearly defines whether information is allowed to flow through the system from one environmental element to another. Accordingly, the integrated model represents a security policy for the system itself. To enforce this policy in the downstream phases of development, engineers need to ensure that the information processing inside the system prevents all of the specified unauthorized flows. Thus, along with the stepwise decomposition of the system into the active structure, we enable systems engineers to refine a coarse-grained security policy into a finer-grained policy. Thereby, we facilitate the definition of unauthorized information flows at the level of the constituent system elements resulting from the decomposition. To ensure validity of these refinements, we provide systems engineers with means to check formally that the security requirements documented by the coarse-grained policy are properly enforced by the finer-grained policy. Thereby, the proposed validity check enables engineers to systematically preserve the initial security requirements during the stepwise decomposition of the active structure. We evaluate the contributions made in this chapter on the basis of a quality assessment framework for security methodologies [UFF18].

Novelty. The novelty of our solution is the integration of formal, rigorously founded concepts from the theory of information flow into the Consens technique used in systems engineering practice. As a benefit, we enable systems engineers to *front-load* the specification of information flow policies to the early, discipline-spanning stage of MBSE, thereby resolving the aforementioned problems associated with an underspecification of confidentiality or integrity requirements. Furthermore, by taking the specified requirements into account during the stepwise decomposition of models, we facilitate a systematic, security-preserving refinement of the information flow policies along the Consens process.

Publication. The challenges addressed in this chapter were initially outlined in [*Ger16]. Among other security measures, the proposed concept has been embedded into the Consens process in a conference paper [*GGB18]. The main technical contributions described in this chapter have been elaborated and published in a workshop paper [*Ger18].

Outline. The remainder of this chapter is structured as follows. We introduce our scientific contributions in Section 3.1 and consider relevant quality factors in Section 3.2. Next, we give an overview on our solution in Section 3.3, before we elaborate on the documentation of security policies at the level of environmental models in Section 3.4. Subsequently, Section 3.5 describes the validation of refined security policies at the level of active structures. We assess the considered quality factors in Section 3.6 and discuss limitations in Section 3.7. Finally, we survey related work in Section 3.8, before summarizing this chapter in Section 3.9.

3.1 Scientific Contributions

In this chapter, we make the following contributions:

- We integrate flow policies into the DSL for environmental models provided by Consens. Thereby, we establish a visual specification technique for confidentiality and integrity requirements in MBSE, enabling engineers to front-load the security requirements engineering to the discipline-spanning stage.

- We adopt the proposed specification technique at the level of active structures, enabling systems engineers to refine the specified requirements during the decomposition of systems into subsystems.

- We propose a validity check for refinements. Thereby, we provide systems engineers with means to ensure that the system-level security policy is properly enforced by the refined policy at the subsystem level.

- We evaluate the above contributions with the help of a dedicated framework for quality assessments of security methodologies [UFF18].

3.2 Quality Factors

As a benchmark for our contributions, we refer to a dedicated framework for the quality assessment of *security methodologies* proposed by Uzunov et al. [UFF18]. By taking Consens as a basis, we regard our contribution as a security methodology because it provides engineers both with a process and the accompanying artifacts involved in that process. The assessment framework is based on different factors that make up the overall quality of security methodologies. Due to our restricted research focus on the early requirements engineering and the application domain of CPSs, we do not address the quality factors of comprehensiveness, effectiveness, and adaptability. Furthermore, we also leave a consideration of the usability factor to future works. Instead, we take the following quality factors (QF1–QF3) into account:

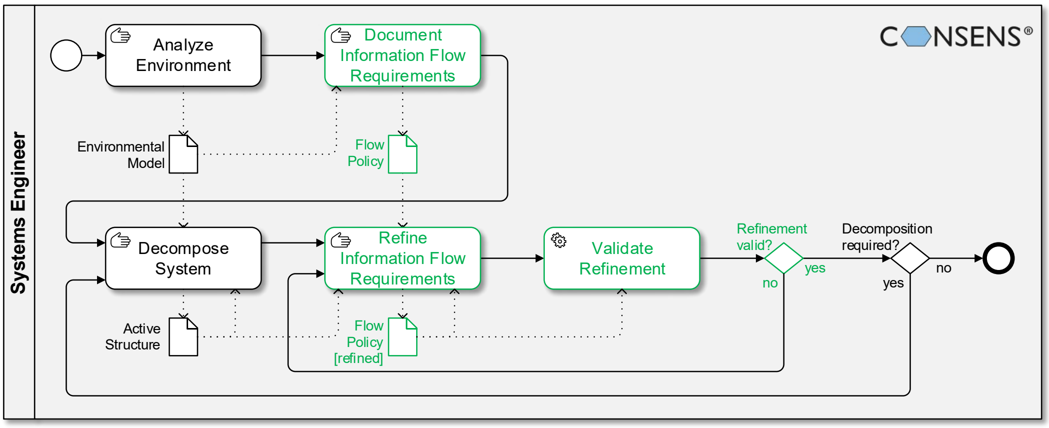


Figure 3.1: Proposed extensions to the CONSENS process.

(QF1) Correctness of Construction: This basic factor refers to the plausibility of the process that systems engineers undergo when adopting our approach. In particular, the documentation of security requirements, the refinement of the documented requirements, and the validation of these refinements must be well-ordered. Thereby, this factor seeks to answer the question whether the constructed methodology "is free from any obvious contradictions" [UFF18].

(QF2) Completeness: The question addressed by this factor is "whether a methodology possesses certain essential features" [UFF18]. In particular, systems engineers require features enabling them to document relevant security requirements early and to assure the validity of the documentation whenever the security requirements are refined.

(QF3) Assurance: This factor takes into account whether a methodology is able to give security guarantees. In our context, the decomposition of a system must "provide appropriate protection" [UFF18] with respect to the documented security requirements. Thus, it must be possible for systems engineers to assure that security restrictions at a certain level of decomposition are properly enforced at the next lower level.

3.3 Overview

We take into account the quality factors from Section 3.2 by extending the CONSENS process conceptually. Focusing on the role of a systems engineer, Fig. 3.1 illustrates our proposed extensions in green. Compared to the original process from Fig. 2.2 on page 14, we omit activities and artifacts that are irrelevant to the information flow. Instead, we focus on the environmental model and the active structure, which are partial models that inherently deal with information flow (cf. Section 2.2). To use these models for the specification of security needs, we integrate them with formal concepts from the theory of information flow.

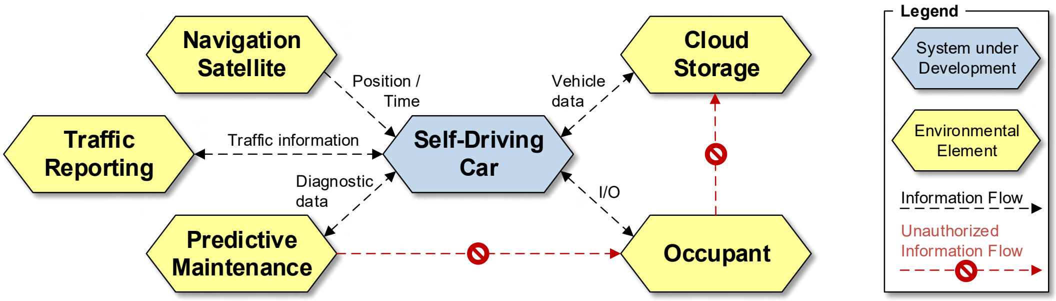


Figure 3.2: Environmental model of the self-driving car extended to a flow policy.

In particular, after the activity Analyze Environment, we propose to turn the resulting environmental model into a flow policy during the activity Document Information Flow Requirements. The documentation of these requirements is a manual step taken by the systems engineer as described in the upcoming Section 3.4. Similarly, after the activity Decompose System, we propose to turn the resulting active structure into a flow policy as well. This flow policy, which is created manually by the systems engineer during the activity Refine Information Flow Requirements, is a refined version of the prior policy documented at the level of the environmental model. We address these refinements in the upcoming Section 3.5. In response to the refinement, we propose an automatable activity named Validate Refinement, in which we check whether the refined policy is valid by enforcing the prior policy. If the refinement is invalid, the systems engineer must refine the documented requirements again. Once the refinement is valid, a recursive decomposition of the active structure gives rise to further refinements including validity checks. The final outcome is a refined flow policy that refers to the active structure and is a valid refinement of the initial policy at the level of the environmental model.

3.4 Documentation of Policies

In this section, we describe our integration of environmental models and flow policies. We treat the environmental elements, as well as the system under development, as individual security domains (cf. Section 2.4.1). Thereby, we enable systems engineers to relate the elements by means of the different flow relations ($\rightarrow$, $\rightsquigarrow$, $\not\rightsquigarrow$) known from flow policies.

Interference. We treat each authorized information flow as an interference. Accordingly, the interference relation ($\rightarrow$) includes the full set of information flows from an environmental model. In addition, since interference relations need to be reflexive in the scope of flow policies (cf. Section 2.4.1), every element is implicitly said to interfere with itself. As an example, we turn the environmental model of the self-driving car from Fig. 2.3 on page 15 into a flow policy. In Fig. 3.2, we restrict ourselves to information flows and those environmental elements that participate in such flows. Each of the depicted flows constitutes one tuple that is part of the interference relation $\rightarrow$. We abstract from the aforementioned interference between an element and itself, and therefore omit the corresponding self-loops from Fig. 3.2.

Noninterference. On the basis of the interference relation, we extend the environmental model with additional relations needed to represent a full-fledged flow policy. In particular, we add a dedicated noninterference relation (↝̸), which enables the documentation of unauthorized information flows. Systems engineers may use this relation to restrict the flow of information from particular environmental elements to others. For example, in Fig. 3.2, we make use of the noninterference relation to document the security requirements introduced in Section 1.2. First, to satisfy the confidentiality requirement, a noninterference between the Occupant and the Cloud Storage stipulates that no personal information is allowed to be stored. In particular, the user input received from an occupant must not influence the storage and retrieval of vehicle data to or from the cloud. Second, the integrity requirement described in Section 1.2 corresponds to a noninterference between Predictive Maintenance and Occupant. Hence, requesting diagnostic data must not enable maintenance engineers to influence the user output given to an occupant. In summary, the documented noninterference relation restricts the flow of information through the car and thereby act as a security policy for its internal information processing.

Neutrality. In general, the absence of an interference from one element to another states that no data is communicated between them directly. Nevertheless, they are still allowed to exchange information by communicating indirectly over other, intermediary elements. Such indirect information flows correspond to the neutral relation (↝) described in Section 2.4.1. We therefore assume that, whenever two elements are not explicitly related by an interference or noninterference, they are implicitly related by the neutral relation. According to this implicit role, we omit the neutral relation from the visual representation. As an example in the context of Fig. 3.2, vehicle data stored by the Cloud Storage may be used for analytics. It is therefore authorized to flow to the Predictive Maintenance back-end, whereas the Self-Driving Car acts as an intermediary element. Similarly, to enable crowdsourcing of traffic conditions, the car's location is authorized to flow from the Navigation Satellite to the Traffic Reporting.

In Table 3.1, we tabularize the flow policy from Fig. 3.2. Vertically, we show each of the elements as sources of a flow, whereas the elements shown horizontally represent the targets. As can be seen, the table relates each pair of source and target elements by exactly one of the flow relations →, ↝̸, or ↝. According to the reflexivity of →, each element relates to itself by means of an interference. In Table 3.1, we denote these self-relations by ↻.

Table 3.1: Flow policy of the self-driving car.

		Target					
		Self-Driving Car	**Navigation Satellite**	**Cloud Storage**	**Occupant**	**Predictive Maintenance**	**Traffic Reporting**
Source	**Self-Driving Car**	↻	↝	→	→	→	→
	Navigation Satellite	→	↻	↝	↝	↝	↝
	Cloud Storage	→	↝	↻	↝	↝	↝
	Occupant	→	↝	↝̸	↻	↝	↝
	Predictive Maintenance	→	↝	↝	↝̸	↻	↝
	Traffic Reporting	→	↝	↝	↝	↝	↻

Please note that a noninterference typically relates two environmental elements, thereby restricting the flow of information through the system under development. However, a noninterference may also be used to document explicitly that there is no exchange of information from an environmental element to the system or vice versa. For example, in Fig. 3.2, it would be possible to add a noninterference from the Self-Driving Car to the Navigation Satellite, which documents explicitly that no data is communicated from the car to the satellite. However, this fact is already known implicitly because there is no information flow between these elements in the environmental model (cf. Fig. 2.3). Besides the explicit documentation of this known fact, no further restrictions are imposed on the internal information processing of the car itself. Hence, in the scope of this thesis, such a noninterference is of minor importance because it does not restrict the information flow through the system under development.

In contrast, an interference is typically used to describe communication between the system and an environmental element. However, environmental elements might also communicate externally beyond the bounds of the system under development. Although this external communication is uncontrollable because the information is flowing *around* the system, it might still impose additional restrictions on the information flow *through* the system. Therefore, external communication may be a crucial factor that decides about the validity of a policy refinement. Hence, to detect invalid refinements in the presence of external communication, we allow an interference to connect two environmental elements as well.

Well-Formedness. In our approach, an environmental model is interpreted as a flow policy and must therefore follow the syntactic well-formedness rules described in Section 2.4.1. These rules affect the environmental model as follows:

1. Conceptually, the environmental model must represent a *complete* graph, which connects any two security domains exactly once. However, since the neutral relation is implicitly assumed whenever two elements are not explicitly related, we only need to handle conflicts between the interference and noninterference relations. Thus, two elements must not be related both by an interference and a noninterference.

2. In order to enable the implicit reflexivity of the interference relation, a noninterference must not be a self-loop. Accordingly, source and target elements of a noninterference must not be the same.

3.5 Validation of Refined Policies

When decomposing the system under development into subsystems, systems engineers need to transfer their documented confidentiality and integrity requirements to the active structure that results from the decomposition (cf. Section 2.2.2). Thereby, engineers refine the system-level security policy into a subsystem-level policy, restricting the information flow between nested system elements. In Section 3.5.1, we describe the refinement of policies, before addressing the validity of such refinements in Section 3.5.2.

3.5.1 Refinement

To enable the refinement of security policies, we adopt the specification technique introduced in Section 3.4 and apply it to the active structure as well. As described in Section 2.2.2, the active structure arises from the recursive decomposition of the system under development into nested system elements. At each step of the decomposition process, a coarse-grained flow policy must be refined by a finer-grained policy. To describe the refinement of flow policies, we make use of the following terminology: we consider a *refined* policy at a certain, higher level of decomposition, which is refined by a *refining* policy at the next lower level. Below, we address the refinement of authorized and unauthorized information flows separately.

Refining Authorized Information Flow

In general, when decomposing a higher-level element in the scope of the Consens process (cf. Section 2.2.2), systems engineers add new relations between the nested elements, thereby describing their interfaces at the lower level of decomposition. In particular, if the decomposed element is of relevance to software engineering, new information flows are added in order to describe the internal communication between its nested elements. In this context, a refining flow policy must not contradict the assumptions made by the refined flow policy. Hence, every lower-level information flow being added must necessarily correspond to a pre-existing flow at the higher level. Thus, from the higher-level perspective of the refined policy, a refining policy must not give rise to any new information flows, which were not included prior to the decomposition. Instead, only low-level flows corresponding to pre-existing flows at the higher level are allowed to be added. In order to define the above condition technically, we formalize a refinement by means of a *graph homomorphism* [HT97] between two flow policies with respect to their interference relations $\rightarrow$:

Definition 3.1 (Refinement)**.** "A flow policy $Pol_a = (D_a, \rightarrow_a, \rightsquigarrow_a, \not\rightsquigarrow_a)$ is refined by a flow policy $Pol_b = (D_b, \rightarrow_b, \rightsquigarrow_b, \not\rightsquigarrow_b)$, if and only if there is a graph homomorphism f from the graph $(D_b, \rightarrow_b)$ to the graph $(D_a, \rightarrow_a)$, i.e., there is a function $f : D_b \rightarrow D_a$ where $d_1 \rightarrow_b d_2$ implies that $f(d_1) \rightarrow_a f(d_2)$" [*Ger18].

This definition is based on a function f describing the decomposition of elements into nested elements. In particular, the function f maps every nested element to its ancestor that is being decomposed. According to Definition 3.1, every information flow $d_1 \rightarrow_b d_2$ inside the refining flow policy Pol_b must correspond to an information flow $f(d_1) \rightarrow_a f(d_2)$ inside the refined policy Pol_a.

Please note that, when an element is being decomposed, the internal communication between its nested elements is never effectively restricted by the above condition. Since every element is assumed to be self-related with respect to interference (cf. Section 3.4), an information flow between two of its nested elements will always correspond to the pre-existing flow between the decomposed element and itself. Hence, on decomposition of an element, systems engineers may freely add new internal information flows between its nested elements, without ever causing an ill-formed refinement.

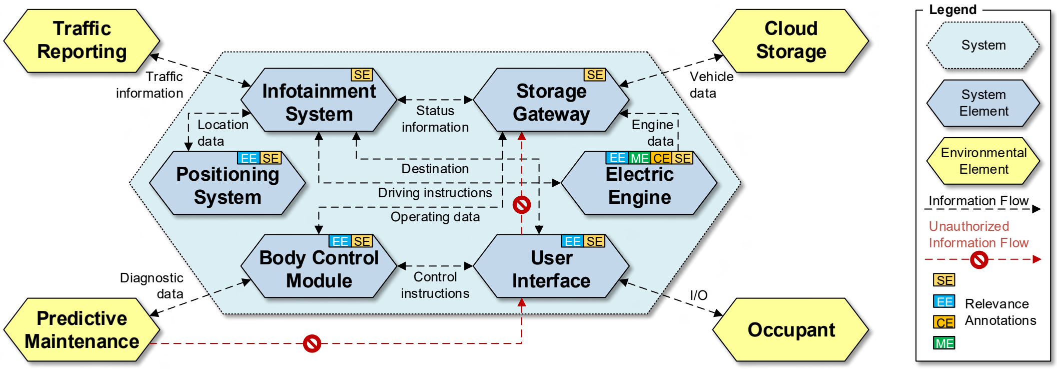


Figure 3.3: Active structure of the self-driving car extended to a flow policy.

As an example, Fig. 3.3 turns the active structure from Fig. 2.4 on page 17 into a flow policy. In the light of Definition 3.1, the policy is based on a function f mapping each nested system element to the Self-Driving Car being decomposed. Since the environmental elements are not affected by the decomposition, each of them is mapped to itself. The depicted policy is a proper refinement because only internal information flows between the nested elements are added. Each of these flows corresponds to the implicit interference of the car with itself (cf. Table 3.1). Thus, f is a graph homomorphism as required by Definition 3.1. In contrast, adding an external flow, e.g., between Traffic Reporting and Cloud Storage, would violate Definition 3.1 because no corresponding flow exists in the environmental model.

Refining Unauthorized Information Flow

In our approach, we enable systems engineers to use the noninterference relation ($\not\rightsquigarrow$) to document unauthorized information flow between the nested system elements inside the active structure. For example, in Fig. 3.3, we give a refined documentation of the car's security requirements. On the one hand, a noninterference specifies that the User Interface must not interfere with the Storage Gateway. This restriction refines the confidentiality requirement that prevents the car from leaking personal data to the cloud. On the other hand, a second noninterference states that an information flow between Predictive Maintenance and User Interface must be avoided as well. Thereby, we refine the integrity requirement that forbids maintenance engineers to interfere with the user output when requesting diagnostic data.

In general, to pinpoint the security requirements to the highest possible extent, a refined security policy should refer to elements at the bottom level of decomposition whenever possible. However, it can be seen from the above examples that this is not a mandatory restriction. In some cases, referring to a higher-level element is even necessary to ensure well-formedness of a policy (cf. Section 3.4). For example, in Fig. 3.3, introducing a noninterference between Body Control Module and User Interface would cause an ill-formed policy because both elements are already related by an interference. Therefore, the documented noninterference must necessarily refer to the Predictive Maintenance as a higher-level environmental element.

Note that even if a noninterference refers to nested system elements, it still represents a plain security requirement because it does not determine how or where the documented restriction is to be enforced. For example, although the User Interface must not leak information to the Storage Gateway, it is open which of the system elements from Fig. 3.3 are actually affected by this restriction and therefore responsible for its enforcement. Hence, in general, a flow policy at the level of the active structure does not yet impose any concrete security restrictions on particular elements. In our approach, such restrictions are derived from the documented requirements during the downstream CBSE, which we address in the upcoming Chapter 4.

3.5.2 Validity

In general, decomposing a system element introduces additional ways of communication between the nested elements. In the previous Section 3.5.1, a refinement has been defined such that none of these ways must contradict the information flows that were documented at the higher level of decomposition. However, the definition does not yet take unauthorized flows into account. Thus, to provide evidence that a refining flow policy actually enforces the information flow restrictions of the refined policy, we enable systems engineers to reason about the *validity* of refinements. To represent a valid refinement, the refining policy must enforce each noninterference documented by the refined policy, thereby blocking any ways of communication that would otherwise enable an unauthorized flow of information.

For example, the refining flow policy depicted in Fig. 3.3 must ensure that the internal communication between the nested system elements does not give rise to an information flow from the Occupant to the Cloud Storage, which is unauthorized according to the refined flow policy from Fig. 3.2. One possible way of communication that would enable such a flow crosses the User Interface, the Body Control Module, and the Storage Gateway. Hence, it is essential for the refining policy to block the flow of information across this way.

In the following, we introduce the notion of *illegalization* to ensure that critical ways of communication are properly blocked. For example, the noninterference between User Interface and Storage Gateway acts as an illegalization that rules out a critical information flow from the Occupant to the Cloud Storage. In general, to represent a proper illegalization, a noninterference must connect two arbitrary elements on a particular way that needs to be illegalized. Accordingly, we define an illegalization as follows:

Definition 3.2 (Illegalization)**.** “A flow policy $(D, \rightarrow, \rightsquigarrow, \not\rightsquigarrow)$ illegalizes a flow from $d_1 \in D$ to $d_n \in D$, if and only if for every path $(d_1 \rightarrow \ldots \rightarrow d_n) \in \rightarrow^{\star}$ there is a noninterference $d_i \not\rightsquigarrow d_j$ where $1 \leq i < j \leq n$” [*Ger18].

We rely on the notion of illegalization to define the validity of refinements. A valid refinement requires the refining policy to enforce every noninterference documented by the refined policy. To enforce a noninterference, *any* possible way of communication between the respective elements must be blocked. In Fig. 3.3, this is the case for the information flow between Occupant and Cloud Storage. All the possible ways of communication between these elements cross the User Interface and the Storage Gateway, which are restricted by means of a noninterference. Accordingly, the information flow is properly illegalized.

In general, a decomposition may also affect an element that acts as source or target of a noninterference inside the refined policy. In this case, the resulting nested elements give rise to numerous start and end points for information flows, which all require an illegalization by the refining policy. To account for such cases, we define the validity of a refinement in a general sense. To that end, we refer to the graph homomorphism f from Definition 3.1, which describes the correspondence between a decomposed element and its nested elements:

Definition 3.3 (Validity)**.** A refinement of a flow policy $Pol_a = (D_a, \rightarrow_a, \rightsquigarrow_a, \not\rightsquigarrow_a)$ by a flow policy $Pol_b = (D_b, \rightarrow_b, \rightsquigarrow_b, \not\rightsquigarrow_b)$ is valid with respect to the graph homomorphism f, "if and only if for every noninterference $d_s \not\rightsquigarrow_a d_t$, a flow from each $d'_s \in f^{-1}(d_s)$ to each $d'_t \in f^{-1}(d_t)$ is illegalized by Pol_b" [*Ger18].

In the above Definition 3.3, the refined policy Pol_a requires that any information flow between two noninterfering elements $d_s \in D_a$ and $d_t \in D_a$ is illegalized by the refining policy Pol_b. Hence, we take into account the set of nested elements of d_s and d_t, which are given by $f^{-1}(d_s) \subseteq D_b$ and $f^{-1}(d_t) \subseteq D_b$. Since the environmental elements have not been decomposed in our example, it holds that $f^{-1}(d_{env}) = \{d_{env}\}$ for each environmental element d_{env} depicted in Fig. 3.3. In contrast, the set $f^{-1}(d_{sud})$ of the system under development d_{sud} includes each of the nested system elements. In general, according to the above definition, the refined policy must illegalize a flow of information from each $d'_s \in f^{-1}(d_s)$ to each $d'_t \in f^{-1}(d_t)$ in order to represent a valid refinement.

3.6 Quality Assessment

In this section, we refer to the framework by Uzunov et al. [UFF18] in order to assess the quality of our approach as a security methodology. To this end, we discuss relevant quality criteria by constructing a *situational criteria profile* in Section 3.6.1. Next, in Section 3.6.2, we describe both our process and the involved artifacts by creating a *methodology model*. Finally, in Section 3.6.3, we present the assessment results obtained by matching the methodology model against the situational criteria profile.

3.6.1 Situational Criteria Profile

For the purpose of evaluation, we refer back to the quality factors QF1–QF3 from Section 3.2 and asses the quality of our approach with respect to these factors. To this end, the quality framework provides a *base profile* of dedicated criteria, which are associated with the individual quality factors. From this base profile, we select those criteria that are associated with our three quality factors. However, the framework allows the set of criteria to be adapted to the specific situation of the assessed methodology, leading to a *situational profile*. We make use of this adaptation because certain criteria do not apply to the context of our approach. The reason is that, due to the discipline-spanning nature of systems engineering, certain development activities are deliberately deferred to the downstream, discipline-specific engineering. Therefore, such activities are irrelevant to the quality of our approach. Accordingly, any criteria that refer explicitly to these activities are omitted from our situational profile.

In particular, with respect to assurance (QF3), we omit the quality criterion *Software support (verified implementations)*, which aims to answer the question whether the implementation and deployment of a system are supported by dedicated, verified software solutions. In our scope, both implementation and deployment are discipline-specific activities that are no essential parts of our approach at the level of systems engineering. Furthermore, in terms of completeness (QF2), we omit the quality criterion *Specific artifacts (basic security solutions)*, which requires a methodology to integrate explicit artifacts for countermeasures like access control or encryption. Since such countermeasures are specific to the discipline of software engineering, their adoption is regarded as a discipline-specific development activity as well. Finally, from the assessment of the completeness, we also omit the criterion *Documentation (security process)* because documenting the process is the subject of this thesis. The remaining portion of the base profile is reused by our situational profile as is. We refer the reader to the upcoming Section 3.6.3 for an overview of the assessed criteria, and to [UFF18] for a detailed description of the indicators and resulting measures used to assess a certain criterion.

3.6.2 Methodology Model

To prepare the quality assessment, we construct a *methodology model* of our engineering process and the conceptual artifacts involved. In Fig. 3.4, we use the notation by Uzunov et al. [UFF15] to illustrate the *process model* on the left and the *framework model* of the involved artifacts on the right. In the following, we describe both models separately.

Process Model

Uzunov et al. introduce a set of *process patterns* to describe recurring, security-related activities that engineers engage in during the development of systems. In the following, we check our proposed process against the set of process patterns and arrange the recognized patterns as part of the process model depicted in Fig. 3.4. The authors distinguish between patterns at different levels of granularity, separating (i) coarse-grained *phase patterns*, which relate to general development phases, (ii) *stage patterns* referring to particular stages within a certain phase, and (iii) fine-grained *task patterns*, which correspond to individual development activities at a certain stage. In particular, we recognized the following patterns in our approach:

Phase patterns: Due to our focus on information flow requirements, the only matching phase pattern is the *security requirements determination* (SecReq). However, we apply SecReq at different steps of the development life-cycle: during *requirements analysis*, engineers document security requirements by turning an environmental model into a flow policy, whereas the documented policy is refined during *design* at the level of the active structure. To distinguish these different steps, Uzunov et al. introduce a set of *generic life-cycle modifiers* [UFF15]. In Fig. 3.4, we use the ReqAn and Des modifiers to denote the initial requirements analysis and the subsequent design. Other existing phase patterns besides SecReq either integrate specific security countermeasures, or focus on their implementation or administration. Since such activities are specific to the software engineering, they are beyond the scope of our discipline-spanning approach.

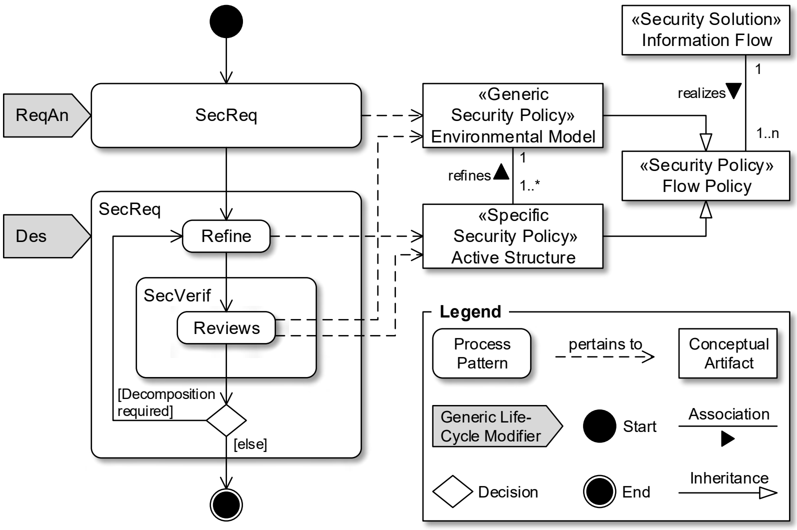


Figure 3.4: Methodology model of the process (left) and the conceptual artifacts (right).

Stage patterns: By validating the refinement of security policies, our process matches the *security verification* (SecVerif) stage pattern, which is used to “verify the correctness, consistency and completeness” [UFF15, p. 226] of security-related artifacts.[1] As shown in Fig. 3.4, we apply this pattern as part of the SecReq phase during design.

Task patterns: At the level of individual tasks, our process matches the *refinement* (Refine) pattern because we consider security policies at different levels of granularity, refining coarse-grained to finer-grained policies. As depicted in Fig. 3.4, the Refine pattern is applied during design, prior to the SecVerif stage. Subsequently, inside SecVerif, the *manual reviews* (Reviews) pattern matches as well because we provide engineers with a validity check carried out manually. Both Refine and Reviews are executed in a loop-like manner as long as further decomposition of the system is required (cf. Fig. 3.4).

Framework Model

With respect to conceptual artifacts, the framework model in Fig. 3.4 shows that our approach is based on a Flow Policy as a specific kind of Security Policy. Uzunov et al. define a security policy as “the solution space form of a security requirement, which is realized by a security solution of some form” [UFF15]. In our work, the realizing security solution is the theory of Information Flow (cf. Fig. 3.4). Dashed arrows indicate that both requirements analysis and design lead to an individual Security Policy. The Environmental Model acts as a Generic Security Policy, which is refined by any Specific Security Policy given in terms of the Active Structure.

[1] Boehm [Boe81] distinguishes the *validation* of requirements (“building the right product”) from their *verification* (“building the product right”). Although our approach provides validation only, it still qualifies for the *security verification* pattern by verifying that policies “provide the necessary level of security” [UFF15].

3.6.3 Results

In this section, we present the assessment results obtained by measuring the criteria from the situational profile (created in Section 3.6.1) on the basis of the methodology model (constructed in Section 3.6.2). In Table 3.2, we summarize the quality assessment by tabulating the individual quality criteria, their corresponding quality factors, and the resulting quality measures. For the presentation of measures, we apply the color scheme used by Uzunov et al. [UFF18]. In the following, we discuss the measure for each criterion in detail.

Specific Activity Sequence: This criterion investigates whether process patterns "are placed in a sensible order in a given methodology" [UFF18]. As indicators for this criterion, Uzunov et al. provide a set of invalid pattern orderings. Comparing the order of process patterns from Fig. 3.4 against these invalid orderings, no inconsistencies were detected. Therefore, we asses our process as *well ordered*.

Compatibility: The compatibility criterion excludes inappropriate combinations among process patterns, or incompatibilities between a process pattern and a generic life-cycle modifier [UFF18]. Checking our process model against the indicators provided by Uzunov et al., we detected that the embedding of a Reviews task inside a SecReq phase is assessed as inappropriate. In general, the quality framework does not provide for any verification tasks during SecReq at all, thereby contradicting the statement that manual reviews "can be done at all development stages and as part of verification activities in all phase [...] patterns." [UFF15, p. 227]. Nevertheless, according to the given indicators, we conclude that there are *incompatibilities present* in our process.

Broadness of Scope: This criterion refers to the encompassed development phases of a methodology. Our process model from Fig. 3.4 uses two generic life-cycle modifiers (ReqAn and Des). Thus, the resulting measure for the broadness of our approach is *multi-phase*. However, please note that any subsequent modifiers correspond to the implementation or deployment of systems, which are discipline-specific activities that have been excluded deliberately from our system engineering approach.

Coverage: This criterion investigates whether the process model covers each encompassed development phase with security-related activities. Our process model provides ReqAn and Des with activities for the documentation and validation of security requirements. Thus, our approach covers *most* phases, which is the optimal measure [UFF18].

Early Security Introduction: This criterion requires "an attempt to consider security requirements as early as possible in the development process" [UFF18]. Since the SecReq pattern is part of our process model and applies to the earliest possible generic life-cycle modifier (ReqAn), the resulting measure for this criterion is *yes*.

Threat Assessment: Our approach does not consider security threats as an explicit part of the engineering process. Thus, *threat modeling* [XL19] and *threat analysis* [TÇS18] are promising future extensions to help identify a set of threats that need to be mitigated by a security policy. Due to that shortcoming, the measure for this criterion is *no*.

Table 3.2: Assessment results for the situational criteria profile.

Quality Factor	Quality Criterion	Quality Measure
Correctness of Construction	Specific activity sequence (valid pattern ordering)	well ordered
Correctness of Construction	Compatibility (process patterns)	incompatibilities present
Completeness	Broadness of scope (encompassed development phases)	multi-phase
Completeness	Coverage (related development activities)	most {policy documentation and validation}
Completeness	Specific activities (early security introduction)	yes
Completeness	Specific activities (threat assessment)	no
Completeness, Assurance	Specific activities (verification)	manual
Assurance	Specific activity approach (formal verification)	formal {information flow}
Assurance	Specific activities (security measurement)	no
Assurance	Reusability (generated security designs)	high {generic policy indicates validity of specific policies}

Verification: Validating refinements is one of our primary objectives, as indicated by the SecVerif pattern in our process model. However, according to the Reviews task pattern, no automation has been implemented yet. Although our formalization provides the basis for an automated validation, we assess the current approach as *manual*. Note that, by focusing on validation, we do not address verification in the proper sense [Boe81]. Instead, verifying that security requirements are satisfied is the subject of Chapter 5.

Formal Verification: By integrating flow policies, our work applies formal methods from the theory of information flow security. Since the SecVerif stage in our process model is based on the formal rigor of this approach, we assess the verification as *formal*.

Security Measurement: No explicit measurement activities have been used to quantify the level of security enforced by a specified policy. Hence, in particular, systems engineers need to trade off different valid refinements against each other manually. In the absence of a quantified level of security, the resulting measure for our approach is *no*.

Reusability: This criterion demands that artifacts "linked in earlier phases are also used in later phases" [UFF18]. As can be seen from the Reviews task in Fig. 3.4, the Generic Security Policy serves as an indicator for the validity of a Specific Security Policy, thereby reusing artifacts across ReqAn and Des. Accordingly, we assess the reusability as *high*.

In the following, we discuss the significance of the above assessment results with respect to the quality factors given in Section 3.2, starting with the *correctness of construction* (QF1). In this context, the defect identified when assessing the *compatibility* criterion is caused by an essential feature of our approach, which has been added deliberately in order to enable requirements validation. Apart from this, no further deficiencies were identified, which is why we conclude that our approach has been constructed correctly.

With respect to *completeness* (QF2), our approach benefits from its *early security introduction* and the *coverage* of most development activities. The limited *broadness of scope* is due to the fact that our systems engineering approach deliberately omits any discipline-specific activities for the implementation or deployment. Hence, we conclude that this criterion does not compromise the completeness of our approach. In contrast, the missing *threat assessment* clearly indicates that our work could be improved by integrating an upstream threat analysis, which would enable security requirements not only to be documented, but also to be elicited in a systematic and thorough way. Moreover, in the present form, our approach is also limited by the manual validation that was revealed by the *verification* criterion. However, due to the formalization given in Section 3.5.2, the validation is conceptually prepared for an automated, rigorously defined check. Nevertheless, we conclude that our works leave room for additional improvement with respect to completeness.

In terms of *assurance* (QF3), our approach meets the *formal verification* criterion and also benefits from a high *reusability* of security policies. In contrast, the missing *security measurement* indicates that engineers would benefit from additional support when deciding which refinement is most precise, i.e., enforces the required level of security without being overly restrictive. The assurance is also affected by the lack of automated *verification*. Whereas both shortcomings give rise to promising future extensions, we conclude that they do not affect the intrinsic assurance given by the formal rigor of our work.

3.7 Limitations

Security Threats. As suggested by the assessment in Section 3.6, our approach is limited by its missing consideration of security threats. This view is shared by Türpe [Tür17], who points out that *threats* are an indispensable dimension besides the stakeholders' *goals* and the system *design*, such that the interplay of all three dimensions must be taken into account during security requirements engineering. Likewise, Bastys et al. [BPS18] propose *attacker-driven security* as a design principle for information flow control, according to which the threats posed by specific attackers must be modeled explicitly by means of *threat modeling* [XL19]. Consequently, we propose to enhance our work with an upstream, discipline-spanning *threat analysis*, which enables systems engineers to "identify, analyze and prioritize potential security [...] threats to a software system and the information it handles" [TÇS18, p. 275]. Currently, our work supports the specification of security policies, but does not help identify threats that a policy must address. Due to the discipline-spanning nature of MBSE, the challenge for a threat analysis is that any discipline-specific details are deliberately omitted from the models in use. Therefore, such information cannot be used to identify threats.

Side Channels. Another current limitation is that the documentation and validation of security policies only takes into account the data communication, which is represented by information flows. Thus, our view is currently restricted to the *cyber* layer of a CPS. Thereby, we leave out of consideration that critical information could also be transferred by physical effects such as noise, heat radiation, or energy consumption. In the field of cryptography, a physical transfer of information beyond the designated communication channels is commonly known as a *side channel* [Cad11]. As described in Section 2.2, physical interactions are (ideally) represented in terms of energy or material flows at the level of Consens. Hence, the logical consequence is to incorporate such flows into the definition of flow policies as well, and take them into account when checking the validity of refined security requirements. Accordingly, by extending our view to the *physical* layer of CPSs, we could promote the early identification of potential side channels.

Declassification. Furthermore, a limitation of our work is that the specified policies are ultimate in the sense that they restrict the information flow unconditionally. It is therefore impossible to specify that a flow is restricted only under certain conditions. This inflexibility affects the applicability of our approach to real-world scenarios, where certain flows of critical information are often indispensable to meet a system's functional requirements. Introducing well-defined exceptions to a security policy, and thereby downgrading the sensitivity of otherwise classified information, is also referred to as *declassification* [SS09]. For example, in the scope of Consens, an obvious extension is to use the specified system behavior (cf. Section 2.2.2) in order to authorize certain flows under specific behavioral conditions, e.g., when the system is in a particular state. Such an exception describes *when* a critical information flow is authorized, which is one of four crucial dimensions of declassification [SS09]. However, in the context of our work, such an extension poses the challenge that the specified exceptions must also be taken into account when checking the validity of refinements. Any two flow policies involved in a refinement would need to be consistent with respect to their specified exceptions. In particular, a refining policy must keep the conditions specified by the refined policy without adding any inconsistent exceptions.

Requirements Verification. In addition, our approach is also limited in the sense that we do not check the aforementioned system behavior against the specified flow policies. Hence, no requirements verification takes place to find out whether the behavioral scenarios adhere to a specified flow policy, or give rise to dependencies between critical and observable information. In general, reasoning about information flow security on the basis of possible execution traces is feasible [Man00a; McL96]. However, in order to make a clear statement about the presence or absence of unauthorized information flows, systems engineers would need to specify the behavior of a system exhaustively because secure information flow is a *hyperproperty* (cf. Section 2.4.2). As such, it requires "sufficiently many possible traces" [Man02, p. 91] that are indistinguishable with respect to observable information, regardless of which critical information is processed on an individual trace. Therefore, engineers would need to anticipate large parts of the system behavior upfront. This is a challenge because usually only single execution traces are represented in the form of basic scenarios. Therefore, the behavior specified at the level of Consens is likely to be incomplete and therefore does not enable information flow requirements to be verified reliably.

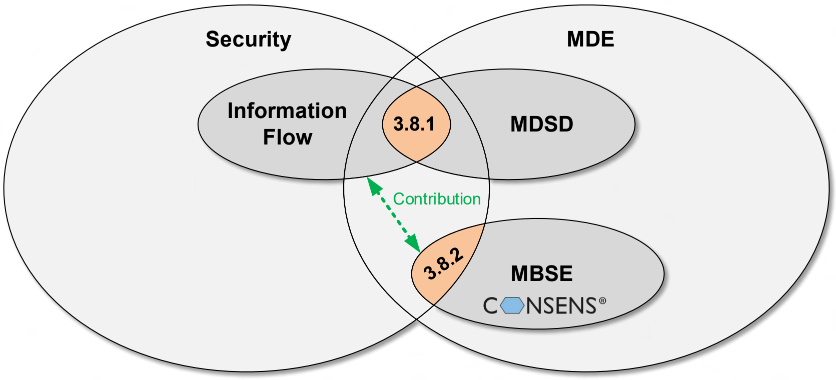


Figure 3.5: Overview of related works from different areas.

3.8 Related Work

According to our integration of MBSE and information flow security, we consider related work from two areas (cf. Fig. 3.5). First, in Section 3.8.1, we address the intersection of information flow and the more general field of MDE. Conversely, Section 3.8.2 covers approaches towards MBSE that integrate security in a broader sense. In general, we lay the focus on related works for security requirements engineering.

3.8.1 Information Flow in Model-Driven Security Requirements Engineering

FlowUML [AFW06] enables an early validation of information flow policies during requirements engineering, similar to our work. To this end, the authors extract flows from UML sequence diagrams and check these flows against logic-based security policies. Similarly, Hoisl et al. [HSS14] address the security of object flows in business process models given in terms of UML activity diagrams. Encryption is used as a security solution during the downstream development in order to enforce the confidentiality and integrity of such flows. Thereby, unlike our work, the approach is not based on information flow in a narrower sense.

RIFL [Bau+17] is a requirements specification language for secure information flow. As such, flow policies relating different security domains can be specified natively by the approach. As a tool-independent specification language for information flow requirements, RIFL is also applicable to specific domains such as CPSs. However, compared to our work, RIFL is a plain requirements specification language, which does not address the refinement of the specified policies as our major goal in this chapter.

In contrast, *STAIRS* [SSS09] is an approach with an explicit notion of refinement. On the basis of a system specification that corresponds to a set of UML sequence diagrams, the approach aims to preserve the information flow security whenever the system specification is refined. However, please note that this notion of refinement differs profoundly from our work because it corresponds to a refinement of the specified system behavior. In contrast, our approach refines the security policies that act as requirements for this behavior.

A well-known approach for model-driven security engineering on the basis of UML design models is *UMLsec* [Jür05]. Like our work, the approach provides a design notation that is augmented with concepts for secure information flow. Moreover, dedicated extensions have been proposed to connect the design approach of UMLsec with security requirements engineering [HHJS11; Hou+10; MJ10]. Therefore, similar to our work, the approach supports the transition between security requirements and secure designs. However, unlike our work, a systematic refinement of security policies is not addressed.

IFlow [KSBR15] is another approach that integrates information flow security into UML-based software designs. For the specification of security requirements, the authors make use of policies that conceptually resemble the flow policies underlying our work. On this basis, they enable formal verification of the system behavior with respect to the specified policies, and the generation of code skeletons that are amenable to static analysis.

In summary, numerous related works combine information flow requirements and MDE. However, most of them employ models from the UML and are therefore tailored to be used by software engineers. Thus, as shown in Fig. 3.5, the intersection of Information Flow and MDE includes merely approaches for MDSD . In contrast, none of these works are tailored to systems engineering, which was the objective of our work in this chapter.

3.8.2 Security in Model-Based Systems Engineering

In the past, multiple technology-independent methodologies for secure systems engineering have been proposed. For example, *Tropos* [MGM03] promotes the model-based engineering of security requirements throughout all phases of the systems engineering process, leading to the *Secure Tropos* extension [MG07]. Similar to our work, the methodology applies incremental refinements to the system models in use. As a general methodology, Tropos is also applicable to the engineering of information flow requirements. However, the approach does not explicitly integrate formal methods from the field of information flow. Another general methodology for secure systems engineering is *ISSEP* [RMR15]. The authors separate the responsibilities of conventional systems engineers from those of security experts, who provide the engineers with libraries of reusable security solutions to satisfy the specified requirements. However, as distinct our work, these security solutions do not comprise any formal methods from the field of information flow security.

Vasilevskaya et al. [VGNH14] present another security-aware development process for networked embedded systems, referred to as *SEED* [VN16]. Similar to ISSEP, the approach aims to provide systems engineers with the required security knowledge, which is shared by security experts. In particular, the approach has been used to quantify the level of risk that a system is exposed to [VN15]. Another approach towards risk analysis has been proposed by Grunske and Joyce [GJ08]. By analyzing SysML-based component architectures, the authors explicitly address the discipline-spanning systems engineering. The work by Ouchani et al. [OMD13] also enables risk assessment on the basis of SysML, however, their analysis takes into account the behavior of a system given in terms of sequence diagrams. In Section 3.6, such a quantitative security analysis has already been identified as a shortcoming of our constructive approach.

Besides these risk-centric approaches, other works have also integrated security into SysML. For example, Oates et al. [OTH13] address the security of industrial control systems by extending SysML with concepts for the representation of threats, vulnerabilities, assets, and security solutions like encryption. Whereas the approach takes security threats into account, there is no specific support for security requirements engineering. The *SoSSec* method [Hac+17] provides an architecture description language [MT00] for *systems of systems*, which are defined as "systems that are composed of independent constituent systems" [Nie+15]. The authors enable known vulnerabilities and appropriate countermeasures to be specified in SysML, and propose a simulative analysis to detect potential exploits. Similarly, Lemaire et al. [LVJN17] analyze vulnerabilities in SysML models of industrial control systems. The analysis is conducted by a knowledge-based system with rules taken from dedicated databases or derived from domain-specific guidelines. By focusing on vulnerabilities, both works do not explicitly account for the security requirements engineering.

Ouchani and Lenzini [OL15] enable the detection of attack surfaces, using SysML activity diagrams to describe both the behavior of systems and known attack patterns. Thereby, they promote the validation of security requirements. Belloir et al. [Bel+14] present an elicitation process for security requirements of systems of systems on the basis of SysML. They support the translation of requirements into design models, similar to the intent of this chapter. Another prominent methodology for secure systems engineering is *SysML-Sec* [AR16]. Since the approach covers the entire development process, it also enables systems engineers to account for predefined security requirements at the design level. To that end, SysML-Sec enables software components to be refined systematically, which is a commonality shared with our work. Nevertheless, none of the above approaches based on SysML involve information flow and corresponding security policies.

Lemaire et al. [LVDN17] analyze data flows within SysML models of CPSs. The analysis detects whether stakeholders may access specific data assets and thereby violate predefined confidentiality policies. By specifying such policies, their work is similar to our approach. However, the authors design a system in terms of its constituent hardware components, whereas our approach aims at the enforcement of security policies by software components.

In summary, the multitude of the above approaches suggests that security has evolved into a crucial factor for MBSE. Nevertheless, none of the previous works provide systems engineers with formal methods from the area of information flow security, as indicated by the empty intersection between Information Flow and MBSE in Fig. 3.5. Hence, connecting these isolated fields is the novel contribution made in this chapter.

3.9 Summary

We presented a novel integration of formal methods from the theory of information flow into the Consens technique used in systems engineering practice. Thereby, we enabled the documentation of unauthorized information flows in terms of flow policies at an early stage of MBSE. On this basis, our contributions help refine these policies during the system design and validate that the intended restrictions are still enforced after each refinement.

To assess our contributions, we referred to the quality framework for security methodologies by Uzunov et al. [UFF18]. According to this framework, our approach promotes the assurance of information flow restrictions by supporting the early requirements validation with formal methods. In contrast, to improve the completeness of our work, a promising future extension is the integration of an upstream threat analysis.

Engineers benefit from our approach by documenting information flow requirements at an early, discipline-spanning stage, without deferring security needs to the downstream software engineering. On this basis, our proposed validity check enables systems engineers to refine the documented requirements consistently from the system-level to the subsystem-level. The refined requirements lay the foundation for the derivation of security policies for individual software components during the downstream software engineering.

4

Architectural Refinement of Component-Based Security Policies

As demonstrated by the MechatronicUML component model described in Section 2.3.2, component-based principles are often applied to the architectural design of CPSs [CMMS16]. Accordingly, systems are decomposed into numerous software components, whereas the global system behavior emerges from the local behavior of the constituent components. Recently, the principle of CBSE has gained in importance due to the emergence of the *microservice* architectural style [FLM19; Jam+18; Dra+17], which promotes a fine-grained decomposition into small-size components that can be deployed and scaled independently.

When reasoning about the quality of component-based systems, a key feature is *composability* [CSVC11]. According to this feature, local quality properties of constituent components act as indicators for the global quality of a composite system. In the context of security, composability enables software architects to check single components against localized security policies and thereby reason about the global security of the composite system. We refer to this approach as *local reasoning* [ORY01]. Whereas reasoning about extra-functional[1] properties like security is a well-established line of research [SSCC09; Zsc10; MRRW16], their composability is regarded as “the most difficult challenge in CBSE” [CSVC11].

Local reasoning about information flow properties is particularly challenging. Since such properties are not generally composable (cf. Section 2.4.2), a composite system may be compromised by global information flows, even if all components are locally secure [Man02]. This problem gets even more difficult if the components communicate with each other in both directions, which is a form of composition that is known as *feedback* [McL96].

In the presence of feedback composition, the composability of information flow properties is a challenging problem [ZL96], which we address in the context of MechatronicUML. According to the top-down decomposition described in Section 2.3.2, the coarse-grained security policy of a composite component must be enforced by finer-grained policies of its subcomponents, thereby requiring security policies to be *refined* [LS11]. To ensure security of composite components, the refined policies of the subcomponents need to be composable. Hence, during the refinement, software architects must be supported by appropriate guidelines that guarantee composability and thereby enable local reasoning about security.

[1] Following Crnković et al. [CSVC11], we use the terms *extra-functional* and *non-functional* interchangeably.

State of Research. The composability of information flow is a long-established research topic, which has been approached from a theoretical viewpoint by McCullough [McC88], Zakinthinos and Lee [ZL95], McLean [McL96], as well as Mantel [Man02]. On the basis of these fundamentals, recent approaches towards composability even address CPS characteristics like message passing [LMT17] or time-sensitive behavior [RJB17]. However, instead of taking architectural components into account, these works are based on lower-level program specifications, thereby neglecting the refinement of security policies during the architectural decomposition. In contrast, whereas the scientific literature also covers approaches towards security for component-based CPSs [MA11; BPKN18; SLCS12], these works do not provide architects with the formal rigor of information flow. Finally, existing approaches that apply information flow to a component-based design do not address specific characteristics of CPSs. For example, previous works do not take into account the real-time behavior [SARL13; CM15; GMB17] and therefore fail to detect timing channels (cf. Section 2.4). Likewise, existing approaches assume a synchronous handshake communication [GMB17; SABB14], which contradicts the asynchronous message passing of CPSs that underlies our work.

Contributions. To enable local reasoning about security in MechatronicUML, this chapter makes the following contributions. Initially, we introduce a novel notion of component-based security policies, in which the ports of a component are classified according to the security sensitivity of their exchanged information. These sensitivities specify unauthorized dependencies between ports and thereby restrict the information flow through a component. On this basis, we augment the transition from Consens to MechatronicUML with a semi-automatic translation that partially derives such component-based policies from the flow policies used in Chapter 3. Thereby, we enable a seamless consideration of information flow restrictions across MBSE and CBSE. Next, our major contribution is a set of architectural *well-formedness* rules for the refinement of component-based security policies. By establishing these rules, we draw up design guidelines that stipulate how the policies of subcomponents must be shaped to be composable and thereby enforce the policy of their composite component. Following these rules during refinement enables software architects to reason locally about security, preventing the emergence of global information flows regardless of whether components are assembled with or without feedback. The proposed rule set is based on the theory of compositional information flow by Mantel [Man02] and applies these theoretical principles to the field of CBSE. Finally, we demonstrate the soundness of our approach by showing that our rules guarantee composability in the context of the MechatronicUML component model.

Novelty. The key novelty of our contribution lies in the synergy between CBSE and formal methods [Lau17], which enables the design of secure software architectures for CPSs. Thereby, we are the first in the field of architectural security to combine crucial CPS characteristics on the one hand, and the formal rigor of information flow security on the other hand. By formally underpinning our approach, we assure software architects that their policies are refined in a well-formed way. Thereby, we enforce the intended security restrictions without providing an opportunity for unauthorized information flows to emerge on composition. Moreover, another novelty is the transition from flow policies used in MBSE to component-based policies in the context of CBSE, which enables software architects to account for the security requirements specified during the upstream systems engineering.

Publication. The well-formed refinement of component-based security policies, as the main contribution of this chapter, was initially proposed in a conference paper [*GS18], before being elaborated and evaluated in the context of MechatronicUML by another conference paper [*GS19]. Furthermore, the transition of security policies from MBSE to CBSE has been sketched in a workshop paper [*Ger18].

Outline. The remainder of this chapter is structured as follows. We condense our scientific contributions in Section 4.1, before specifying the requirements for our approach in Section 4.2. We give an overview on our work in Section 4.3 and introduce our notion of component-based security policies in Section 4.4. Section 4.5 describes the transition from flow policies to component-based policies, whereas we establish well-formedness rules for these policies in Section 4.6. We provide evidence for the composability of our approach in Section 4.7 and discuss known limitations in Section 4.8. Finally, we survey related work in Section 4.9, before summarizing this chapter in Section 4.10.

4.1 Scientific Contributions

In summary, this chapter makes the following contributions:

- We augment the MechatronicUML component model with a notion of component-based security policies, which are used by software architects to restrict the information flow of individual software components.
- We enhance the integration of Consens and MechatronicUML with a translation of flow policies into component-based security policies.
- We establish a set of architectural well-formedness rules for component-based security policies. These rules act as a guide to software architects when refining policies during the architectural decomposition.
- We provide evidence for the composability of component-based security policies. Thereby, we assure software architects that the policies of subcomponents are guaranteed to enforce the information flow restrictions of a composite component.

4.2 Requirements

Crnković et al. [CSVC11] identify three characteristic features of component models, according to which the handling of extra-functional properties can be classified. These features are the specification, management, and composability of properties, which will be used as requirements (R1–R3) for the handling of security in our approach:

(R1) Specification: The specification of extra-functional properties is the “most basic support that a component model can provide” [CSVC11]. Accordingly, our approach must enable software architects to impose security restrictions at the level of components as building blocks of a software architecture.

(R2) Management: This advanced feature describes how the conformance to a specified property is managed by a component model. Instead of containerizing components to manage their security from the outside, we delegate this responsibility to the inside of each individual component, such that their implementations must conform to the imposed security restrictions. Thereby, we adopt an *endogenous* approach to security management [CSVC11]. Furthermore, we also assume that security is managed directly by the collaboration of components, instead of utilizing security services provided by the underlying computing platform to which components are deployed. Our approach therefore corresponds to a security management *per collaboration* [CSVC11]. By combining these two characteristics, we do not directly require our component model to support the security management.

(R3) Composability: The most challenging feature of component models is their ability to compose extra-functional properties of individual components [CSVC11]. Thus, in our context, the local security restrictions of components must be composable and thereby enable reasoning about the global security of overall software architectures.

4.3 Overview

To meet the requirements specified in Section 4.2, we extend MechatronicUML's architectural design process from Fig. 2.5 on page 19. In Fig. 4.1, we highlight our extensions in green. Initially, in response to the Derive Component Architecture activity, software architects derive a set of partial *component security policies*, restricting the information flow at the level of software components. We introduce these policies in the upcoming Section 4.4. Each policy is derived from one or more non-unauthorized information flows inside the flow policy documented by systems engineers at the level of the active structure (cf. Chapter 3). On the basis of the prior derivation of the component architecture from the active structure, the Derive Security Policies activity is an automatable procedure, which we describe in the upcoming Section 4.5. Since component security policies can only be partially derived from a flow policy, they need to be manually completed by software architects in the subsequent Refine Security Policies activity.

Next, the completed component security policies are checked during the automatable Check Well-Formedness activity. In the upcoming Section 4.6, we establish the well-formedness rules underlying this check. If the refined policies are not well-formed, we require software architects to revise the refinement by repeating the Refine Security Policies activity and to recheck the policies during the Check Well-Formedness activity. Once all policies are well-formed, the Decompose Component activity may lead to a recursive decomposition of the architecture as described in Section 2.3.1. In response to each decomposition, the security policies of the decomposed composite component need to be refined by corresponding policies for the resulting subcomponents. To this end, software architects repeat the Refine Security Policies and Check Well-Formedness activities. Finally, the architectural design process terminates if all security policies have been refined in a well-formed way and no further decomposition of the component architecture is required.

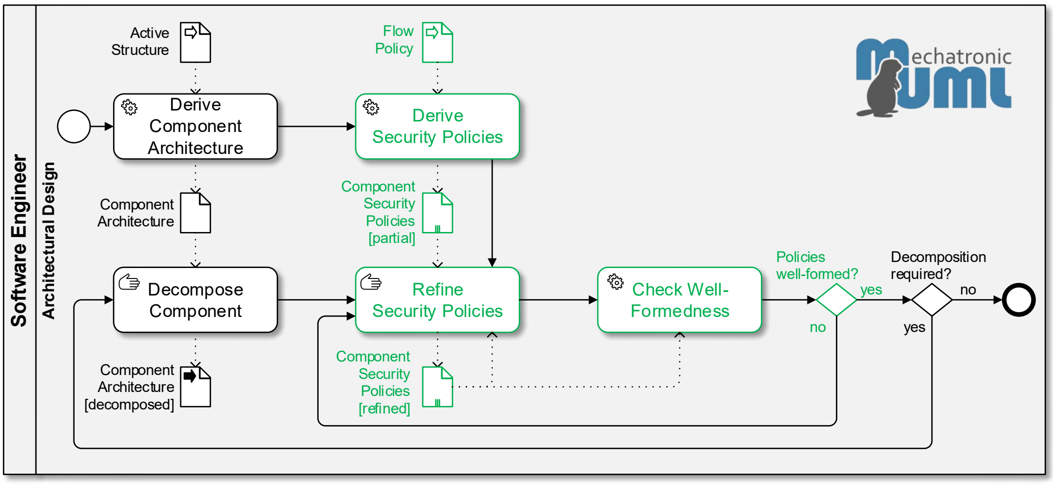


Figure 4.1: Proposed extensions to the MechatronicUML architectural design process.

4.4 Component-Based Security Policies

To enable the specification of security restrictions (R1 from Section 4.2), security policies must distinguish authorized from unauthorized flows of information. Thus, on the one hand, a policy must define which information is security-critical and may therefore act as the source of an unauthorized information flow. On the other hand, a policy must define what information is publicly accessible and may therefore represent the sink of an unauthorized flow. In the scope of CBSE, we enable software architects to specify security policies on a per-component basis. Since ports reflect the information exchange of a component, we consider them as sources or sinks of an information flow. As a principle proposed by Bastys et al. [BPS18], we thereby separate policies from the implementations of components. To that end, we classify ports according to the sensitivity of the exchanged information. On the basis of the sensitivity levels from Section 2.4.1, this leads to the following classification scheme, which also introduces the visual encoding of sensitivities that we use during the remainder of this thesis:

- *Critical* ports are sources of unauthorized information flows through a component. Hence, in order to avoid public disclosure, information received over such a port must not become publicly known.

- *Observable* ports are sinks of unauthorized information flows through a component. Hence, information exchanged over such a port is publicly accessible and, therefore, must not enable its observers to draw any conclusions about critical information.

- *Neutral* ports are neither sources nor sinks and therefore used to authorize information flows through a component. Unlike critical ports, information received over a neutral port may become publicly known. Unlike observable ports, information exchanged over a neutral port allows conclusions to be drawn about critical information.

Since we restrict ourselves to the coordination behavior of MechatronicUML, our policies involve only discrete ports (cf. Section 2.3.2). Hereafter, we define the security restrictions imposed by the above sensitivities more precisely. As described in Section 2.4.2, definitions of information flow security are given with the help of *perturbations* and *corrections* [Man03], which we adjust to the asynchronous communication model underlying discrete ports.

On the one hand, asynchronous communication implies that the sending of messages cannot be blocked. Thus, unlike communication by means of synchronous handshake, a component may never actively refuse to receive a message sent by another component. Hence, components can only be perturbed by influencing the messages they receive. This is also reflected by traditional information flow properties such as *generalized noninterference* [McC87], *generalized noninference* [McL94], or *nondeterministic noninterference* [FG95], which all restrict perturbations to incoming information. Since we adopt this approach, critical information is limited to received messages, whereas sent messages will not undergo perturbation. Instead of being perturbed, messages sent over critical ports may be used for corrections, as it is the case for messages exchanged over neutral ports.

On the other hand, non-blocking communication implies that the receiving of a message is not observable. Thus, the sender does not obtain any information about whether the receiver is ready to receive a message. Intuitively, the observable behavior of a component is therefore limited to the messages being sent. Nevertheless, according to the above classification, messages received over observable ports must not depend on critical information as well. We thereby prevent such received messages from being used for corrections, which would only be possible if we assumed that the sending component delivers these messages as needed. In case of observable ports, we avoid this unsafe assumption to ensure that our policies are more freely composable without compromising the security of composite systems. Nevertheless, note that messages received over neutral ports are still allowed to be used for corrections according to our classification. In the upcoming Section 4.6, we will therefore restrict the possible connections of neutral ports to ensure composability of security policies.

In Table 4.1, we summarize the differences induced by the above sensitivity levels. On the one hand, perturbations are restricted to the receiving of messages over *in* (or *in/out*) ports. However, only messages received over critical ports are assumed to be perturbed. In contrast, messages received over neutral or observable ports will not undergo perturbation. On the other hand, corrections are restricted to messages exchanged over neutral ports or sent over critical ports. By contrast, observable ports must not be used for corrections, which would otherwise constitute an unauthorized information flow. In the following, we describe an example policy in Section 4.4.1, before discussing limitations of our policies in Section 4.4.2.

Table 4.1: Perturbations and corrections induced by the sensitivities of ports.

	Critical	Neutral	Observable
	⊘	◎	●
In ◀	Perturbation	Correction	-
Out ▶	Correction	Correction	-

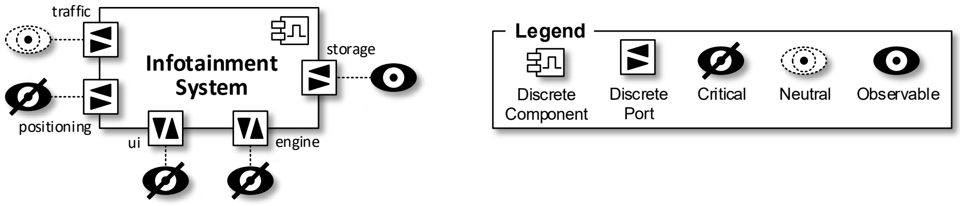


Figure 4.2: Example component security policy of the Infotainment System.

4.4.1 Example Policy

We depict an example security policy for the Infotainment System in Fig. 4.2. Since the positioning, ui, and engine ports are *in/out* ports being classified as critical, they represent sources of unauthorized information flows. Accordingly, the information received over these ports must not become publicly known. Therefore, the receiving of messages over these ports is subject to perturbations. By contrast, the storage port, which represents the component's interface to the storage gateway, is classified as observable. The storage port thereby represents the sink of an unauthorized information flow, and the exchange of information over this port must not enable any conclusions to be drawn about the information that the component receives over its critical ports. Thus, the messages sent to or received from the storage gateway must not be affected by the aforementioned perturbations.

Finally, the traffic port is classified as neutral and therefore represents neither source nor sink of an unauthorized flow. Hence, the information sent over this port can be used for corrections and may therefore depend on critical information. For example, if a geographic location is received from the positioning system over the critical positioning port, it is allowed to be sent to the traffic reporting over the traffic port. Moreover, information received over the traffic port may also become publicly known by influencing the exchange of information over the storage port. If the traffic reporting was malicious, it could therefore exploit the neutrality of the traffic port to circumvent the specified policy, e.g., by storing a the car's location in the cloud. As illustrated by this example, classifying ports as neutral requires special care when assembling multiple components. We will address this challenge by proposing well-formedness rules for component security policies in the upcoming Section 4.6.

4.4.2 Limiting Factors

In general, our component security policies are limited to discrete ports, which are used for message passing. Hence, our policies do currently not restrict the signal exchange over continuous and hybrid ports because this would require an extended, hybrid definition of information flow security, which is beyond the scope of our work. Since continuous components may use continuous ports only (cf. Section 2.3.2), we thereby also limit our policies to discrete and hybrid components. According to these limitations, we assume that a security policy is fully enforceable by controlling the message passing over discrete ports, instead of controlling the signal exchange over hybrid ports.

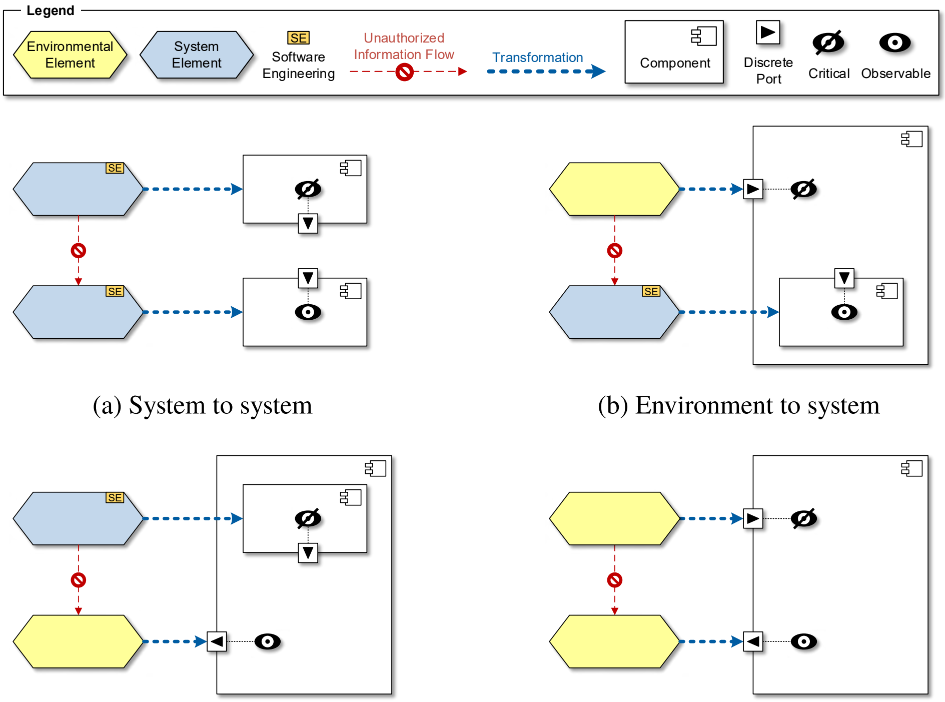
Another limitation relates to the determination of the sensitivities comprised by a component security policy. These sensitivities can be partially derived from the flow policies specified during MBSE, as we will describe in the upcoming Section 4.5. Nevertheless, the derived policies still need to be completed manually by determining the sensitivities of additional discrete ports that have not yet been classified. In general, the sensitivity of a port should reflect the criticality of the exchanged information, whereas the information exchanged by a discrete port corresponds to the messages that are eventually passed over it. However, according to the MechatronicUML process described in Section 2.3.1, these messages will only become known during the behavioral design. Accordingly, they cannot be used to determine the sensitivity of ports during the architectural design. Thus, to pinpoint a port's sensitivity, software architects must anticipate the information content that will be eventually exchanged over that port, and overapproximate the criticality of the exchanged information.

Finally, by restricting perturbations to received messages, we assume that components receive all security-critical information from the outside. Thus, our policies cannot be used to protect critical information that is created by a component internally [Man03, pp. 33, 67].

4.5 Policy Derivation

As described in Section 2.3.1, a MechatronicUML component architecture is derivable from the active structure used in MBSE with Consens. This derivation is based on the transformation of software-relevant system elements into software components [Hol19; Rie15; Ana+14b; Gau+09]. System elements are regarded as software-relevant if they are provided with a relevance annotation that indicates the involvement of software or control engineers. On the basis of these annotations, a system element at the bottom level of the active structure is transformed into a discrete component if it is relevant to software engineering. In contrast, if a bottom-level element is relevant to control engineering, it is transformed into a continuous component. Higher-level system elements are transformed into composite components. Such a composite component is either hybrid if it is recursively composed of at least one continuous subcomponent, or discrete if composed of discrete subcomponents only. Information flows inside the active structure are transformed into connectors between the resulting components, representing either a continuous signal exchange or a discrete message passing.

In this section, we enhance the transition between Consens and MechatronicUML with a derivation of component-based security policies. Since policies are limited to discrete ports (cf. Section 4.4), we focus on the derivation of discrete or hybrid components, which communicate by message passing. To derive component security policies for such components, we take into account a flow policy specified at the level of the active structure (cf. Section 3.5.1) and label a subset of the discrete ports inside the resulting component architecture with corresponding sensitivities. Thereby, we translate an unauthorized information flow from the flow policy into an equivalent component-based representation that indicates sources and sinks. In the following, we establish rules for the derivation in Section 4.5.1. In Section 4.5.2, we apply these rules by deriving example security policies, before discussing the generalization from one to many unauthorized flows in Section 4.5.3.



(a) System to system

(b) Environment to system

(c) System to environment

(d) Environment to environment

Figure 4.3: Rules for the derivation of sensitivities from unauthorized information flows.

4.5.1 Derivation Rules

Figure 4.3 illustrates our derivation rules, which translate unauthorized information flows into sensitivity labels for specific ports. In Fig. 4.3a, we illustrate the translation of an unauthorized flow between two system elements, which correspond to a source and a target component inside the software architecture. According to the unauthorized flow at hand, the information sent by the source component must not influence any information received by the target component. Therefore, discrete ports of the source component are classified as critical if they enable messages to be sent, which is the case for *out* and *in/out* ports. Furthermore, discrete ports of the target component are classified as observable if they enable messages to be received, as it is the case for *in* and *in/out* ports. Hence, the policy from Fig. 4.3a excludes an information flow from the source to the target component.

Figure 4.3b shows the translation of an unauthorized information flow from an environmental element to a system element. Similar to Fig. 4.3a, the discrete *in* and *in/out* ports of the corresponding target component are classified as observable. In addition, discrete *in* and *in/out* ports of the top-level component are classified as critical, provided that they represent the communication with the respective environmental element. The resulting security policy from Fig. 4.3b restricts the architecture such that the information received over specific external ports must not flow to a particular internal component.

Analogously, Fig. 4.3c illustrates the reverse case of an unauthorized flow from a system element to an environmental element. In this case, the *out* or *in/out* ports of the corresponding source component are classified as critical, whereas *out* and *in/out* ports of the top-level component are observable if they represent the communication with the respective environmental element. According to the resulting policy shown in Fig. 4.3c, no information must flow from an internal component to specific external ports of the architecture.

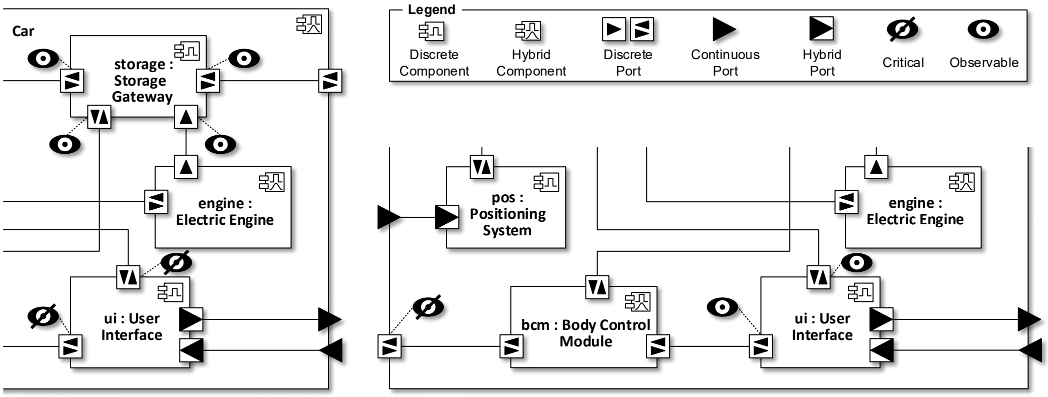
Finally, in Fig. 4.3d, we address an unauthorized flow between two environmental elements. By analogy with Fig. 4.3b and Fig. 4.3c, any *in* or *in/out* ports that receive information from the source element are critical, whereas *out* or *in/out* ports that send information to the target element are observable. Thereby, the security policy in Fig. 4.3d prevents an end-to-end flow of critical information between specific external ports of the architecture.

In summary, our rules illustrated in Fig. 4.3 will always derive a critical sensitivity label from the source of an unauthorized information flow, and an observable label from the target of a flow. The result of a derivation by means of the proposed rules is a partial security policy that still needs to be manually completed. This completion requires software architects to label any discrete ports that have not been classified during the automated derivation. We consider this completion as a *refinement* of a component security policy, which must be well-formed in order to fully enforce the security restrictions specified by the underlying flow policy. We address the well-formedness of such refinements in the upcoming Section 4.6.

4.5.2 Example Derivation

In Fig. 4.4, we show two partial security polices, which are derived from the flow policy of the self-driving car shown in Fig. 3.3 on page 41. In particular, the partial policy in Fig. 4.4a is derived from the unauthorized flow between User Interface and Storage Gateway. Thus, the resulting partial policy represents a confidentiality requirement that prevents user data from being leaked to the cloud. The unauthorized flow at hand restricts two system elements. Hence, according to the derivation rule from Fig. 4.3a, the discrete ports of the User Interface (which is the source of the flow) are classified as critical. In particular, both *in/out* ports are critical because they enable the sending of messages. Furthermore, the discrete ports of the Storage Gateway are classified as observable and thereby indicate the target of the unauthorized flow. Again, this labeling affects each discrete port of the component because all of them are either *in* or *in/out* ports, which enable messages to be received. Since the discrete ports of other components such as the Electric Engine have not yet been classified, the depicted labeling represents a partial security policy.

In contrast, the partial policy depicted in Fig. 4.4b is derived from the unauthorized flow between Predictive Maintenance and User Interface. Hence, it specifies an integrity requirement, preventing manipulation of the information displayed to occupants. The unauthorized flow at hand affects an environmental and a system element. According to the rule from Fig. 4.3b, the external maintenance port of the top-level component is classified as critical, representing the source of the flow. Since the User Interface is the target of the flow, its discrete ports are classified as observable. Again, the derived policy is partial because the discrete ports of other components like Positioning System or Electric Engine have not been classified yet.



(a) Confidentiality policy

(b) Integrity policy

Figure 4.4: Partially derived component security policies of the self-driving Car.

4.5.3 Generalization

Each of the derivation rules established in Section 4.5.1 refers to an individual unauthorized information flow. However, a flow policy may generally include more than one such flow. In particular, one and the same element may also act as the source and target of two distinct unauthorized flows. For example, in Fig. 3.3 on page 41, this is the case for the User Interface. However, as described in Section 4.5.1, source elements are always classified as critical, whereas target elements are generally classified as observable. Thus, in this situation, deriving a shared component security policy from both flows in combination leads to conflicting sensitivities for the discrete ports of the User Interface. This is also illustrated by the different sensitivities assigned in Fig. 4.4a and Fig. 4.4b, which would conflict with each other when combined into a shared policy. To ensure consistency of component-based security policies, such conflicts need to be avoided.

Even if multiple flows do not lead to conflicting sensitivities, a shared security policy can still be overly restrictive. In such a case, the component security policy renders an information flow unauthorized although it is authorized according to the flow policy. Thus, the intended security restrictions of the flow policy are enforced, but not precisely preserved. In order to avoid conflicting or overly restrictive policies, we generally assume that each individual unauthorized flow is handled separately, deriving one distinct component security policy per unauthorized flow.

In the general case, we therefore assume that multiple distinct security policies are derived from a single flow policy. This is in accordance with the extended MechatronicUML process depicted in Fig. 4.1, which refers to a collection of multiple component security policies. In the downstream steps of the process, all of these policies need to be handled separately. First, each of the derived policies must be refined in a well-formed manner. Second, the component behavior determined during the behavioral design (cf. Section 2.3.3) must adhere to *all* of the resulting security policies as is.

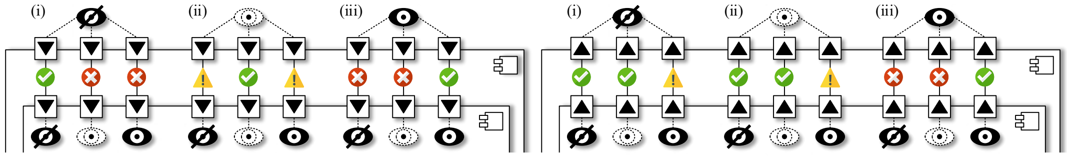
4.6 Well-Formedness of Refinements

The MechatronicUML process from Fig. 4.1 requires software architects to refine a component-based security policy under two different conditions. First, a partial policy must be completed after it is derived from a flow policy. Second, on decomposition of a component, the ports of the resulting subcomponents must be included in the policy. In both cases, additional discrete ports must be classified by labeling them with sensitivities, which must be compatible with the pre-existing sensitivities of connected ports. To this end, we propose a set of well-formedness rules that require any two connected ports to be classified compatibly. The proposed rule set is based on the formal constraints for the composition of secure systems by Mantel [Man02], which we transfer to the level of CBSE. In particular, we adjust the rules to the asynchronous communication between components in MechatronicUML.

Our proposed rules are applicable to particular connectors between two ports. We restrict the presentation of our rules to unidirectional connections between *in* or *out* ports, whereas a bidirectional connection between *in/out* ports must follow the rules for both individual directions. The application of a rule leads to one of three different results, assessing the compatibility of the connected ports. First, a well-formed refinement is denoted by ✓, indicating that the intended security restrictions are enforced by a connection, without being obviously enforceable by less restrictive sensitivities. In this case, software architects are not required to revise the associated sensitivities and may proceed with the process as described in Section 4.3. Second, ✗ denotes an ill-formed refinement, which applies if a connection does not enforce the intended security restrictions and therefore requires architects to revise the sensitivities of the connected ports. Third, we denote a refinement by ⚠ if it is neither well-formed nor ill-formed. This is the case if the intended security restrictions are enforced, but could obviously be enforced by less restrictive sensitivities. In this case, it is strongly recommended that architects revise the associated sensitivities. In the following, we present well-formedness rules for delegations in Section 4.6.1 and for assemblies in Section 4.6.2. Subsequently, we provide software architects with best practices for the refinement in Section 4.6.3 and illustrate the refinement with an example in Section 4.6.4.

4.6.1 Delegation

The information exchange between composite and subcomponents over delegation connectors creates two different obligations. First, the composite component relies on the fact that its security policy is enforced by its constituent subcomponents. Thus, subcomponents must treat critical information reliably without downgrading its criticality, which would otherwise enable unauthorized information flows. Second, the security policy of the composite component guarantees that observable information does not depend on critical information, which is a guarantee that must be provided by the subcomponents. To meet these obligations, the ports bound by a delegation connector must be classified in a well-formed manner. In the following, we provide dedicated well-formedness rules for incoming delegations between *in* (or *in/out*) ports and for outgoing delegations between *out* (or *in/out*) ports.



(a) *In* ports (b) *Out* ports

Figure 4.5: Well-formedness rules for delegations [*GS19].

Incoming

We depict the well-formedness rules for *in* ports in Fig. 4.5a (i)–(iii). If a subcomponent receives critical information from a composite component, it must meet the obligation to process this information reliably without downgrading its criticality. To ensure this form of reliability, the perturbations supported by the composite component must also be supported by the subcomponent. It is therefore mandatory that a critical *in* port of a composite component only delegates to critical *in* ports of subcomponents, whereas delegations to neutral or observable *in* ports are both ill-formed (i). In contrast, since a neutral *in* port of a composite component is not affected by the two aforementioned obligations, it may delegate to arbitrarily classified *in* ports of subcomponents (ii). However, only a delegation to a neutral *in* port is well-formed because treating the received information as critical or observable would restrict the information flow of the subcomponent without any reason. Finally, an observable *in* port of a composite component must not depend on critical information. To provide this guarantee given by the composite component, it is mandatory that such a port delegates to another observable *in* port of a subcomponent, whereas other connections are ill-formed (iii).

Outgoing

In Fig. 4.5b (i)–(iii), we depict our well-formedness rules for *out* ports. As illustrated in Table 4.1, we assume both critical and neutral *out* ports to be used for corrections. Therefore, their associated rules (i–ii) are equivalent because the delegation is not affected by the aforementioned obligations. First, since only received information is considered critical, the port of the subcomponent does not have to be reliable with respect to the criticality of information. Second, the port of the subcomponent does not need to guarantee that the information being sent will not depend on critical information. Therefore, a critical (i) or neutral (ii) *out* port of a composite component may delegate to arbitrarily classified ports of subcomponents. However, a delegation to an observable *out* port leads to an overly restrictive policy because the subcomponent will never use that port for corrections, opposed to the assumption of the composite component. In contrast, only a delegation to a critical or neutral *out* port is actually well-formed. Finally, an observable *out* port of a composite component must not depend on critical information. To provide this guarantee, it may only delegate to observable *out* ports of subcomponents (iii), whereas a delegation to a critical or neutral *out* port is ill-formed.

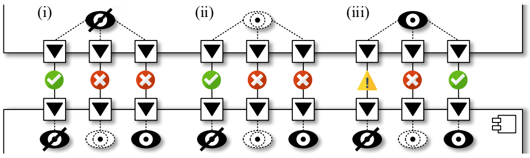


Figure 4.6: Well-formedness rules for assemblies [*GS19].

4.6.2 Assembly

When information is sent from one component to another over assembly connectors, it requires special care if it allows critical conclusions to be drawn. Thus, such information must not be circulated by the receiving component in an uncontrolled manner. We depict the respective well-formedness rules for assembly connectors between a source and a target component in Fig. 4.6 (i)–(iii). Similar to Fig. 4.5b, the rules for critical and neutral *out* ports are equivalent because both are potentially used for corrections (cf. Table 4.1) and may thereby depend on critical information. To prevent such information from being observable, both critical (i) and neutral (ii) *out* ports of the source component must be mandatorily assembled with critical *in* ports of the target component. Thus, connections to neutral or observable *in* ports are both ill-formed because they would enable the exchanged information to be disclosed to the public.

In contrast, an observable *out* port of the source component (iii) must not be assembled with a neutral *in* port because such a port may be used by the target component for corrections (cf. Table 4.1). As described in Section 4.4, the target component thereby assumes that the source component will always support such corrections by sending the right messages as needed. To prevent this wrong assumption, an assembly connector between an observable *out* port of the source component and a neutral *in* port of the target component is ill-formed. Furthermore, assembling the observable *out* port with a critical *in* port leads to an overly restrictive security policy: whereas the *in* port of the target component assumes to receive information that allows critical conclusions to be drawn, no such information is ever sent by the source component over its observable *out* port. Therefore, the refinement is only well-formed when assembling the observable *out* port with an observable *in* port.

4.6.3 Best Practices

The established rule set provides software architects with an analytical, reactive assessment regarding the well-formedness of the refined policies. However, due to their level of detail, the proposed rules are not suited as a constructive guideline that architects may establish as a proactive mindset. In particular, our rules refer to a one-way communication over *in* and *out* ports. Thus, a two-way communication over *in/out* ports requires architects to follow the rules for both ways of communication separately. We therefore condense our well-formedness rules and provide software architects with best practices for the refinement of security policies in the presence of *in/out* ports [*GS18]. Whereas the following best practices (BP1–BP3) are in accordance with the proposed well-formedness rules, they guide software architects by means of concrete recommended actions for the refinement:

(BP1) Inheritance on Delegation: The *in/out* ports of subcomponents should always inherit the sensitivity level from the delegating ports of their composite component. As depicted in Fig. 4.5, such a refinement is always well-formed. This practice ensures that the sensitivities of ports are not upgraded or downgraded by subcomponents.

(BP2) Non-Neutrality on Assembly: Assembly connectors must not be bound to neutral *in/out* ports, as can be seen from Fig. 4.6. We thereby prevent components from using received messages for corrections because an assembled component might not deliver these messages as required.

(BP3) Equivalence on Assembly: Assembled components should agree on the sensitivity of their connected ports. Therefore, *in/out* ports bound by an assembly connector should share equivalent sensitivity levels. As shown in Fig. 4.6, such a refinement is well-formed for critical and observable ports.

4.6.4 Example Refinement

In the following, we apply the best practices from Section 4.6.3 to the software architecture of the self-driving car. In Fig. 4.7, we complete the partial confidentiality policy from Fig. 4.4a. The resulting policy is well-formed and thereby classifies the discrete ports in such a way that no information is leaked from the User Interface to the Storage Gateway. To this end, particular subcomponents of the Car are restricted such that they must not leak specific information to the Storage Gateway. First, the Body Control Module must not leak any information received from the User Interface. Second, the Infotainment System must not disclose any information received from the Positioning System, the User Interface, or the Electric Engine. Third, the Electric Engine must not circulate the information received from the Infotainment System.

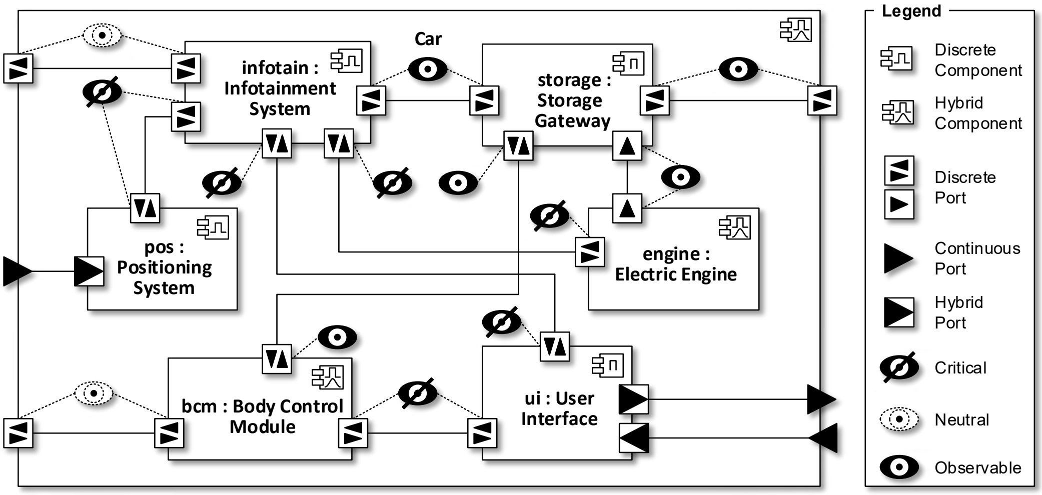


Figure 4.7: Completed confidentiality policy of the self-driving Car.

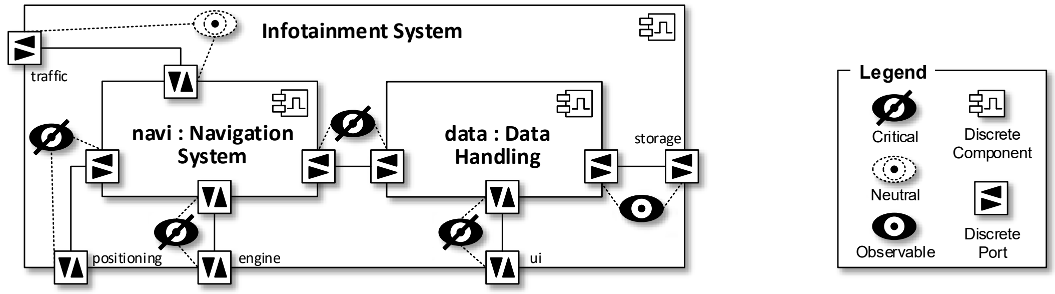


Figure 4.8: Refined confidentiality policy of the Infotainment System.

Since User Interface and Storage Gateway represent source and sink of the unauthorized flow that needs to be prevented, these subcomponents are not effectively restricted because all of their discrete ports are labeled with equivalent sensitivities. Since their security policies do not combine critical and observable sensitivities, these components are secure by definition and the information flow security of their behavior does not have to be verified. This is also true for the Positioning System because it is limited to a single discrete port.

Note that the resulting security policy of the Infotainment System corresponds to the policy known from Fig. 4.2. In Fig. 4.8, we further refine this policy on decomposition of the component as illustrated in Fig. 2.7b on page 22. Again, the refinement is well-formed and therefore ensures that the Navigation System and Data Handling subcomponents enforce the security policy of their composite component. On the one hand, the policy of the Data Handling requires that critical information received from the Navigation System or from the User Interface through the ui port must not be disclosed over the observable storage port. On the other hand, the information flow of the Navigation System is not effectively restricted because its security policy does not combine critical and observable sensitivities. According to this policy, the Navigation System is secure by definition. Thus, by means of the depicted refinement, the security restrictions of the Infotainment System have been narrowed down to finer-grained restrictions of the Data Handling without restricting the Navigation System.

4.7 Composability

In this section, we refer back to the requirements specified in Section 4.2. Whereas the specification of security restrictions (R1) has been enabled by the policies introduced in Section 4.4, the security management (R2) is not addressed directly at the level of the MechatronicUML component model (cf. Section 4.2). We therefore focus on the composability (R3). Provided that policies are refined according to our well-formedness rules established in Section 4.6, we show that a global policy can be securely composed of local policies. To this end, Section 4.7.1 defines a security property that must be satisfied by a component in order to adhere to a given policy. Subsequently, in Section 4.7.2, we show that this property will be preserved by any composite component that is composed of secure subcomponents.

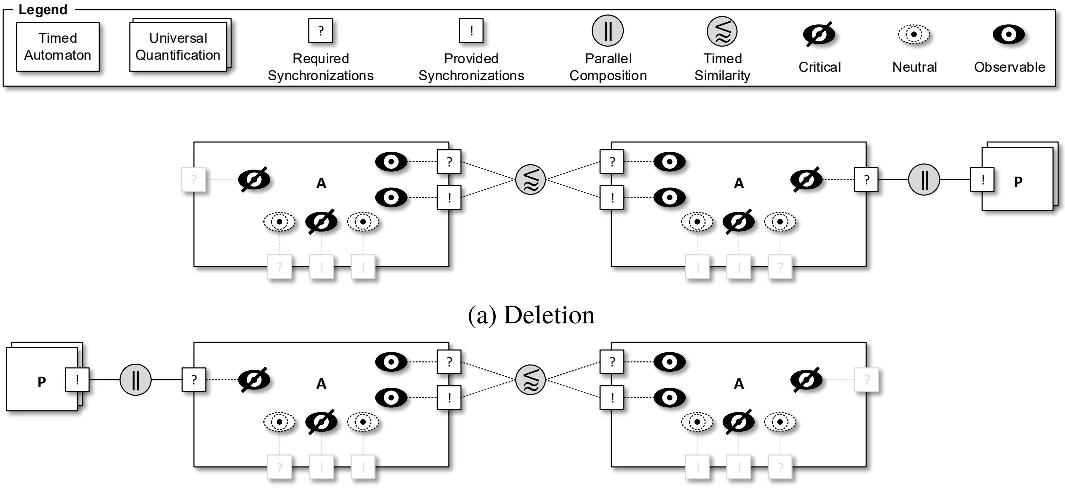
4.7.1 Defining Security

In the following, we provide our component-based security policies with an underlying information flow property. As described in Section 2.4.2, we must address the problem that information flow properties are not generally composable [Man02] because one component might block the communication behavior of another component. Thus, security properties of individual components might no longer hold on composition. This problem becomes even more challenging in case of feedback composition [ZL96] because the two-way communication enables components to block each other mutually. To ensure composability, the information flow property we select is *generalized noninterference* as introduced by McCullough [McC87]. This property is known to be composable even in case of feedback, provided that components communicate asynchronously [ZL95]. In this case, a sending component cannot be blocked by a receiving component, such that the security of the individual components still holds on composition. Below, we first consider the perturbations involved by generalized noninterference, before we transfer the property to real-time systems.

Perturbations

Generalized noninterference involves two different forms of perturbations, which are (i) the *deletion* of critical inputs from an execution and (ii) the *insertion* of critical inputs into an execution [Man03]. In both cases, the observable inputs and outputs processed during that execution must not be affected by the perturbation. Thus, from the perspective of an observer, the perturbed and unperturbed executions must be indistinguishable. Observations that can be made in the absence of a perturbation must still be possible in the presence of that perturbation. This suggests the use of an inclusion relation to encode generalized noninterference: the observable behavior of an unperturbed system must be fully included in the observable behavior of the perturbed one. In the context of formal languages, a similar approach is followed by D'Souza et al. [DHRS11] who encode perturbations by means of language operations and define information flow security in terms of language inclusion.

As described in Section 2.3.3, we assume the component behavior to be given in the form of timed automata, for which language inclusion is undecidable [AD94] and therefore unsuitable for the definition of verifiable information flow properties. Instead, we relate the observable behaviors of automata by means of a *simulation* preorder [Mil71], which is an even stronger condition [cf. Sti03]. Accordingly, a perturbed automaton must be able to simulate each observable execution step of an unperturbed automaton. We illustrate this approach in Fig. 4.9, depicting the timed automaton A reduced to its communication interface. Required (?) and provided (!) synchronizations represent received and sent messages of a component. These synchronizations are classified according to the sensitivity of a discrete port, over which the represented messages are assumed to be passed. We generally assume that components combine all possible sensitivities and both directions of communication. Therefore, the depicted automata include required and provided synchronizations at each possible sensitivity level. Please note that, in the following, ? and ! do not necessarily represent a single synchronization, but a set of synchronizations with a common sensitivity.



(a) Deletion

(b) Insertion

Figure 4.9: Perturbations of critical inputs required by Generalized Noninterference.

In Fig. 4.9a, we illustrate the perturbation by deletion. On the left, we depict the unperturbed automaton A. On the right, we encode the deletion of critical inputs by composing A in parallel with a *perturbator* automaton P. The perturbator may or may not provide a critical synchronization required by the automaton A. In contrast, the unperturbed A may freely execute any of the critical synchronizations it requires. Thus, during execution, it may be the case that a critical input is processed by the unperturbed automaton, whereas the perturbed automaton is unable process the same input. Regardless of such deletions, the observable synchronizations required and provided by the unperturbed automaton must also be required or provided by the perturbed one. In Fig. 4.9a, we illustrate this relation by means of a *timed similarity* between both automata, as described in Section 2.3.3. Unlike observable synchronizations, all neutral and critical synchronizations are treated as *hidden* (cf. Section 2.3.3) and thereby excluded from the timed simulation. Hence, during execution, the perturbed and unperturbed automata may differ arbitrarily with respect to all neutral synchronizations as well as critical synchronizations being provided. Therefore, any of these synchronizations may be used for corrections (cf. Section 2.4.2). In Fig. 4.9, we depict hidden synchronizations grayed out.

For the perturbation by insertion illustrated in Fig. 4.9b, we need to represent the case that the perturbed automaton receives a critical input whereas the unperturbed automaton does not. We encode this requirement indirectly by reattaching P from the right to the left automaton. Hence, whenever a critical synchronization required by the left automaton is not provided by P, the right automaton may freely execute the same synchronization, which is thereby regarded as inserted into the execution. However, since we encode the insertion on the right by means of deletion on the left, we only insert synchronizations that are *admissible* [Man03]. Regardless of this perturbation, the right automaton must still simulate the left automaton, as illustrated by the timed similarity in Fig. 4.9b.

Timed Generalized Noninterference

The difference between Figs. 4.9a and 4.9b can also be considered as a change in the direction of the timed similarity. In fact, Fig. 4.9 effectively represents two simulations of the perturbed by the unperturbed automaton and vice versa. This observation enables us to define information flow security by means of a *bisimulation* [Sti98], which implies that two automata simulate each other in both directions. Bisimulation is a long-established instrument for the formal definition of information flow properties [FG95]. Thus, we rely on bisimulations to transfer generalized noninterference to the real-time behavior underlying MechatronicUML. In particular, we use timed bisimilarity ($\approx$) as described in Section 2.3.3 to define a notion of generalized noninterference that applies to timed automata:

Definition 4.1 (Timed Generalized Noninterference)**.** Let A be a timed automaton with the alphabet $\Sigma(A) = I \uplus O \uplus \{\tau\} = V \uplus N \uplus C \uplus \{\tau\}$. The automaton A satisfies *timed generalized noninterference*, if and only if $(P \parallel A) \;/\; (N \cup C) \approx A \;/\; (N \cup C)$ for each P with $\Sigma(P) \subseteq \overline{C \cap I} \uplus \{\tau\}$.

According to Definition 4.1, we require a timed bisimulation between the unperturbed automaton A and the perturbed automaton $P \parallel A$ for each perturbator P. Since the perturbator must only provide A with critical inputs, it is restricted to complementary synchronizations from the set $\overline{C \cap I}$. Since we apply the hiding operator / (cf. Section 2.3.3) to all neutral and critical synchronizations from the set $N \cup C$, the bisimulation is restricted to observable synchronizations from V. By involving a universal quantification over all P in Definition 4.1, our notion of information flow security resembles an existing property called *nondeducibility on compositions* [FG95]. In Chapter 5, we will address the challenge of handling this universality during verification.

4.7.2 Preserving Security

In the following, we assume that timed generalized noninterference (cf. Definition 4.1) holds for two timed automata A_1 with $\Sigma(A_1) = \Sigma_1 = I_1 \uplus O_1 \uplus \{\tau\} = V_1 \uplus N_1 \uplus C_1 \uplus \{\tau\}$ and A_2 with $\Sigma(A_2) = \Sigma_2 = I_2 \uplus O_2 \uplus \{\tau\} = V_2 \uplus N_2 \uplus C_2 \uplus \{\tau\}$. A_1 and A_2 represent the behaviors of two subcomponents. We depict this constellation in Fig. 4.10. To provide for the composition of A_1 and A_2, both are equipped with external synchronizations at the outside of Fig. 4.10 (representing communication of a subcomponent with a composite component over delegation connectors), as well as internal synchronizations at the inside of Fig. 4.10 (enabling communication between subcomponents over assembly connectors). We consider only connections that are not ill-formed. Thus, since neutral *in* ports are always ill-formed in the context of assemblies (cf. Fig. 4.6), none of the internal inputs of A_1 and A_2 are neutral. Furthermore, as a deviation from Definition 4.1, we divide the perturbator automaton P into P^i for the perturbation of internal and P^e for the perturbation of external communication. In particular, we compose A_1 in parallel with the perturbators P_1^e and P_1^i, and A_2 with the perturbators P_2^e and P_2^i. In the following, we gradually extend Fig. 4.10 to demonstrate that the depicted bisimulations are preserved under composition of A_1 and A_2. In accordance with Fig. 4.10, the following holds:

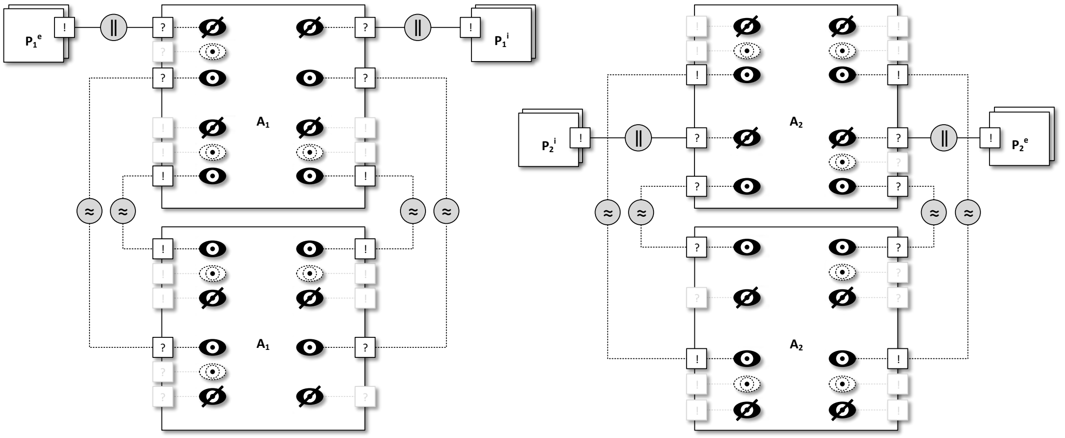


Figure 4.10: Timed generalized noninterference as a property of the automata A_1 and A_2.

$$\begin{aligned} \forall P_1^e, P_1^i : P_1^e \parallel A_1 \parallel P_1^i \,/\, (N_1 \cup C_1) \approx A_1 \,/\, (N_1 \cup C_1) \\ \forall P_2^e, P_2^i : P_2^e \parallel A_2 \parallel P_2^i \,/\, (N_2 \cup C_2) \approx A_2 \,/\, (N_2 \cup C_2) \end{aligned} \tag{4.1}$$

As described in Section 2.3.3, the asynchronous communication in MechatronicUML can be encoded by synchronizing the communicating automata with an intermediary automaton [KMR02]. To encode an output, the intermediate automaton requires a synchronization from the sending automaton, before providing a second, delayed synchronization to the receiving automaton to encode the input. Thus, we introduce auxiliary automata C_s to represent assembly connectors, enabling the subcomponents to communicate at a specific sensitivity level s. This level s reflects the sensitivity of the required synchronizations used to receive the corresponding inputs. Since inputs can only be critical or observable in case of an assembly (cf. Fig. 4.6), we assume $s \in \{c, v\}$. To enforce asynchronous communication over connector automata and thereby prevent A_1 and A_2 from synchronizing with each other directly, we assume that $\Sigma_1 \cap \Sigma_2 = \{\tau\}$, which can be achieved by renaming shared messages. Since connector automata enable a one-way communication, we denote them by $C_s^{\leftarrow}$ or $C_s^{\rightarrow}$ to indicate their direction. Due to the asynchronous communication, the sending of messages is non-blocking: a component can always send every message without restriction of any kind. We therefore assume every C_s to be *input-total* (cf. Section 2.3.3).

We show that timed generalized noninterference also holds for an automaton A with $\Sigma(A) = I \uplus O \uplus \{\tau\} = V \uplus N \uplus C \uplus \{\tau\}$. Since A results from the feedback composition of A_1 and A_2 with $\Sigma(A) \subseteq \Sigma_1 \cup \Sigma_2$, it represents the behavior of a composite component. As shown in Fig. 4.11, we use connector automata to establish the internal communication[2], such that $A = A_1 \parallel C_c^{\leftarrow} \parallel C_v^{\leftarrow} \parallel C_c^{\rightarrow} \parallel C_v^{\rightarrow} \parallel A_2$. The depicted synchronizations reflect our well-formedness rules for assemblies from Fig. 4.6: critical and neutral outputs can only lead to critical inputs, whereas observable outputs may lead to both critical and observable inputs.

[2] The depicted form of composition is known as *hookup* [McL96], enabling external communication as well.

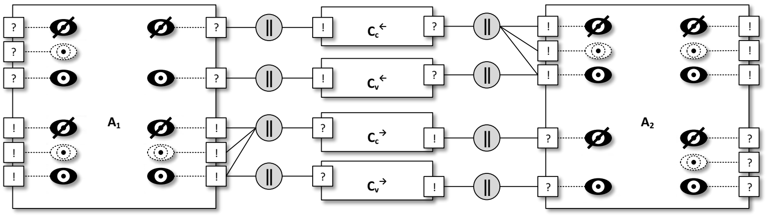


Figure 4.11: Composite automaton A resulting from the composition of A_1 and A_2.

To show that A preserves information flow security, we start by focusing on A_1 only. Due to the universal quantification in Eq. (4.1), we may replace the generic perturbator P_1^i with a specific connector $C_c^{\leftarrow}$ that provides A_1 with critical inputs. As shown in Fig. 4.11, these critical inputs in C_1 can be caused by outputs from Σ_2 with arbitrary sensitivity levels, such that $\Sigma(C_c^{\leftarrow}) \subseteq C_1 \cup \Sigma_2$. Thus, we need to hide Σ_2 in order to preserve the bisimulation:

$$\forall P_1^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2) \approx A_1 \,/\, (N_1 \cup C_1) \tag{4.2}$$

Due to the universal quantification over all perturbators in Eq. (4.2), the bisimulation also holds for a perturbator that is always ready to synchronize with A_1. However, such a construction is equivalent to omitting the perturbator entirely, enabling us to detach P_1^e from Eq. (4.2) without violating the bisimulation. Thus, it holds that $A_1 \,/\, (N_1 \cup C_1) \approx A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2)$. Therefore, we can replace the right-hand side of Eq. (4.2) as follows:

$$\forall P_1^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2) \approx A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2) \tag{4.3}$$

At this point, we attached the connector automaton $C_c^{\leftarrow}$ both to the perturbed automaton $P_1^e \parallel A_1$ on the left of Eq. (4.3) and the unperturbed automaton A_1 on the right. In the next step, we make use of the fact that weak timed bisimulation is preserved both under the composition operator $|$ and the restriction operator $\setminus$, as shown by Yi [Yi91]. Thus, in particular, it is also preserved under the $\parallel$ operator, which combines $|$ and $\setminus$ [Ben+96]. Therefore, we may compose both sides of Eq. (4.3) in parallel with the automaton $C_c^{\rightarrow} \parallel A_2$, whereas the bisimulation is preserved:

$$\begin{aligned} \forall P_1^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2) \parallel C_c^{\rightarrow} \parallel A_2 & \\ \approx A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2) \parallel C_c^{\rightarrow} \parallel A_2 & \end{aligned} \tag{4.4}$$

Next, we use the fact that timed generalized noninterference holds for A_2 as per Eq. (4.1). In particular, it holds that $\forall P_2^e : C_c^{\rightarrow} \parallel A_2 \parallel P_2^e \,/\, (N_2 \cup C_2) \approx C_c^{\rightarrow} \parallel A_2 \,/\, (N_2 \cup C_2)$. We may therefore compose the right-hand side of Eq. (4.4) in parallel with the perturbator P_2^e, provided that we hide neutral and critical synchronizations in $N_2 \cup C_2$ on both sides:

$$\begin{aligned} \forall P_1^e, P_2^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2) \parallel C_c^{\rightarrow} \parallel A_2 \parallel P_2^e \,/\, (N_2 \cup C_2) & \\ \approx A_1 \parallel C_c^{\leftarrow} \,/\, (N_1 \cup C_1 \cup \Sigma_2) \parallel C_c^{\rightarrow} \parallel A_2 \,/\, (N_2 \cup C_2) & \end{aligned} \tag{4.5}$$

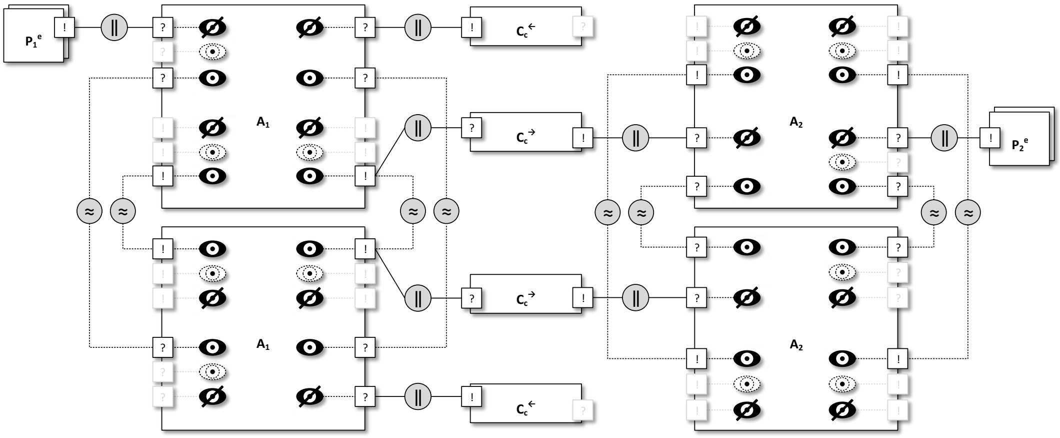


Figure 4.12: A_1 and A_2 composed in parallel with connector automata $C_c^{\leftarrow}$ and $C_c^{\rightarrow}$.

Up to now, we connected A_1 and A_2 by means of the connector automata $C_c^{\rightarrow}$ and $C_c^{\leftarrow}$. However, due to the hiding of $N_1 \cup C_1$, the communication over $C_c^{\rightarrow}$ is restricted to observable outputs of A_1, leading to critical inputs of A_2. We depict this restriction in Fig. 4.12. In particular, both critical and neutral synchronizations are hidden from A_1 and therefore assumed to be replaced by τ (cf. Section 2.3.3). However, we may show that the bisimulation in Eq. (4.5) still holds when unhiding neutral and critical communication from A_1 to A_2, synchronizing the corresponding outputs with $C_c^{\rightarrow}$ instead. This is due to the assumed input totality of connector automata, which will never prevent a connecting automaton from sending an output. Thus, whenever A_1 can execute a τ that hides a critical or neutral output, then $A_1 \parallel C_c^{\rightarrow}$ can synchronize on the unhidden output as well. We make use of this property by enabling $C_c^{\rightarrow}$ to synchronize with A_1 on neutral or critical outputs. To this end, we move the hiding of $N_1 \cup C_1$ to the end of the terms on both sides of the equation:

$$\begin{aligned}\forall P_1^e, P_2^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \,/\, \Sigma_2 \parallel C_c^{\rightarrow} \parallel A_2 \parallel P_2^e \,/\, (N_2 \cup C_2) \,/\, (N_1 \cup C_1)& \\ \approx A_1 \parallel C_c^{\leftarrow} \,/\, \Sigma_2 \parallel C_c^{\rightarrow} \parallel A_2 \,/\, (N_2 \cup C_2) \,/\, (N_1 \cup C_1)& \qquad (4.6)\end{aligned}$$

Furthermore, Fig. 4.12 also shows that the hiding of Σ_2 prevents A_2 from sending any outputs over $C_c^{\leftarrow}$. Again, we may enable the communication over $C_c^{\leftarrow}$ thanks to the input totality of connector automata. Thus, for any synchronization provided by A_2, there will always be a corresponding required synchronization in the connector $C_c^{\leftarrow}$. As depicted in Fig. 4.13, this enables us to remove the hiding of Σ_2 in Eq. (4.6). At the same time, we may join the hiding of $N_2 \cup C_2$ and $N_1 \cup C_1$, leading to the following bisimulation:

$$\begin{aligned}\forall P_1^e, P_2^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \parallel C_c^{\rightarrow} \parallel A_2 \parallel P_2^e \,/\, (N_1 \cup C_1 \cup N_2 \cup C_2)& \\ \approx A_1 \parallel C_c^{\leftarrow} \parallel C_c^{\rightarrow} \parallel A_2 \,/\, (N_1 \cup C_1 \cup N_2 \cup C_2)& \qquad (4.7)\end{aligned}$$

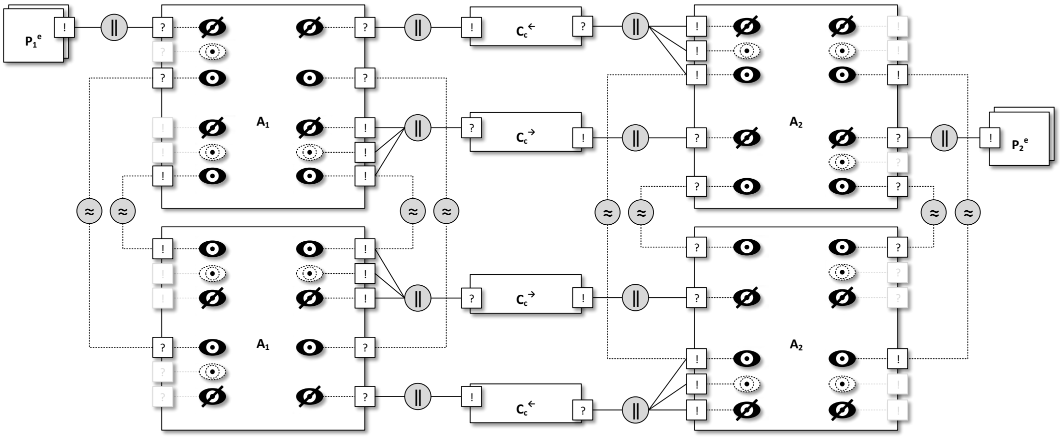


Figure 4.13: Unhiding of outputs due to the input totality of $C_c^{\leftarrow}$ and $C_c^{\rightarrow}$.

In the next step, we again exploit the fact that weak timed bisimulation is preserved under the $|$ and $\backslash$ operators [Yi91], thereby ensuring preservation under parallel composition ($\|$) as well. Therefore, we may safely compose both sides of Eq. (4.7) in parallel with the missing connectors $C_v^{\leftarrow}$ and $C_v^{\rightarrow}$, whereas the bisimulation is preserved:

$$\begin{aligned}\forall P_1^e, P_2^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \parallel C_c^{\rightarrow} \parallel A_2 \parallel P_2^e \,/\, (N_1 \cup C_1 \cup N_2 \cup C_2) \parallel C_v^{\leftarrow} \parallel C_v^{\rightarrow} \\ \approx A_1 \parallel C_c^{\leftarrow} \parallel C_c^{\rightarrow} \parallel A_2 \,/\, (N_1 \cup C_1 \cup N_2 \cup C_2) \parallel C_v^{\leftarrow} \parallel C_v^{\rightarrow}\end{aligned} \tag{4.8}$$

According to Fig. 4.11, the connector automata $C_v^{\leftarrow}$ and $C_v^{\rightarrow}$ will only provide and require observable synchronizations from the set $V_1 \cup V_2$. It follows that $\Sigma(C_v) \cap (N_1 \cup C_1 \cup N_2 \cup C_2) = \varnothing$. This enables us to include $C_v^{\leftarrow}$ and $C_v^{\rightarrow}$ in the scope of the hiding operator without violating the bisimulation:

$$\begin{aligned}\forall P_1^e, P_2^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \parallel C_v^{\leftarrow} \parallel C_c^{\rightarrow} \parallel C_v^{\rightarrow} \parallel A_2 \parallel P_2^e \,/\, (N_1 \cup C_1 \cup N_2 \cup C_2) \\ \approx A_1 \parallel C_c^{\leftarrow} \parallel C_v^{\leftarrow} \parallel C_c^{\rightarrow} \parallel C_v^{\rightarrow} \parallel A_2 \,/\, (N_1 \cup C_1 \cup N_2 \cup C_2)\end{aligned} \tag{4.9}$$

Finally, from our well-formedness rules for delegations depicted in Fig. 4.5, we may conclude that a neutral or critical port of a subcomponent may only be connected to neutral or critical ports of a composite component, whereas other connections are ill-formed. Thus, it holds that $N_1 \cup C_1 \cup N_2 \cup C_2 \subseteq N \cup C$. We may therefore extend the hiding of neutral and critical synchronizations to the set $N \cup C$ as follows:

$$\begin{aligned}\forall P_1^e, P_2^e : P_1^e \parallel A_1 \parallel C_c^{\leftarrow} \parallel C_v^{\leftarrow} \parallel C_c^{\rightarrow} \parallel C_v^{\rightarrow} \parallel A_2 \parallel P_2^e \,/\, (N \cup C) \\ \approx A_1 \parallel C_c^{\leftarrow} \parallel C_v^{\leftarrow} \parallel C_c^{\rightarrow} \parallel C_v^{\rightarrow} \parallel A_2 \,/\, (N \cup C)\end{aligned} \tag{4.10}$$

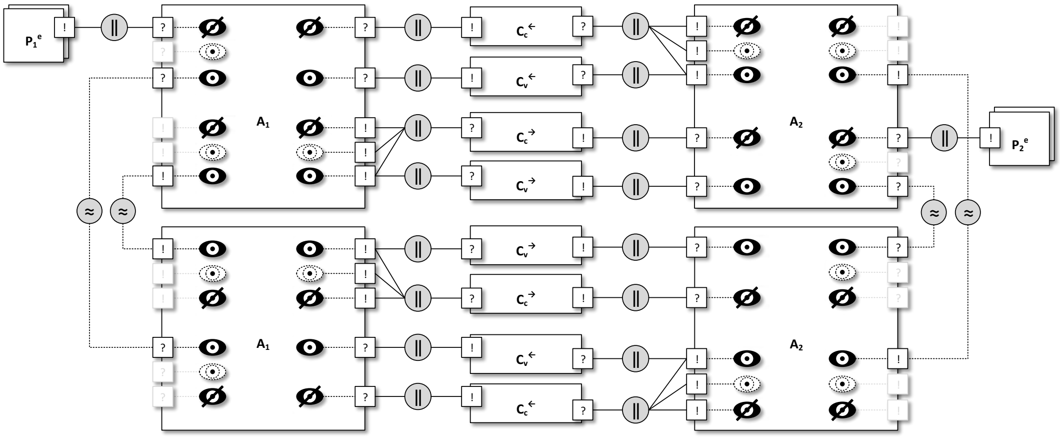


Figure 4.14: Timed generalized noninterference as a property of the composite automaton A.

We may conclude from Eq. (4.10) that timed generalized noninterference holds for the automaton A representing the behavior of a composite component. This final constellation is depicted in Fig. 4.14. In summary, we showed that the property is preserved under feedback composition if subcomponents communicate asynchronously, which is reflected in the input totality of the connector automata used for communication. Note that no additional assumptions were made about the behavior of connectors. This is in accordance with the observations by Zakinthinos and Lee, who point out that an intermediary component representing a connector "does not have to do much" [ZL96]. Thus, a connector may be regarded as "a component that simply copies its input to its output after a fixed time" [ZL96], thereby reflecting the role of connectors in MechatronicUML (cf. Section 2.3.2). In particular, the delay of messages induced by a connector does not affect the information flow property and its preservation because "the amount of delay is immaterial" [ZL96]. In addition, no assumptions have been made about the reliability of connectors in MechatronicUML, which defines whether messages may get lost in transit. Thus, whereas message delay or message loss need to be considered explicitly when verifying functional properties of components [Dzi17], we can abstract from these characteristics when taking information flow security into account.

4.8 Limitations

Continuous Signals. Our component-based security policies refer to discrete ports only and are thereby limited to the message passing between components (cf. Section 4.4). However, continuous signals transfer information as well and could be equally affected by critical information flows. Thus, in future work, we propose to enhance our policies by integrating both continuous and hybrid ports. However, such an integration would require a widened, hybrid definition of information flow security that restricts not only the discrete message passing, but also the continuous signal exchange of a component. Hybrid notions of information flow have already been considered by previous works [PK13; LMT04].

Overspecified Policies. In our approach, security policies are specified by labeling individual ports with sensitivities. Thus, as described in Section 4.4, the same sensitivity applies to all messages that are eventually passed over a certain port, regardless of how critical their information content actually is. A possible consequence is an *overspecification* of the information flow restrictions, leading to overly restrictive security policies for particular components. Hence, a promising future extension is to revise the security policies by assigning more precise sensitivities to the individual messages once they are becoming known during the behavioral design (cf. Section 2.3.3). In particular, revising the sensitivities could be used for *declassification* [SS09], enabling engineers to downgrade the criticality of messages under certain conditions (cf. Section 3.7). However, as a prerequisite of this approach, the revised sensitivities must be aligned with any information flow restrictions that have been specified at earlier stages of the development. For example, when declassifying individual messages, an equivalent declassification would need to take place at the level of flow policies, thereby ensuring consistency between the policies used in CBSE and MBSE. Furthermore, the revised sensitivities must also preserve the composability of the approach, such that no unauthorized information flows are accidentally introduced when assembling components.

Reconfiguration. Whereas the contributions of this chapter are limited to static component architectures, the MechatronicUML component model supports highly variable architectures, providing both ports and subcomponents of a component with multiplicities [Hei15, Chapter 3]. Thus, architectures give rise to various instantiations that differ in the number of instances being created for certain ports or subcomponents. Thereby, MechatronicUML supports flexible scaling of architectures, which is a main ingredient of the *microservice* architectural style [Dra+17]. On the basis of this scalability, MechatronicUML also enables structural reconfiguration of architectures as a form of self-adaptation [HBV19; Hei15, Chapter 4]. Accordingly, at runtime, a CPS may autonomously add or delete specific instances of ports or subcomponents in response to changing context conditions. With respect to security, reconfiguration poses two different challenges that need to be addressed in future works. First, a reconfiguration must not compromise the security of the architecture, as recently addressed by Khakpour et al. [KSNW19]. Hence, the proposed well-formedness rules would need to be taken into account at runtime to distinguish secure from insecure reconfigurations. Second, during operation, a CPS may face changing security situations, which require variable security restrictions to be enforced [Ben+19]. Thus, the security situation itself might be a context condition that undergoes change and must trigger adaptations of security policies. Hence, as a prerequisite, a component-based security policy would need to be adaptable itself.

Multiple Security Levels. In accordance with [Man03], our approach uses three different levels of sensitivities to represent component-based security policies. As described in Section 4.5.3, a consequence of this limitation is that, in general, multiple co-existing policies are needed in order to specify which information flows are unauthorized from different viewpoints. For reasons of clarity and comprehensibility, it might be desirable for software architects to represent these different viewpoints in a single, shared security policy that is based on an arbitrary number of sensitivity levels. We refer the reader to [WM16] for an in-depth investigation of how effectively information flow policies can be reduced from multiple levels to a fixed number of levels as in our case.

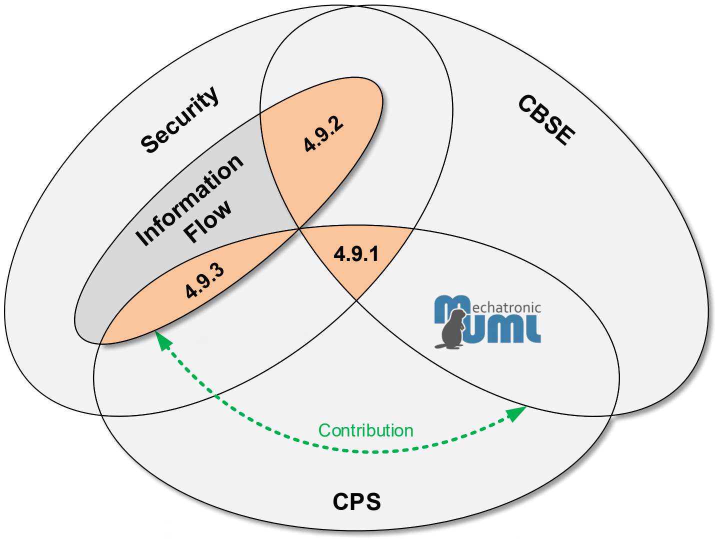


Figure 4.15: Overview of related works from different areas.

4.9 Related Work

Our contribution is an integration of composable information flow into MechatronicUML, which lies at the intersection of CBSE and CPSs. As shown in Fig. 4.15, we discuss related work from different adjacent areas. In Section 4.9.1, we survey previous approaches for CBSE of CPSs considering security in general. In Section 4.9.2, we focus more specifically on the area of information flow security and its application to CBSE. Finally, Section 4.9.3 covers the field of compositional information flow in the presence of crucial CPS characteristics.

4.9.1 Security for Component Architectures of Cyber-Physical Systems

Whereas a general overview on security in CBSE is given in [KBN14], we lay the focus on the domain of CPSs. Saadatmand et al. [SLCS12] transform unsecured component architectures for embedded real-time systems into a secured form. Similar to our work, the authors enable architects to specify security requirements by indicating security-critical data at an architectural level. On this basis, they augment the architectures by generating additional components for security mechanisms like cryptography. Whereas the resulting architectures are analyzable with respect to the real-time behavior of components, there is no dedicated analysis of the information flow or other security properties. Mohammad and Alagar [MA11] present a component-based approach for the development of embedded systems, using formal methods to verify numerous trustworthiness properties including security. By addressing real-time behavior, the authors take a crucial property of CPSs into account. Furthermore, they also focus on the preservation of trustworthiness on composition. However, compared to our work, the authors use *access control* as their underlying security model.

Borda et al. [BPKN18] present a compositional verification approach for self-adaptive CPSs. To this end, the authors extend a process algebra with means to specify the adaptation of systems. On this basis, they apply formal methods to check specifications against requirements like security, however, without focusing on data confidentiality or integrity in particular. Accordingly, the approach falls outside the area of information flow security.

Other approaches address the more general class of distributed systems and are therefore applicable to CPSs in particular. For example, Jaskolka and Villasenor [JV17] detect insecure interactions between components of distributed systems. Whereas the authors use formal methods to specify systems in terms of algebraic representations, their underlying definition of security is not based on information flow. Uzunov et al. [UFF13] propose a conceptual framework guiding software architects during the decomposition of distributed systems. Whereas the authors emphasize the need to integrate security policies into the decomposition, they only propose a general methodology, without referring to specific forms of policies. Thereby, they differ from our work, which refers to information flow policies explicitly.

Heyman et al. [HSJ09] address the refinement of formal security requirements, which are attached to the interfaces of component architectures. Due to its generality, the approach is also applicable to CPSs. On architectural decomposition, the authors check formally whether the attached requirements are preserved. Whereas the general approach resembles our work closely, the authors do not address information flow requirements. Another general approach applicable to CPSs is proposed by Zhou and Alves-Foss [ZA08]. Similar to the goal of this chapter, the authors propose a set of patterns for the preservation of security properties during the architectural decomposition. These patterns enable software architects to decompose, aggregate, or eliminate specific architectural elements such as components, without violating predefined security policies. In contrast to this approach, which imposes restrictions on the architectural decomposition itself, we aim to refine security policies without directly restricting architects in the way they decompose an architecture. Furthermore, although the authors refer to policies encompassing multiple security levels, they do not address information flow security and the preservation of corresponding policies.

In summary, none of the aforementioned related works at the intersection of CBSE and secure CPSs are based on information flow security. Hence, opposed to our contribution proposed in this chapter, these approaches are not able to provide software architects with the formal rigor and the associated security guarantees offered by the theory of information flow. In particular, none of them address its inherent challenge of ensuring composabilty.

4.9.2 Information Flow Security in Component-Based Software Engineering

Sfaxi et al. [SARL13] address the information flow security of component-based distributed systems. Similar to our work, the authors consider asynchronous communication by means of message passing between components. Moreover, they also specify information flow policies by labeling ports and establish rules that indicate whether these labels allow specific ports to be connected securely. However, the underlying component model of the approach does not address CPSs. In particular, there is no dedicated consideration of real-time behavior, which we took as a basis when demonstrating the composability of our policies in Section 4.7.

Ben Said et al. [SABB14] propose the *secBIP* framework for information flow security of component-based systems. The authors consider components with stateful behaviors, which communicate synchronously and thereby differ from our approach in the context of CPSs. Yurchenko et al. [Yur+17] use component-based information flow policies to derive security obligations that are verified by means of *theorem proving*. Unlike our contribution, the three works discussed above do not directly address the composability of information flow [Man02], which consists in enforcing global security on the basis of localized policies.

Greiner et al. [GMB17; Gre18] address this composability explicitly and thereby enable a modular security verification for component-based systems. However, opposed to our work in the scope of CPSs, the authors assume a synchronous communication between components. Sun et al. [Sun+14] propose another compositional approach for secure information flow, which involves a translation from component-based SysML models into representations that are amenable to formal verification. Unlike our work, none of the above approaches cover the refinement of global security policies into local ones during the architectural decomposition.

In contrast, Chong and van der Meyden [CM15] address the preservation of architectural information flow policies when decomposing components in a top-down fashion. Another commonality shared with our work is that the authors deduce security from a system's global structure and from local properties of its components, thereby adopting a compositional approach. Whereas the authors present a general framework without defining precisely how local properties are verified, we propose an accompanying verification technique in Chapter 5.

In summary, although the above works benefit from the formal rigor of information flow, none of them combine asynchronous message passing and real-time behavior as crucial CPS characteristics that we address in our work. In addition, the majority of approaches does neither tackle the composability challenge of information flow security, nor the refinement of policies during the architectural decomposition.

4.9.3 Compositional Information Flow Security for Cyber-Physical Systems

Wang and Yu [WY14] address the composability of information flow security for CPSs represented by petri nets. However, similar to the works discussed in Section 4.9.2, the authors do not take CPSs characteristics such as real time or message passing into account. In contrast, Li et al. [LMT17] enable compositional reasoning about information flow under the assumption that processes communicate asynchronously by message passing. Thereby, they address one of the crucial characteristics underlying our work. However, the authors do not consider the real-time behavior exhibited by CPSs. Another compositional approach is presented by Rafnsson et al. [RJB17], addressing the preservation of information flow security under composition of processes. The authors consider both asynchronous communication and time-dependent behavior of processes, thereby combining similar characteristics as our work in this chapter. However, whereas we focus on dense real time, the authors rely on a discrete notion of time, which carries a remaining security risk because "the expressiveness of dense-time models [. . .] is strictly larger than discrete-time ones" [Cas09, p. 22]. Accordingly, "there are cases in which dense time must be used since there is no discretization (no matter how small) that can faithfully emulate all dense time behaviors" [Gor+04, p. 183].

A general difference between our work and the compositional approaches discussed above is that they compose behavioral specifications, given either in terms of program-level processes or petri nets. Thereby, compared to our approach, none of the above works take the architectural design of systems into account. As a consequence, the systematic refinement of security policies during the architectural decomposition is beyond the scope of these works.

4.10 Summary

In this chapter, we addressed the well-formed refinement of component-based security policies, restricting the information flow through software components in MechatronicUML. Initially, we proposed a partial derivation of such policies from the flow policies specified by systems engineers as described in Chapter 3. On this basis, we established a set of architectural well-formedness rules, which ensure composability of the information flow restrictions imposed by our policies. Thus, when security policies are completed or otherwise refined during the architectural decomposition, the application of our rules indicates whether the intended security restrictions are enforced by the refinement or not.

To validate that our contribution is sound, we demonstrated the composability of the proposed approach. To this end, we showed that a well-formed security policy will preserve our underlying information flow property under composition, provided that the assembled components communicate asynchronously. Thereby, we reproduced a known result [ZL96; ZL95] in the scope of the real-time behavior underlying MechatronicUML.

Our contributions enable a seamless, security-preserving translation of the information flow restrictions from MBSE to CBSE. On this basis, our well-formedness rules assure software architects that their refined policies will enforce the intended security restrictions and prevent unauthorized information flows from emerging on composition of components. Thereby, we enable them to reason about the global security of a software architecture on the basis of localized security policies for the constituent components. In the upcoming Chapter 5, we will address the problem of checking the real-time behavior of components against these localized policies, which thereby act as *proof obligations*.

5

A Verification Technique for Real-Time Information Flow Security

The security policies introduced in Chapter 4 enable software engineers to impose information flow restrictions on MechatronicUML's software components. Thereby, these policies specify security requirements that must be satisfied by the implemented behavior of components. In order to *verify* that a component meets these requirements, software engineers need a corresponding verification technique that automatically detects unauthorized information flows and thereby checks whether the component's behavior adheres to its security policy.

In the scope of CPSs, a key challenge arises from the real-time behavior of components (cf. Section 2.3.3). An observer who knows about the imposed real-time restrictions may use this knowledge to draw conclusions about critical information from the observed response times. In this case, the system is compromised by a *timing channel* (cf. Section 2.4), which must be detected to verify the absence of unauthorized information flows. Therefore, in the context of MechatronicUML, a verification technique for secure information flow must be *timing-sensitive* [KWH11] by taking the specified real-time behaviour into account.

To prepare for such a timing-sensitive verification, we defined the information flow security of timed automata by means of a timed bisimulation in Section 4.7.1. Unlike language equivalence of timed automata, timed bisimulation is known to be decidable [Čer93] and can therefore be checked by existing verification techniques for real-time systems [cf. Wan04]. Such techniques are highly sophisticated because they must explore the real-valued state space of timed automata (cf. Section 2.3.3), comprising an infinite number of states. In particular, to be processed algorithmically, the state space must be converted into a finite representation. On the one hand, it is therefore desirable for software engineers to verify the information flow security of real-time systems using off-the-shelf tools and thereby benefit from the maturity and efficiency of their built-in verification algorithms. On the other hand, as described in Section 2.4.2, information flow is known to be a hyperproperty [CS10], which interrelates multiple execution traces. Therefore, it falls outside the traditional classification of safety and liveness properties by Alpern and Schneider [AS85]. Since standard verification tools for real-time systems are restricted to safety and liveness properties, engineers face the problem that these tools are not applicable out of the box. Hence, in the scope of MechatronicUML, verifying the security policies of software components by means of existing, tool-supported techniques is an unresolved problem, which we address in this chapter.

State of Research. Traditionally, information flow properties are often verified through *unwinding* [GM84; Man00b], a technique that localizes the verification to properties of individual state transitions. Another prominent technique is referred to as *self-composition* [BDR11], reducing the verification to a problem that can be solved by considering single execution traces in isolation, without interrelating multiple traces. This approach enables information flow to be checked by standard verification techniques restricted to safety and liveness properties, including model checking [DHRS11; MZ07; HWS06] or theorem proving [DHS05]. As a complementary approach, model-checking techniques have also been specially tailored to information flow [Dim+12] or even general hyperproperties [FRS15; Rab16, Chapter 5]. However, as of today, verification techniques for hyperproperties of timed automata are only in the early stages of development [HZJ19; Hei18]. As a special case of hyperproperties, the scientific literature also provides dedicated works on the information flow security of timed automata [AS19; VNN18; BCLR15; LMT10; Cas09; BT03]. Whereas these approaches provide timing-sensitive verification techniques in general, none of them apply existing, off-the-shelf tools while producing accurate verification results.

Contributions. The major contribution made in this chapter is a novel verification technique for the information flow security of real-time systems, which is based on the application of an off-the-shelf verification tool for timed automata. Thereby, we enable the detection of timing channels by means of formal, tool-supported verification. Our verification technique adopts the aforementioned idea of self-composition by checking whether a timed automaton is properly *refined* by a variant of itself, differing in the amount of critical information that is accessible. A proper refinement requires that this difference will have no effect on the processing of any observable information, which would otherwise constitute an unauthorized information flow. Thereby, we reduce the verification to a refinement check as introduced in Section 2.3.3, applying the work by Heinzemann et al. [HBDS15; Hei15, Chapter 5] to the problem domain of security. The refinement check adopts a two-step approach. First, an automaton that undergoes verification is transformed into a *test automaton* [ABL98], introducing an auxiliary location that encodes a violation of the refinement. In our case, such a violation indicates the presence of an unauthorized information flow. Second, the state space of the constructed automaton is explored to check whether the auxiliary location can ever by reached. In our case, the verified automaton is secure whenever the location is unreachable. To reason about the reachability, we apply the off-the-shelf model checker Uppaal [LLN18], which implements efficient, practice-approved verification algorithms for the real-time behavior of timed automata. Thereby, we save software engineers from the need to explore the infinite state space by implementing proprietary and error-prone algorithms from scratch. To study the accuracy of the verification results, we apply our technique to a security-oriented extension of the Common Component Modeling Example (CoCoME) [GH17].

Novelty. Along the lines of self-composition, verifying information flow properties by means of model-checking techniques is a state-of-the-art approach. In contrast, the novelty of our work is that we adopt this idea in the area of real-time systems in order to support a timing-sensitive verification. Thereby, we are the first in the field to provide software engineers with a method that enables them both to detect timing channels of timed automata and benefit from the efficiency and maturity offered by an off-the-shelf model checking tool.

Publication. Initially, we outlined the contributions of this chapter in [*Ger16]. Subsequently, the technical details underlying our work have been published in a conference paper [*GSB18]. In combination with the contributions of Chapter 4, we applied and evaluated our approach in another conference paper [*GS19].

Outline. The remainder of this chapter is structured as follows. We sketch our scientific contributions in Section 5.1, before specifying the requirements to be satisfied by our verification technique in Section 5.2. Section 5.3 gives an overview on the integration of our technique into the MechatronicUML process. Next, we introduce our general verification approach in Section 5.4, and describe the construction of the underlying automata in Section 5.5. In Section 5.6, we conduct our case study before discussing limitations of our approach in Section 5.7. Finally, we survey related work in Section 5.8 and summarize this chapter in Section 5.9.

5.1 Scientific Contributions

The following contributions are made in this chapter:

- We propose a tool-supported, timing-sensitive verification technique for the information flow security of timed automata. In particular, we reduce the verification to a *refinement check* for real-time systems, which enables us to detect timing channels by means of model-checking techniques.

- At the core of our proposed technique, we transform a timed automaton at hand into a *test automaton*, enabling us to use the off-the-shelf model checker Uppaal to verify the information flow security by checking the reachability of particular locations.

- We use a security-oriented extension of the community case study CoCoME [GH17] to evaluate the accuracy of the verification results produced by our technique.

5.2 Requirements

In the following, we specify and justify three requirements (R1–R3) that must be satisfied by our verification technique.

(R1) Timing Sensitivity: To enable the detection of timing channels, the security model underlying our verification technique must be *timing-sensitive* [KWH11]. Accordingly, we assume that observers are able to measure the instant of real time at which a component sends a message and thereby responds to specific events. Furthermore, during verification, we must take into account the amount of real time that passes during the operation of a system, which is reflected in the real-time restrictions imposed on the behavior of components. On this basis, the information flow property verified by our technique must ensure that a component will only be deemed secure if its observable response times do not depend on any critical information.

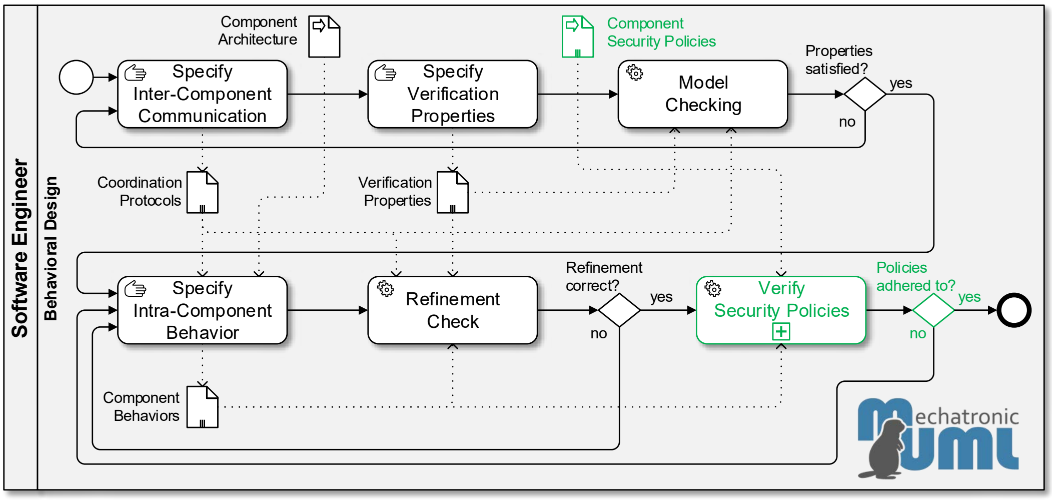


Figure 5.1: Proposed extensions to the MechatronicUML behavioral design process.

(R2) Off-The-Shelf Verification: Since exploring the infinite, real-valued state space of timed automata is tedious and error-prone, we delegate this task to an existing verification tool. Thus, instead of developing verification algorithms from scratch, our proposed technique should apply an off-the-shelf tool with built-in verification algorithms.

(R3) Accuracy: As a requirement for analysis techniques in general, the verification must detect unauthorized information flows (including timing channels) as accurately as possible. The implications of accuracy are twofold. On the one hand, all unauthorized information flows must be rejected by our technique. Accordingly, the results produced during verification must be *sound*, such that insecure systems are never classified as secure. On the other hand, authorized information flows should not be rejected by our technique. The verification results should therefore be *complete*, preventing secure systems from being classified as insecure.

5.3 Overview

This chapter complements MechatronicUML's compositional verification approach from Section 2.3.3 by adding a verification step for information flow security. Figure 5.1 highlights our extension in green, which takes place after the specified component behaviors have been verified as correct refinements of the respective coordination protocols. Once all protocols are refined correctly, the information flow through the components undergoes verification in the Verify Security Policies activity. In this step, we take into account the component security policies resulting from the architectural design (cf. Fig. 4.1 on page 59) and check whether the component behaviors adhere to these policies. In Fig. 5.1, the verification is depicted as a collapsed subprocess [OMG13], which will be detailed in the course of this chapter.

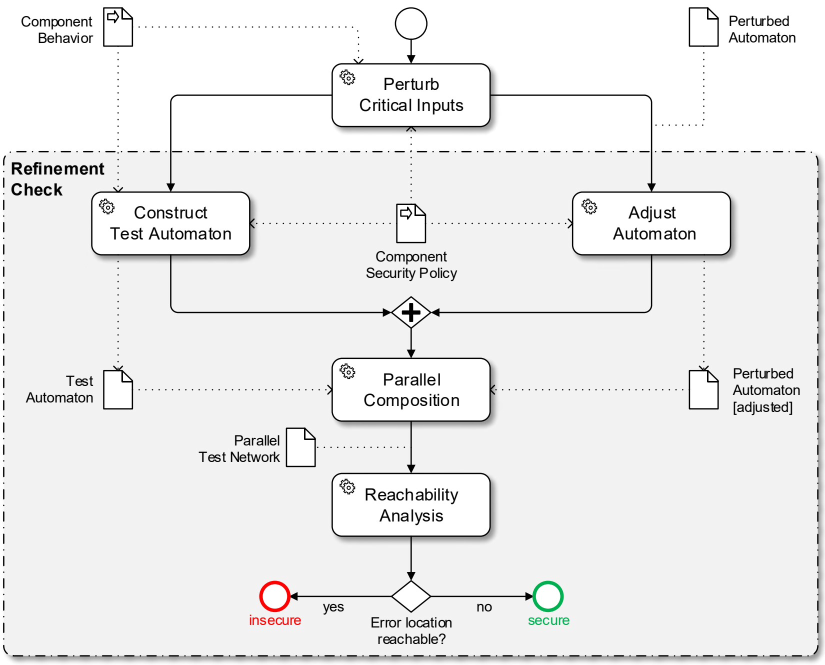


Figure 5.2: Overview of the verification approach [adapted from *GSB18; HBDS15].

If the behavior of a component does not adhere to at least one of the component's associated security policies, we require software engineers to revise the behavior by repeating the Specify Intra-Component Behavior activity and also rerun the Refinement Check. According to our extension, the behavioral design terminates once all coordination protocols have been refined correctly and all components adhere to each of their security policies.

5.4 General Verification Approach

To present a general view of our verification approach, we extend the Verify Security Policies subprocess from Fig. 5.1 and show its interior process in Fig. 5.2. As depicted, the input of the process comprises a *component behavior*, which is given in the form of a timed automaton and will be checked against a *component security policy*. The final result of the process indicates whether the component behavior is secure or insecure with respect to the given policy. In Section 4.7.1, we defined the security restrictions imposed by our policies with the help of timed bisimilarity. As described in Section 2.3.3, timed bisimulation is one of the refinement definitions underlying MechatronicUML's compositional verification approach. Therefore, we reapply the refinement check [HBDS15; Hei15, Chapter 5] to the field of information flow security. To verify the aforementioned bisimilarity, we check whether the component behavior is properly refined by a perturbed variant of itself. Thereby, our technique applies the principle of self-composition [BDR11].

Figure 5.2 illustrates our approach, which uses model transformations to reduce the verification to a problem that is amenable to off-the-shelf verification. In the initial activity Perturb Critical Inputs, the component behavior is transformed such that inputs classified as critical by the component security policy will undergo perturbation. Thereby, the activity produces a perturbed automaton as a variant of the original component behavior. In the scope of Definition 4.1 on page 73, the original component behavior represents the automaton A, whereas the perturbed automaton corresponds to the construction $\mathsf{P} \parallel \mathsf{A}$.

The perturbation enables us to verify the information flow security by adapting the refinement check for timed bisimulation [HBDS15], which ensures timing sensitivity as per requirement R1. According to this approach, the activity Construct Test Automaton transforms the component behavior into a *test automaton* [ABL98], which encodes the security policy to be verified and thereby serves as an *oracle* [How78] for the perturbed automaton. Accordingly, the perturbed automaton must not deviate from the test automaton in terms of any behavior that is considered observable by the policy. To keep track of such deviations, the test automaton introduces an auxiliary location named *error* that encodes a policy violation. Thus, the verification of the policy reduces to testing the reachability of this location during execution. To this end, the activity Adjust Automaton ensures that the perturbed automaton synchronizes with the test automaton when receiving equal inputs or sending equal outputs. If the perturbed automaton deviates from the original component behavior in terms of an observable input or output, the test automaton switches to its *error* location.

The synchronized execution of both automata is enabled by the Parallel Composition activity, composing a network of timed automata that we refer to as parallel test network. Inside this network, the unperturbed behavior is composed in parallel with its perturbed variant, as can be seen from Fig. 5.2. Therefore, this part of the process reflects the underlying principle of self-composition [BDR11]. Finally, in the Reachability Analysis activity, the information flow is verified by analyzing whether the *error* location is ever reachable during execution of the parallel test network, comprising both test automaton and perturbed automaton. Thereby, we reduce the information flow security to a safety property verified using model-checking techniques. In particular, as demanded by requirement R2, we analyze the reachability using the off-the-shelf model checker UPPAAL [LLN18] for timed automata. If the *error* location may be reached on some execution trace, then the component behavior is deemed insecure and the verified security policy is violated. In contrast, if the location is never reachable on any possible trace, the component adheres to its policy and is therefore deemed secure. In the broader context of the behavioral design (cf. Fig. 5.1), we use these verification results to decide whether a component behavior adheres to a specific component security policy or not.

5.5 Automata Construction

In this section, we go into detail about the transformations used to construct the timed automata underlying our verification approach. In Section 5.5.1, we address the perturbation of automata, before we elaborate on the construction of our test automata in Section 5.5.2. Finally, we illustrate the adjustment of the perturbed automata in Section 5.5.3.

5.5.1 Perturbed Automaton

In our initial publication [*GSB18], we relied on a notion of perturbation that corresponds to the *removal* [Man00a] of all critical inputs at all times. We achieved this perturbation by applying the restriction operator introduced in Section 2.3.3, thereby removing all edges from an automaton that are triggered by some critical input. Accordingly, the resulting perturbed automaton is considered *secure by definition* [*GSB18] because it is never able to receive any critical inputs at all. This form of perturbation reflects the notion of *purge*, which was also used in the original definition of noninterference [GM82] and provides the basis for well-established information flow properties such as *generalized noninference* [McL94]. However, Definition 4.1 on page 73 requires us to use a different form of perturbation, in which critical inputs are not completely removed but situationally *deleted* [Man00a] from the execution in piecemeal fashion. Thus, instead of restricting all edges that are triggered by critical inputs, we need to disable the individual edges independently and temporarily.

To enable this situative deletion, we extend a perturbed automaton with a dedicated Boolean variable for each critical input it may receive.[1] These variables are used to *conditionalize* [*GS19] the receiving of inputs. Thus, to each edge triggered by a critical input, we attach a condition referring to the variable associated with that input. These conditions correspond to additional *guards* (cf. Section 2.3.3) for the respective edges. Thereby, the edge is only enabled if the current value of the associated variable is *true*. In contrast, if the current value is *false*, the concerning input undergoes perturbation and is regarded as deleted from the execution. In Fig. 5.3, we perturb the component behavior of the Engine Control Unit known from Fig. 2.10 on page 28. In particular, we depict the perturbation of the adjust input, which we assume to be classified as critical by the associated component security policy. Therefore, we use the Boolean variable isAdjustEnabled to conditionalize the receiving of the input. As shown in green, we also add a corresponding guard to the edge triggered by adjust?, which is therefore able to fire only if the associated variable is *true*. In contrast, setting the variable to *false* will disable the edge and prevent the input from being received.

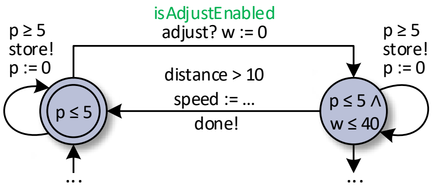


Figure 5.3: Example perturbation.

Using these variables, individual inputs may be situationally deleted from the execution by disabling the corresponding edges on a temporary basis. According to Definition 4.1 on page 73, this situational deletion is controlled by a perturbator automaton P that may be chosen freely. At each point in time, a concrete perturbator P may or may not provide a perturbed automaton A with a particular critical input. To account for the free choice of the perturbator, we ensure that the values of the aforementioned variables are nondeterministically toggled between *true* and *false* during execution. Thereby, edges triggered by critical inputs will be disabled and re-enabled on a rotating basis at arbitrary times. Accordingly, the perturbed behavior covers all possible ways to delete critical inputs.

[1] In Definition 2.7 on page 24, the discrete state space of a timed automaton is restricted to its locations. However, additional variables of data types like Boolean are supported by Uppaal as our underlying verification tool.

5.5.2 Test Automaton

Since our definition of information flow security given in Section 4.7.1 relies on the notion of bisimilarity, we base our verification technique on the test automata for timed bisimulation as proposed by Heinzemann et al. [HBDS15; Hei15, Chapter 5]. However, instead of reusing this construction as is, we need to account for the fact that the bisimilarity in Definition 4.1 on page 73 is limited to observable messages from the set V, whereas non-observable messages from $N \cup C$ are regarded as *hidden*. According to the hiding operator described in Section 2.3.3, edges are replaced by τ transitions if they are labeled with a symbol to be hidden. Therefore, hiding messages by means of this operator gives rise to potential nondeterminism because distinct edges can no longer be distinguished by their hidden symbols. As a consequence, the use of the hiding operator is incompatible with the technique by Heinzemann et al., which requires the transition function of an automaton to be deterministic [HBDS15, p. 259]. The authors thereby assume that, given a certain input, the current location of the automaton will never have two outgoing edges that can both fire at the same time. To account for this restriction, we refrain from hiding non-observable communication by introducing τ transitions. Instead, we realize the hiding of messages by adapting the construction of the test automata. To that end, we relax the restrictions imposed by the timed bisimulation such that messages may be treated as hidden. We thereby enable specific messages to be used for corrections (cf. Section 2.4.2) without indicating a *false positive* information flow.

We show our adapted construction schema for test automata in Fig. 5.4. Deviating from Fig. 2.8 on page 23, we label locations with names, whereas invariants are displayed outside of their locations. The test automaton T_A is constructed from each edge $\mathsf{S} \rightarrow \mathsf{S}'$ of the unperturbed automaton A. Such an edge is labeled with a guard ρ, a symbol μ, and a set λ of clock resets. The corresponding locations S and S' are transformed into S_TA and S'_TA. As depicted, Heinzemann et al. introduce an auxiliary location named Err that represents a violation of the refinement. In our case, the location indicates that the observable communication of the perturbed and unperturbed automata deviate from each other, constituting an unauthorized information flow. Consequently, we consider communication of the perturbed automaton *secure* if it is allowed by the unperturbed automaton and *insecure* if not. To verify the bisimilarity as proposed by Heinzemann et al., the test automaton must meet the following three responsibilities: ❶ accept the occurrence of secure communication, ❷ reject the occurrence of insecure communication, and ❸ reject the non-occurrence of secure communication. Below, we consider each of these cases in detail, before giving an example construction.

Accepting the Occurrence of Secure Communication

Communication of the perturbed automaton does not constitute an unauthorized information flow if it is also possible for the unperturbed automaton. Accordingly, such communication is secure and must be accepted by the test automaton. To accept communication, Heinzemann et al. introduce the edge $\mathsf{S}_\mathsf{TA} \rightarrow \mathsf{S}'_\mathsf{TA}$ labeled with ❶ in Fig. 5.4. This edge can fire whenever firing $\mathsf{S} \rightarrow \mathsf{S}'$ is possible in A. To this end, $\mathsf{S}_\mathsf{TA} \rightarrow \mathsf{S}'_\mathsf{TA}$ carries a symbol $\overline{\mu}$, which complements the original symbol μ of $\mathsf{S} \rightarrow \mathsf{S}'$. Following Heinzemann et al., the test automaton uses

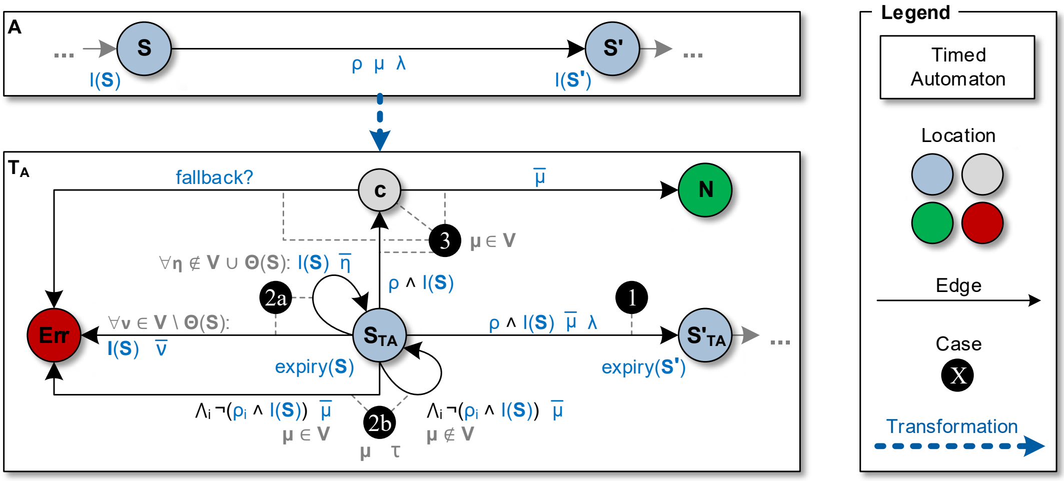


Figure 5.4: Schema for the construction of test automata [adapted from *GSB18; HBDS15].

complementary symbols as a general rule and is thereby able to synchronize its execution with the perturbed automaton when sending or receiving equal messages. In addition to the symbol $\overline{\mu}$, the edge $\mathsf{S_{TA}} \to \mathsf{S'_{TA}}$ is also labeled with a guard $\rho \wedge \mathsf{I(S)}$, which combines the original guard of $\mathsf{S} \to \mathsf{S'}$ and the invariant of the source location S. Thereby, Heinzemann et al. prepare for the case that I(S) cannot be carried over to $\mathsf{S_{TA}}$ under certain circumstances described under case ❷. If not carried over, integrating the invariant into the guard ensures that $\mathsf{S_{TA}} \to \mathsf{S'_{TA}}$ can only fire when S may actually be active. Finally, the set λ of clock resets is carried over to $\mathsf{S_{TA}} \to \mathsf{S'_{TA}}$ as is. By progressing from $\mathsf{S_{TA}}$ to $\mathsf{S'_{TA}}$, the test automaton reflects the regular communication behavior of the unperturbed automaton, independent from the sensitivity of the message μ being communicated. Thus, in this case, no specific adaptations of the original construction are needed to hide non-observable messages.

Rejecting the Occurrence of Insecure Communication

Observable communication of the perturbed automaton that is not possible for the unperturbed automaton violates the bisimilarity in Definition 4.1 on page 73 because $(P \parallel A) \,/\, (N \cup C) \not\approx A \,/\, (N \cup C)$. In this case, disabling critical inputs in $P \parallel A$ gives rise to communication that cannot be observed when these inputs are enabled as in A. Such an observation can therefore be made in the absence, but not in the presence of certain critical inputs. Accordingly, as described in Section 4.7.1, such communication can no longer be observed when inserting critical inputs into the execution. This deviation constitutes an unauthorized information flow, and the test automaton must reject the occurrence of such communication as insecure by switching to its Err location. In accordance with the original construction, we must distinguish between two cases of insecure communication: (2a) observable messages that are not allowed in a particular location and (2b) observable messages that are untimely because they violate the time restrictions that apply in a particular location.

In case 2a, a message is only allowed in S if the corresponding symbol is included in the set $\Theta(\mathsf{S})$ of all symbols labeling an outgoing edge. To track communication that is not allowed, Heinzemann et al. create one edge $\mathsf{S}_{\mathsf{TA}} \rightarrow \mathsf{Err}$ for each message that is not in $\Theta(\mathsf{S})$. In our case, we create these edges only for observable messages $\nu \in \mathsf{V} \backslash \Theta(\mathsf{S})$. We label the created edges with $\mathsf{I}(\mathsf{S})$ as a guard to prevent them from firing outside the period of time in which S can be active. Each of the edges carries a complementary symbol $\overline{\nu}$. Thus, whenever the perturbed automaton sends or receives an observable message ν that is not allowed in S, the synchronization forces the test automaton to enter the Err location. Note that, whenever S is a committed location representing an atomic sequence of state transitions (cf. Section 2.3.3), the edges $\mathsf{S}_{\mathsf{TA}} \rightarrow \mathsf{Err}$ are omissible because the sequence is not assumed to be interruptible.

Unlike the original construction, a non-observable message does not represent an information flow because it is considered hidden. Nevertheless, since handshake communication is mandatory (cf. Section 2.3.3), we must explicitly enable the perturbed automaton to send or receive such messages, even if they are not allowed by the unperturbed automaton. Thus, for each message $\eta \notin \mathsf{V} \cup \Theta(\mathsf{S})$, which is neither observable nor allowed, we create a loop $\mathsf{S}_{\mathsf{TA}} \rightarrow \mathsf{S}_{\mathsf{TA}}$. We thereby adopt an existing approach by Bossi et al. [BFPR02] towards verifying information flow security through bisimilarity. As for observable communication, each loop is labeled with a guard $\mathsf{I}(\mathsf{S})$ and a symbol $\overline{\eta}$. By firing these loops, the unperturbed automaton will ignore non-observable messages that are not allowed. Such messages can therefore be inserted by the perturbed automaton to *correct* its execution.

In case 2b, Heinzemann et al. consider communication as untimely if it occurs outside the period of time in which it is allowed to occur. This period is defined by the time restrictions that apply for a message $\mu \neq \tau$ in location S. In particular, μ can only be sent or received in S if both the guard ρ_{i} of any outgoing edge labeled with μ and the source invariant $\mathsf{I}(\mathsf{S})$ hold. Thus, the time at which such an edge cannot fire is defined by $\neg(\rho_{\mathsf{i}} \wedge \mathsf{I}(\mathsf{S}))$. A message is therefore untimely if sent or received at a time when none of the corresponding edges can fire. This time amounts to the intersection $\bigwedge_{\mathsf{i}} \neg(\rho_{\mathsf{i}} \wedge \mathsf{I}(\mathsf{S}))$. As distinct from Heinzemann et al., we only regard untimely communication as an information flow if the message is observable. Thus, if $\mu \in \mathsf{V}$, we create an edge $\mathsf{S}_{\mathsf{TA}} \rightarrow \mathsf{Err}$ that is labeled with the above intersection as a guard and a complementary symbol $\overline{\mu}$. If the perturbed automaton sends or receives $\mu \in \mathsf{V}$ untimely, the synchronization forces the test automaton to indicate the violation by switching to Err. In contrast to the original construction, if $\mu \notin \mathsf{V}$, then untimeliness does not constitute an information flow. In this case, following Bossi et al. [BFPR02], we create a loop $\mathsf{S}_{\mathsf{TA}} \rightarrow \mathsf{S}_{\mathsf{TA}}$ with the same guard as $\mathsf{S}_{\mathsf{TA}} \rightarrow \mathsf{Err}$. Thus, instead of switching to Err, the adjusted test automaton will ignore non-observable messages that are sent or received untimely.

Note that the above intersection also takes into account that μ occurs after the invariant $\mathsf{I}(\mathsf{S})$ has expired. To detect this untimeliness, S_{TA} must remain active until after $\mathsf{I}(\mathsf{S})$. Therefore, Heinzemann et al. do not carry over invariants to the test automaton [HBDS15, p. 272]. Since our approach treats certain messages as non-observable, invariants can be carried over if they do not prevent the untimeliness of observable messages from being detected. Since invariants enforce progress during execution, we thereby ensure that possible corrections are made as required. To decide whether $\mathsf{I}(\mathsf{S})$ can be carried over, we use a function expiry to provide S_{TA} with an invariant. In Eq. (5.1), we distinguish between three cases:

$$expiry(S) = \begin{cases} I(S) & \forall S \xrightarrow{\rho\ \sigma\ \lambda} S^* : \sigma \notin V \\ I(S) & \exists S \xrightarrow{\rho\ \sigma\ \lambda} S^* : \sigma \notin V \land \rho = true \\ true & \text{else} \end{cases} \tag{5.1}$$

First, if none of the outgoing edges $\mathsf{S} \xrightarrow{\rho\ \sigma\ \lambda} \mathsf{S}^*$ are labeled with an observable symbol, then no untimely communication is possible in $\mathsf{S_{TA}}$ at all. Thus, we can safely assign the invariant $\mathsf{I(S)}$ to $\mathsf{S_{TA}}$. Second, even if observable communication is possible in S, there are still cases in which we may safely assume that untimely communication will never occur after $\mathsf{I(S)}$ because $\mathsf{S_{TA}}$ has already been left at that time. We can make this assumption if at least one outgoing edge of S can fire unconditionally and is labeled with a non-observable symbol. If so, such an edge will always be used to exit $\mathsf{S_{TA}}$ before $\mathsf{I(S)}$ has expired. Thus, we carry over the invariant if there exists an outgoing edge $\mathsf{S} \xrightarrow{\rho\ \sigma\ \lambda} \mathsf{S}^*$ with a symbol $\sigma \notin \mathsf{V}$ and an unconditional guard of $\rho = \mathsf{true}$. Third, if none of the above conditions apply, untimely communication after $\mathsf{I(S)}$ may occur in $\mathsf{S_{TA}}$. In this case, we follow the original approach by Heinzemann et al. and assign a non-expiring invariant of $true$, which will never force $\mathsf{S_{TA}}$ to be left.

Rejecting the Non-Occurrence of Secure Communication

Conversely, the bisimilarity in Definition 4.1 on page 73 may also be violated because any observable communication of the unperturbed automaton is not possible for the perturbed automaton, such that $A \,/\, (N \cup C) \not\lesssim (P \parallel A) \,/\, (N \cup C)$. In this case, enabling critical inputs in A gives rise to communication that cannot be observed when these inputs are disabled as in $P \parallel A$. Such an observation can therefore be made in the presence, but not in the absence of certain critical inputs. This deviation indicates that some communication can no longer be observed when critical inputs are deleted from the execution (cf. Section 4.7.1). Thus, whenever the communication does not occur although it is considered secure, the test automaton must switch to its Err location and thereby indicate an unauthorized information flow. To handle this case, Heinzemann et al. introduce the constructs labeled with ❸ in Fig. 5.4. We adopt this approach only for observable communication of $\mu \in \mathsf{V}$. Accordingly, we add an auxiliary location that is reachable from $\mathsf{S_{TA}}$ at any point in time when $\mathsf{S} \to \mathsf{S}'$ can fire. To ensure this timing, the edge to the auxiliary location is labeled with the guard $\rho \land \mathsf{I(S)}$, intersecting the original guard of $\mathsf{S} \to \mathsf{S}'$ and the invariant of S. Since the edge represents a τ transition, it may reach the auxiliary location without synchronization. In our approach, which is based on the UPPAAL model checker, the auxiliary location is *committed* (cf. Section 2.3.3). Hence, it must be left instantly upon entry by firing one of its outgoing edges, which are used to decide between the occurrence or non-occurrence of the communication.

First, if the auxiliary location is entered at a time when the message μ can be sent or received by the perturbed automaton, the edge labeled with $\overline{\mu}$ can fire and forces the test automaton to enter the N location. According to Heinzemann et al., this location represents a neutral state of the verification. In our case, it indicates that the communication has occurred securely. Second, the edge labeled with fallback? represents the non-occurrence of the communication. We introduce fallback as an auxiliary channel that is always ready to synchronize, i.e., a

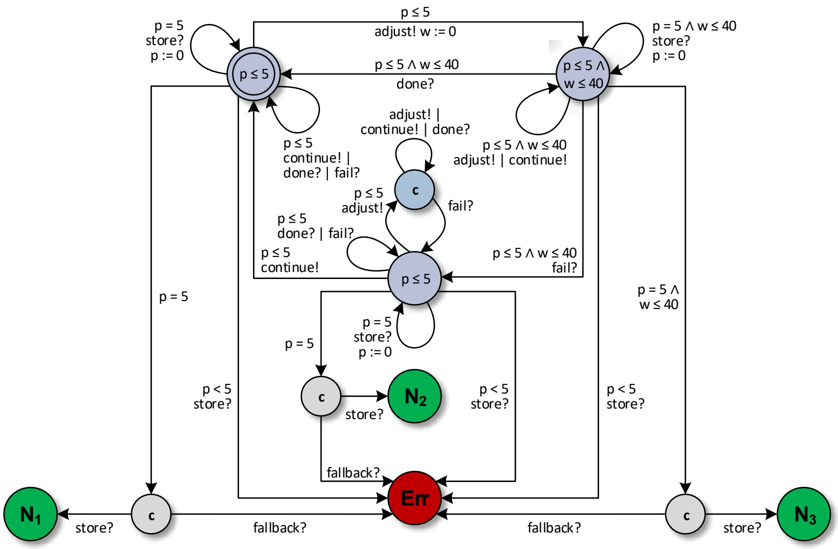


Figure 5.5: Test automaton constructed from the behavior of the Engine Control Unit.

complementary fallback! synchronization is assumed to be provided at any time. Whenever the auxiliary location is entered at a time when the message μ can *not* be sent or received by the perturbed automaton, the test automaton is forced to synchronize over the fallback channel and enter the Err location. We ensure this behavior by giving fallback the lowest priority among all synchronization channels, thereby making use of channel priorities as a feature of Uppaal. Accordingly, any predefined synchronization channel takes precedence over fallback and ensures that Err is only reachable when the message μ cannot occur at some time during the interval in which it should occur.

Example Construction

In the following, we illustrate the construction of test automata using the example of the Engine Control Unit, as previously shown in Fig. 2.10 on page 28. Since hybrid ports are left out of consideration (cf. Section 4.4.2), we restrict ourselves to the message passing over the discrete storage and infotain ports. We assume that the component security policy being verified classifies the infotain port as critical and the storage port as observable. Accordingly, store is the only observable message and must not depend on the adjust and continue messages, which represent critical inputs. Both done and fail are critical outputs used for corrections (cf. Table 4.1 on page 60).

Figure 5.5 shows the test automaton constructed from the component behavior of the Engine Control Unit. For reasons of clarity and comprehensibility, we use multiple neutral locations N_{1-3}. Furthermore, multiple loops that differ only in their synchronizations are merged into a single loop with alternative synchronizations. Finally, we also omit any edges that can never fire because their guards do not intersect with the invariant of their source locations.

As can be seen, there are six edges reaching the Err location to indicate an unauthorized information flow. On the one hand, according to case case 2b, each of the three regular locations connects to Err in order to indicate that the store message occurs untimely when $\mathsf{p} \neq 5$. However, since the clock p can never exceed the value of 5 according to the invariants carried over from the original automaton, the corresponding edges to Err are restricted by a guard of $\mathsf{p} < 5$. On the other hand, as demanded by case ❸, each of the three regular locations also connects to a committed auxiliary location in order to indicate the non-occurrence of store at $\mathsf{p} = 5$. Each of these auxiliary locations is used to activate either one of the neutral locations N_{1-3}, or the Err location by synchronizing over the fallback channel. Note that, since store is allowed to occur in each of the three regular locations, no edges to Err are created for case 2a. This even applies to the pre-existing committed location depicted in blue. Although it does not allow store to occur, it encodes an atomic sequence of state transitions, which is not assumed to be interruptible by any communication that is not allowed (cf. Section 5.5.2).

5.5.3 Adjusted Automaton

In the scope of the refinement check [HBDS15], an automaton is adjusted to ensure the correct verification of a particular refinement definition by the test automaton. Following this approach, we show our schema for the adjustment of automata in Fig. 5.6. According to the original adjustment, each edge $\mathsf{S} \rightarrow \mathsf{S}'$ inside the perturbed automaton A_p is basically carried over to the adjusted automaton $\mathsf{A}_{\mathsf{adj}}$. As depicted, the symbol μ and the clock resets λ are both left unchanged. Since the test automaton uses complementary symbols $\overline{\mu}$ as described in Section 5.5.2, Heinzemann et al. thereby ensure synchronization between both automata when sending or receiving equal messages during parallel composition. However, the adjustment requires a detailed consideration of invariants. If carried over to the adjusted automaton, invariants could prevent the test automaton from detecting the non-occurrence of communication [HBDS15, p. 276]. Whereas Heinzemann et al. do not carry over invariants to the adjusted automaton at all, our approach considers certain messages non-observable. This allows us to carry over invariants as long as they do not affect any observable communication. Thus, when transforming A_p into $\mathsf{A}_{\mathsf{adj}}$, we carry over the invariant $\mathsf{I}(\mathsf{S})$ provided that all messages sent or received in the location S are non-observable, i.e., $\Theta(\mathsf{S}) \cap \mathsf{V} = \varnothing$. If so, carrying over the invariant will never prevent the non-occurrence of communication from being checked because case ❸ in Fig. 5.4 only applies if $\mu \in \mathsf{V}$. Nevertheless, to prepare for cases in which the invariant is not carried over, we follow the original approach by Heinzemann et al. and prevent edges from firing outside of the time in which their source location may be active. Thus, by analogy with Section 5.5.2, we integrate the invariant $\mathsf{I}(\mathsf{S})$ into the guard of each outgoing edge $\mathsf{S} \rightarrow \mathsf{S}'$, resulting in the guard $\rho \wedge \mathsf{I}(\mathsf{S})$.

As distinct from the original adjustment, we also need to ensure that non-observable communication is properly hidden during verification. Accordingly, the amount of non-observable messages communicated by the perturbed and unperturbed automata is allowed to differ arbitrarily. To this end, the loops introduced for case ❷ in Section 5.5.2 already enable the adjusted automaton to send or receive non-observable messages whenever the test automaton is unable to. Conversely, we also need to enable the test automaton to send or

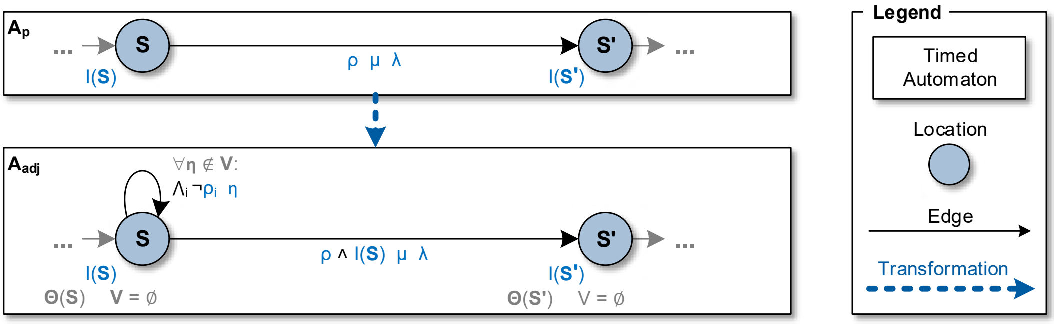


Figure 5.6: Schema for the adjustment of automata.

receive such non-observable messages whenever the adjusted automaton is unable to, e.g., because the message is a critical input that undergoes perturbation. Therefore, we once more apply the concept by Bossi et al. [BFPR02] and enhance the adjusted automaton with loops to ignore non-observable communication of the test automaton. As depicted in Fig. 5.6, we create a loop S → S for each non-observable symbol $\eta \notin V$. The associated guard ensures that a loop fires only at times when none of the outgoing edges of S allow the perturbed automaton to send or receive the same message. The time in which such an edge cannot fire corresponds to the negation $\neg\rho_i$ of its guard. On this basis, the time frame in which the perturbed automaton is unable to sent or receive η amounts to the intersection $\bigwedge_i \neg\rho_i$ over all outgoing edges labeled with η. Using this intersection as a guard ensures that a loop can only fire when the perturbed automaton is not regularly able to send or receive the corresponding message. Thereby, the message is ignored without altering the state of the automaton.

Figure 5.7 exemplifies the adjustment using the example of the Engine Control Unit from Fig. 2.10 on page 28. Since the adjustment takes the perturbed automaton as a basis, the guards inside the depicted automaton also refer to the conditions isAdjustEnabled and isContinueEnabled, which have been added to perturb the receiving of the respective critical inputs (cf. Section 5.5.1). As can be seen, loops have been introduced for the non-observable messages adjust, continue, done, and fail. For the sake of clarity, we merge multiple loops if they differ only in their synchronizations. The guards of these loops enable the adjusted automaton to send or receive the associated messages whenever the perturbed automaton is not regularly able to do so. A loop is therefore only created if the sending or receiving of a message is conditional, i.e., if the corresponding edge in A_p is equipped with a guard. Conversely, if a message can be sent or received unconditionally, no loop must be produced. Since the observable store message may be sent in any of the three regular locations, none of the invariants of the perturbed automaton can be carried over to the adjusted automaton. Instead, in Fig. 5.7, the guards of all edges have been intersected with the corresponding source invariants as described above. The situation is different with the committed location of the perturbed automaton because it does not allow store to be sent (cf. Fig. 2.10 on page 28). Therefore, as shown in Fig. 5.7, it is possible to preserve the committed characteristics including the implicit invariant, which forces the location to be left immediately upon entry.

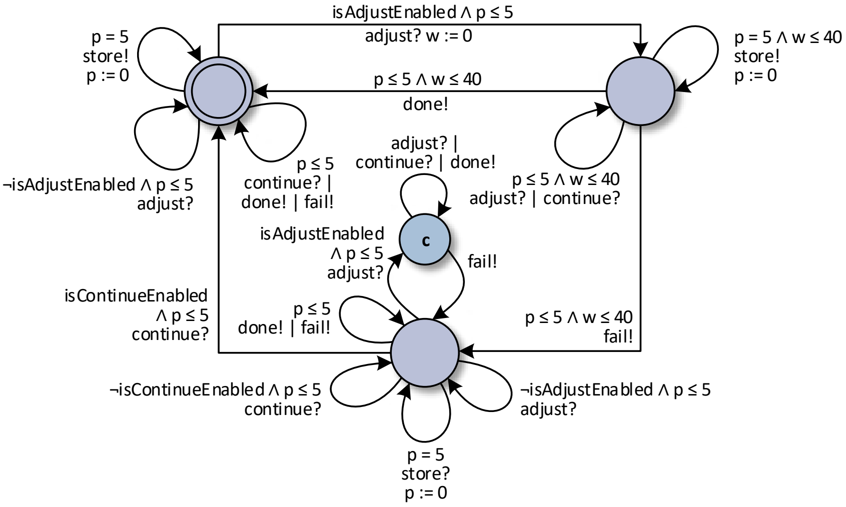


Figure 5.7: Adjusted automaton of the Engine Control Unit.

According to the adjusted automaton depicted in Fig. 5.7, the Err location of the test automaton from Fig. 5.5 is never reachable during execution of the parallel test network. This confirms our expectation that the behavior of the Engine Control Unit adheres to the component security policy encoded by the test automaton. According to this result, the sending of the observable fail message is in no way affected by the receiving of the critical adjust and continue messages, even if the adjusted automaton is arbitrarily perturbed by manipulating the values of isAdjustEnabled and isContinueEnabled.

5.6 Case Study

We evaluate the proposed verification technique by means of a case study originally conducted in [*GS19]. In the context of the requirements specified in Section 5.2, the timing sensitivity (R1) is a functional attribute, which we addressed by reducing the verification to a refinement check for real-time systems. Furthermore, the off-the-shelf verification (R2) is ensured because our technique is based on the application of the Uppaal model checker. In our case study, we therefore evaluate the accuracy (R3) of the proposed technique. In particular, we aim to answer the following research question (RQ):

RQ: *How accurate are the verification results that software engineers obtain when applying our verification technique?*

In the following, we rely on the guidelines by Runeson and Höst [RH09] to report this study. Thus, to answer our RQ, we first justify the selection of the studied case in Section 5.6.1. Subsequently, we describe the data collection in Section 5.6.2 and the analysis of the collected data in Section 5.6.3. Next, Section 5.6.4 presents our obtained results. In Section 5.6.5, we discuss the validity of our findings.

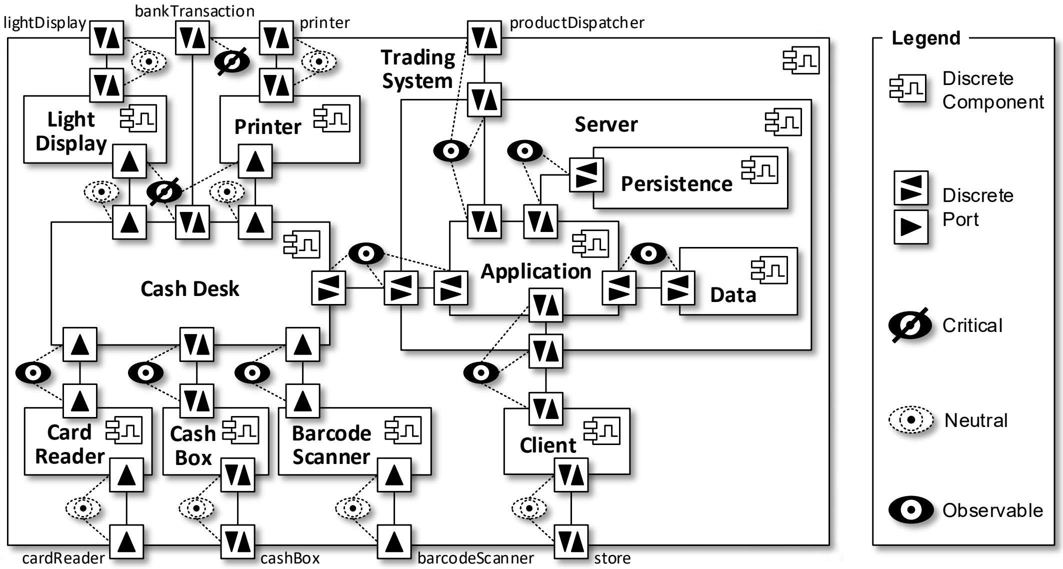


Figure 5.8: Example security policy in the context of CoCoME [adapted from *GS19].

5.6.1 Case Selection

The selected case for our study is *CoCoME*, a community case study for the design of component-based systems [Her+08]. The study is concerned with a trading system that is used by customers (e.g., of supermarkets) to purchase products. Due to CoCoME's focus on CBSE, it matches the MechatronicUML design method and its underlying component-based principles, according to which the system behavior emerges from the composition of multiple components. Due to the composability of our approach in this thesis (cf. Section 4.7), we apply the proposed verification technique to the individual components inside CoCoME's architecture. In particular, we select CoCoME due to the existence of a security-oriented extension [GH17], providing a collection of predefined information flow requirements that we use as a benchmark for our study.

In Fig. 5.8, we illustrate the component architecture of CoCoME. As depicted, the Trading System comprises a Cash Desk and a Server component. The Cash Desk communicates with a Barcode Scanner used to register the purchased products, as well as a Cash Box and a Card Reader for cash or credit card payment. Furthermore, a Light Display and a Printer are used to give information to customers, whereas the Cash Desk also communicates with a bank to process cashless payments. The Server stores the inventory data. To this end, it comprises a Persistence component that provides access to the Data storage. The server runs an Application software, which enables the inventory data to be retrieved or manipulated either by the Cash Desk during the sales process, or by a store manager in the role of a Client. Whenever the stock is running low, the Application is also responsible for reordering specific products from another store automatically.

5.6.2 Data Collection

To conduct our study, we recreate the architecture of CoCoME on the basis of the MechatronicUML component model (cf. Section 2.3.2). To this end, we carry over the set of components from [GH17, p. 19], resulting in the architecture depicted in Fig. 5.8. The interaction between CoCoME's components is based on dedicated *interfaces*, which are provided by particular components and required by other components. In MechatronicUML, we represent CoCoME's interfaces by means of ports connecting the respective components. In particular, a provided interface corresponds to a discrete *in* port, whereas a required interface translates into a discrete *out* port. By restricting ourselves to the use of discrete ports, we account for the fact that the implementation of components in CoCoME is based on the *procedural programming* paradigm. Accordingly, an interface comprises a set of *services*, each one reflecting the signature of an operation that can be called by the requiring component to invoke a behavior of the providing component. If at least one service of an interface involves a return value, we account for this two-way communication by converting the respective ports to *in/out* ports. On the basis of the resulting architecture, we regard each service call as a message passed from the requiring to the providing component, and the corresponding return value (if any) as a message passed back in the reverse direction from the providing to the requiring component. By using discrete ports only, we abstract from the physical interaction of the system with its environment through hardware components such as Barcode Scanner or Printer. However, we thereby account for the fact that CoCoME follows a procedural style of programming, whereas our approach described in this thesis is equally restricted to the message passing over discrete ports (cf. Section 4.4.2).

Security Policies. On the basis of the component architecture, we take into account the security requirements for CoCoME documented in [GH17, Section 4]. These requirements protect the confidentiality of the information exchanged between specific actors (e.g., customers, cashiers, bankers, or store managers) across the interfaces of components. Thereby, the given requirements restrict the information flow through the system. For each service and each piece of information exchanged over that service, the requirements indicate to which actors the information is allowed to be disclosed. In this context, individual pieces of information correspond to the parameter values or return values of a service call. In our study, we address the confidentiality of information entered through the external interfaces, which correspond to the eight external ports of the Trading System depicted in Fig. 5.8. For each external port, we specify a dedicated component security policy that treats the port and any information entered through the corresponding interface as critical. For each policy, we classify the residual external ports depending on whether the corresponding interfaces are authorized to disclose the critical information. To this end, we take into account the actors that interact with the system over a specific port. From the given security requirements, we extract whether the critical information is allowed to be disclosed to these actors or not. If all critical pieces of information may be disclosed to all of the actors that interact over a certain port, then this port is classified as neutral. However, if any critical piece of information must not be disclosed to some actor, then the port over which this actor interacts is classified as observable. The result is a set of eight component security policies for the Trading System.

Table 5.1: Security policies and verification results [adapted from *GS19].

Security Policies of External Ports									Verification Results of Components			
light Display	**bank Trans.**	**printer**	**product Disp.**	**store**	**barcode Scanner**	**cash Box**	**card Reader**	**Expected Result**	**Light Display**	**Printer**	**Cash Desk**	**App.**
⊘	👁	◎	👁	👁	◎	◎	◎	secure	✓	✓	✓	✓
◎	⊘	◎	👁	◎	◎	◎	◎	insecure	✓	✓	✗	✓
◎	👁	⊘	👁	👁	◎	◎	◎	secure	✓	✓	✓	✓
👁	👁	👁	⊘	◎	👁	👁	👁	secure	✓	✓	✓	✓
👁	👁	👁	👁	⊘	👁	👁	👁	insecure	✓	✓	✓	✗
◎	👁	◎	👁	◎	⊘	◎	◎	insecure	✓	✓	✗	N/A
◎	👁	◎	👁	◎	◎	⊘	◎	insecure	✓	✓	✗	N/A
👁	◎	👁	👁	👁	👁	👁	⊘	insecure	✓	✓	✗	✓

We summarize these policies in Table 5.1, where the columns on the left represent the external ports of CoCoME's architecture. Each line reflects the security policy of one specific port, which is classified as critical and highlighted in gray. Subsequently, we refine each of the specified policies manually by classifying the internal ports of the architecture as well. We thereby obtain security policies for the subcomponents of the Trading System, which are all well-formed according to the rules established in Section 4.6. As an example, Fig. 5.8 illustrates the refined policy for the bankTransaction port, which must not leak information to the productDispatcher port. We visualize the residual policies in Appendix A.

Component Behavior. On the basis of CoCoME's use cases described in [Her+08], we use Uppaal timed automata to model the stateful behavior of the bottom-level subcomponents inside the architecture. From the use cases, we extract causal dependencies between service calls. We encode such calls and the corresponding return values (if any) by outputs and inputs attached to specific edges. Since test automata are limited to deterministic systems [JLS00, p. 27], we resolve nondeterminism by providing particular edges with auxiliary synchronizations. We also use clock constraints to account for timeouts, thereby enabling a component to react to situations in which an input from another component is not received within a fictitious time limit. Note that, unlike the original definition of the CoCoME case, we abstract from the detailed data processing of components. In particular, our models do not represent the individual parameters of service calls or return values, and the way in which these parameters are determined or processed by components. Furthermore, we restrict the behavioral modeling to Light Display, Printer, Cash Desk, and Application as the only components that are effectively restricted by at least one of their refined security policies. Since the residual components never combine critical and observable ports in their policies, they are secure by definition and do not require verification at all.

Manual Assessment. Subsequently, we manually assess the security of the specified component behavior with respect to the individual policies. To that end, we use our expert opinion to check the behavior of the components for potential information flows. We thereby specify whether a certain policy is expected to be violated or adhered to by the respective component. According to the composability of our approach (cf. Section 4.7), the specified expectations also assess the security of the overall component architecture: if at least one component is expected to violate its security policy, then the architecture is deemed insecure. Conversely, if we expect all components to adhere to their policies, the architecture is deemed secure. Using these expert opinions, we obtain expected results for the automated verification. Table 5.1 indicates these expected results for each individual security policy. In particular, the architecture is expected to be secure with respect to the policies for the lightDisplay, printer, and productDispatcher ports, and is deemed insecure for the policies of the residual ports.

Automated Verification. Finally, to assess the accuracy of our technique with respect to the expected results, we carry out the verification procedure as described in Section 5.3. To construct the underlying automata as illustrated in Section 5.5, we use model transformations operating on dedicated metamodels for timed automata. We developed these metamodels as part of a publication that enables MDE on the basis of Uppaal [*Sch+17]. Building upon the transformation results, we use Uppaal to check the behaviors of the four aforementioned components against the eight refined security policies. For each component and each of its associated policies, we thereby obtain a corresponding verification result that we use to analyze the accuracy of our technique.

5.6.3 Analysis

To measure the accuracy of our technique, we analyze the collected data by comparing the obtained verification results against the expected results, which have been specified manually. Accordingly, evidence for the measured accuracy is provided by our expert opinion. If the architecture is deemed secure with respect to a policy, this expectation is satisfied if the verification results indicate that all four components adhere to their refined policies. Conversely, if deemed insecure for a certain policy, the architecture satisfies this expectation if the verification detects at least one component that violates its refined policy. On this basis, the measured accuracy amounts to the number of policies for which the verification results satisfy our expectations.

5.6.4 Results

We present the obtained verification results on the right-hand side of Table 5.1. For each of the four components, a result of ✓ indicates that the component behavior adheres to the respective security policy. In contrast, a result of ✗ represents the detection of a policy violation. According to these results, the Light Display and Printer components adhere to all of the eight refined policies. In contrast, both Cash Desk and Application are compromised by information flows detected by the verification, thereby violating individual policies. Moreover, in case of the Application component, no verification result is available for the barcodeScanner and cashBox policies because Uppaal is running out of the memory it is allocated by default. When analyzing these cases in detail, it appears that a large number of ports is classified as critical by the respective security policy. Thus, we attribute the failure to the increased number of critical inputs being perturbed during verification as described in Section 5.5.1. Our findings therefore suggest that the abnormal termination of Uppaal's model checking procedure is caused by the growing state space resulting from the perturbation.

Despite these failures, the security policies for the lightDisplay, printer, and productDispatcher port are always adhered to by all components. Thus, since the architecture is expected to be secure with respect to these policies, we regard this expectation as satisfied. In case of the residual policies, for which the architecture is expected to be insecure, our verification results indicate that either Cash Desk or Application violate their refined policies. Accordingly, the verification of these policies satisfies our expectations as well, regardless of the aforementioned cases in which no verification result could be obtained for the Application. In summary, our proposed technique was able to verify each of the eight policies according to expectations, without assessing an insecure architecture as secure or vice versa. Hence, with respect to our RQ, we regard the verification results as accurate. Nevertheless, the abnormal termination in two cases suggests that further studies should analyze how the proposed technique scales on verification of large state spaces. In particular, the scalability should be considered in the presence of security policies that classify large amounts of information as critical and therefore require an excessive perturbation.

5.6.5 Validity

In the following, we discuss potential threats to the validity of our results from Section 5.6.4. According to Runeson and Höst [RH09], we distinguish between construct validity (questioning the suitability of our measurements to answer the research question), internal validity (questioning outside influences on our measurements), external validity (questioning the generalizability of our results), and reliability (questioning the replicability of our study).

With respect to *construct validity*, a threat to our results is that both the refinement of security policies and the modeling of the component behavior have been carried out as manual tasks by ourselves. Thus, our study is not based on predefined evaluation artifacts that serve as a *ground truth*. On the one hand, the applied well-formedness rules from Section 4.6 admit a certain degree of freedom, such that policies can be refined in various ways. On the other hand, instead of analyzing a given implementation of the component behavior as is, the behavioral models are only loosely based on the description of CoCoME's use cases [Her+08]. As a consequence of these manual interventions, one cannot rule out the possibility that the security policies or component behaviors have been specially tailored to provoke certain results, thereby affecting the measured accuracy. In our study, we aim to reduce the dimension of this threat by taking CoCoME's predefined security requirements and use cases as a basis for our data collection, instead of creating such artifacts artificially.

Moreover, since evidence is given on the basis of expert opinion, the *internal validity* of our study is threatened by human errors that could have been made during the manual assessment of the expected security results. If an expected result is incorrectly assessed upfront, it potentially affects the measured accuracy of the corresponding verification result. Thus, we may have committed measurement errors by regarding verification results as accurate although they are actually inaccurate, or vice versa. Whenever a policy violation is detected during verification, we aim to reduce this threat manually by double-checking the counterexamples provided by Uppaal. Thereby, we ensure that the detected violation actually represents an information flow that justifies our manual assessment of the corresponding expected value.

A threat to the *external validity* is that our findings bear little relation to the original CoCoME case and its associated security requirements. On the one hand, the behavioral models of our study omit the detailed data handling of components, abstracting from the way in which they process, store, or propagate individual parameters on the invocation of a service (cf. Section 5.6.2). On the other hand, according to the procedural programming style underlying CoCoME, the associated security requirements are based on a definition of information flow [GMB17] that differs profoundly from the one applied in our approach. For these reasons, the security values assessed or verified in our study are not necessarily transferable to the original CoCoME case. The external validity is also threatened because CoCoME is not a CPS. Whereas hardware components are part of the trading system, the case does not reflect their interaction with the physical system context. In particular, the time restrictions that we impose on the behavioral models are fictitious, instead of being induced physically. Therefore, it might not be possible to generalize the measured accuracy to real-world cases that are based on CPSs in a narrower sense.

To foster *reliability* of the obtained results, we facilitate the replication of our study by other researchers. To that end, we provide access to our collected data in [*Ger20a], including (i) the component behaviors given in terms of Uppaal timed automata, (ii) the corresponding automata checked by Uppaal to verify the information flow security, (iii) the model transformations used to construct these automata as described in Section 5.5, and (iv) the launch configurations used to invoke these transformations for each security policy.

5.7 Limitations

Inherent Limitations of the Refinement Check. As an extension of MechatronicUML's refinement check [HBDS15] and its underlying concept of test automata [JLS00; ABL98], our verification technique is subject to certain inherent limitations [Bre10, pp. 62–63]. First, checking a refinement relation between two systems by means of test automata requires the transition function of the more abstract system to be deterministic [JLS00, p. 27]. In our approach, due to the underlying principle of self-composition, the behavior of a component acts both as the abstract system being refined and (in its perturbed form) as the refining system. Therefore, our technique is restricted to component behaviors with deterministic transition functions as well. In the context of MechatronicUML, this restriction does not imply any consequences because the real-time statecharts used to model the component behavior must be deterministic anyway, expect for nondeterministic choices being modeled explicitly [HBDS15, p. 259]. Another inherent limitation is that the abstract system must not include any τ transitions [JLS00, p. 27]. Thus, each edge must be labeled with a symbol other than τ. In contrast, MechatronicUML generally allows τ transitions to be used in real-time statecharts because received messages are optional [*Dzi+16]. Hence, this limitation affects our technique. Taken as a whole, our approach is not applicable in a context that requires nondeterminism or τ transitions. In such contexts, alternative verification techniques for timed bisimulation [Čer93; WL97] would need to be taken as a basis. We refer the reader to [Bre10, pp. 62–63] for a closer examination of the aforementioned limitations.

As another inherent limitation, the refinement check does not take message parameters into account [Hei15, p. 134]. Therefore, it is not sensitive to differing parameter values. With respect to security, such differences may obviously constitute an information flow. As an extension of the refinement check, our proposed verification technique is not capable of detecting such flows, which has also been identified as a threat to the validity of our case study discussed in Section 5.6.5. An obvious approach to address this shortcoming is to encode all possible combinations of parameter values as individual symbols used to synchronize edges. However, this approach implies a considerable blowup of the underlying test automata because each possible combination of parameter values gives rise to distinct edges (cf. Section 5.5.2). Future works therefore need to address the scalability problems involved by this approach.

Assumptions & Guarantees. The security guarantee provided by our technique is based on the assumption that the original construction of test automata [HBDS15] ensures accuracy, i.e., indicates whether a weak timed bisimulation is satisfied or violated without committing an error. Moreover, we also assume that the accessible information is restricted to the messages passed over discrete ports. In contrast, we leave out of consideration that a component might make information accessible over other communication channels, such as signal exchange over hybrid ports. Furthermore, the provided guarantee is only transferable to a real system if the accessibility of information is not increased during the downstream development phases, in which components are deployed to concrete execution platforms. Such an increased access could be exploited by malicious actors to establish additional information flows, which are not covered by our model-driven verification technique. Refining the modeled behavior such that verified properties still hold for a real, deployed system is a general challenge. With respect to security, our approach must cope with the additional challenge that verified information flow properties of a system are not guaranteed to be preserved when the system specification is refined [Jac89]. This phenomenon is also known as the *refinement paradox* [Ros95].

At the technical level, another assumption relates to the corrections made by an automaton that undergoes perturbation. Such corrections enable the perturbed automaton to deviate from the communication behavior of the unperturbed automaton with respect to particular messages. As described in Section 4.4, corrections are restricted to neutral or critical outputs as well as neutral inputs. Due to the asynchronous communication, the sending of neutral or critical outputs is not subject to any restrictions because messages can be freely sent without asking the receiver's permission. In contrast, using neutral inputs for corrections is only possible on the assumption that the respective received messages have been delivered as needed. Accordingly, this assumption imposes restrictions on the communication behavior between a sending and a receiving component. Inputs with other sensitivities are not affected by this assumption because deviations are either explicitly considered as for critical inputs, or ruled out completely as for observable inputs. To handle the case of neutral inputs, our architectural well-formedness rules given in Section 4.6.2 therefore prevent inputs from being used for corrections at the global, architectural level. At the local level of individual components, this thesis does not address the question whether MechatronicUML's compositional verification approach (cf. Section 2.3.3) is suitable to guarantee that a sending component fulfills its behavioral contract in such a way that particular messages are always delivered to the receiving component as needed. We leave such considerations to future work.

Soundness & Completeness. On the basis of the aforementioned assumption that the original refinement check provides accurate verification results, we adapted the technique to properly hide non-observable communication. Thereby, we aim to ensure that unobservable deviations do not lead to the indication of false positive information flows. To the best of our knowledge, our adaptations did not introduce false negative errors, in which an information flow fails to be detected. Accordingly, no such errors were revealed when conducting our case study in Section 5.6. Nevertheless, there might be false positives left that have not been ruled out by our adaptations. For example, due to the fact that we omit invariants under certain circumstances described in Section 5.5.2 and Section 5.5.3, our technique might enable an automaton to delay in the concerning locations until after their regular invariants have expired. In this case, particular edges are not forced to fire before the expiry, potentially preventing an automaton from making corrections in response to a perturbation.

Quantitative Information Flow. As illustrated in Section 5.3, our technique is limited to qualitative verification results, indicating whether a system is compromised by an information flow or not. However, it might also be desirable for software engineers to learn the amount of information that is exposed by a flow, enabling them to reason about security from a quantitative point of view. For example, quantitative security issues are raised in the context of *differential privacy* [Dwo11]. Further, reasoning about such issues is the subject of a line of research on *quantitative information flow* [Smi09]. Accordingly, a promising extension of our work is an integration of quantitative analysis techniques. To achieve this goal, future works might build upon various quantitative extensions of the Uppaal model checker, which enable reasoning about costs or probabilities [LLN18]. In the context of security, such techniques have already been used to reason about quantitative aspects of attacks [KS17].

Declassification. Another limitation of our technique is that the verified security policies are unconditional, associating each message with a fixed sensitivity (cf. Section 4.4). Accordingly, we do not support the concept of *declassification* [SS09] as already introduced in Section 3.7. Declassifying information could help instruct our verification technique to *not* report information flows under well-defined exceptional circumstances, e.g., in a particular location or given a certain clock valuation. In the scope of MechatronicUML, the specification of such exceptions has already been prototyped in a bachelor thesis [Cor18]. Future works must integrate the specified exceptions into the construction of the test automata and thereby ensure that declassified information is taken into account during verification.

Hardening. Whereas the goal of our technique is to detect insecure behavior, our approach does currently not help software engineers prevent information flows proactively, or fix flows reactively after they were detected. Both supporting measures could enable a system to be *hardened* against security incidents in a systematic way. On the one hand, the MechatronicUML method already provides engineers with a constructive, semi-automated technique to synthesize the behavior of a component from the predefined coordination protocols that need to be implemented [DGB14; EH10]. The approach enables an incorporation of well-defined interdependencies between the component's ports, each filling a role of an individual protocol (cf. Section 2.3.3). Thus, a promising enhancement is to also consider interdependencies that must be avoided in order to adhere to a given security policy. Thereby, engineers could prevent policy violations from the ground up.

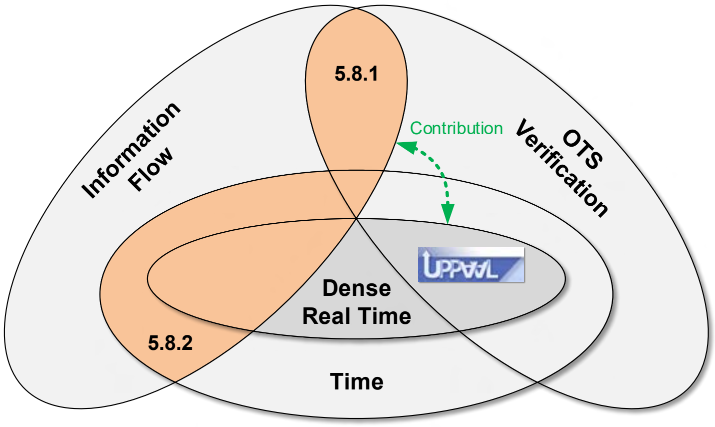


Figure 5.9: Overview of related works from different areas.

On the other hand, further existing works help fix the given behavior of a system if it is compromised by an information flow [AS19; BCLR15; BFPR02]. To apply such techniques in our context, future works need to address potential repercussions that an automated fix may have on the functional requirements of a component. In MechatronicUML, fixing the information flow with the help of automated techniques must not invalidate any properties that have been verified upfront by the compositional approach described in Section 2.3.1.

5.8 Related Work

As shown in Fig. 5.9, our contribution is a novel combination of the following characteristics. First, we consider dense real-time behavior and thereby enable a timing-sensitive verification. Second, we apply Uppaal as an off-the-shelf verification tool. Thus, Section 5.8.1 first gives an overview on general verification techniques for information flow and their support by off-the-shelf tools. In Section 5.8.2, we focus more specifically on timing-sensitive approaches.

5.8.1 General Verification Techniques for Information Flow Security

A traditional verification technique for information flow is *unwinding* [GM84; Man00b; Bos+04]. In this technique, a global security property of a system is deduced from local properties of individual system actions. In the context of state-based systems as we consider, such actions correspond to the individual state transitions. However, due to the real-valued state space of timed automata, localizing the verification to properties of single state transitions is a challenging task. In fact, to the best of our knowledge, unwinding has not been applied in the scope of real-time systems. Furthermore, whereas the verification results obtained through unwinding are generally sufficient to guarantee a certain security property, they are not always necessary [Man00b] and may therefore indicate false positive information flows.

In the area of language-based information flow [SM03], the security of programs is often verified with the help of *type systems* [VIS96]. Such approaches enforce security by inferring the types of the individual language constructs used in a given program. Type systems aim to ensure soundness of the verification results and therefore detect all insecure programs in a reliable fashion. However, similar to the unwinding technique, they may reject secure programs as insecure. Compared to our work in the context of state-based systems, the language-based approach involves a profoundly different model of computation. Therefore, type systems are not directly applicable in our context. We refer the reader to [FRS05; MZ10] for a consideration of information flow security under different models of computation.

Another prominent verification technique that originates from the field of language-based security is commonly known as *self-composition* [BDR11]. According to this approach, the indistinguishability of two different program executions is verified by analyzing single executions of a composite program, being composed of two copies of the original program. By self-composing programs, a hyperproperty such as information flow (cf. Section 2.4.2) is made accessible to standard verification techniques, which are otherwise restricted to properties like safety or liveness (referring to individual executions in isolation). Consequently, numerous language-based approaches have taken advantage of the underlying idea to verify the information flow security of programs using techniques like theorem proving [DHS05], model checking [HWS06], or combinations thereof [TA05]. Furthermore, self-composition has also been adopted outside the field of language-based security. For example, van der Meyden and Zhang [MZ07] as well as D'Souza et al. [DHRS11] both make use of the underlying idea to verify the information flow security of state-based systems by means of model checking. Thereby, these works resemble our approach in the context of timed automata. Nevertheless, they do not take real-time systems into account and therefore fail to conduct a timing-sensitive analysis that helps detect timing channels. Furthermore, whereas these works develop algorithmic decision procedures that are generally amenable to model checking, they do not discuss the applicability of existing, off-the-shelf model checking tools.

Information flow properties like ours, which are intrinsically defined on the basis of bisimulation (cf. Definition 4.1 on page 73), are also amenable to the reuse of standard verification techniques for *bisimilarity*, as discussed by Focardi et al. [FPR02]. Thus, in our context, existing decision procedures for timed bisimulation [Čer93; WL97] are generally applicable. Nevertheless, since such procedures must handle the infinite state space of real-time systems algorithmically, they require great development efforts to ensure efficiency and correctness of the verification algorithms to be implemented. In contrast, our approach takes an off-the-shelf verification tool as a basis, which enables us to benefit from its proven maturity instead of implementing the underlying verification algorithms from scratch.

Instead of reducing information flow security to standard verification problems, a complementary approach is pursued by Finkbeiner et al. who adapt model-checking techniques to information flow properties in particular [Dim+12], or even to hyperproperties in general [FRS15; Rab16, Chapter 5]. Hyperproperties represent a relatively new line of research that has not yet established off-the-shelf verification tools with an advanced degree of maturity. Whereas the above works do not focus on a timing-sensitive analysis, we will discuss ongoing research on the verification of timed hyperproperties in the upcoming Section 5.8.2.

5.8.2 Timing-Sensitive Information Flow Security

In the field of language-based security, Agat [Aga00] uses a type system (cf. Section 5.8.1) to rule out timing channels of imperative programs, whereas the same approach is pursued by Mu and Qin [MQ17] in the context of a formal specification language. Another language-based work on the detection of timing channels in imperative languages is presented by Giacobazzi and Mastroeni [GM05], however, without proposing a particular verification technique. Unlike the above works, the type system proposed by Hedin and Sands [HS05] is sensitive to the history of executed instructions, which may affect the time needed to execute a subsequent instruction. Thereby, the authors account for the effects of an instruction cache on the execution times. The same approach is adopted by Kashyap et al. [KWH11] to enforce information flow security by means of dedicated task scheduling strategies. Similarly, Son and Alves-Foss [SA09] focus on the scheduling of tasks as well, however, explicitly restrict their approach to a discrete notion of time.[2] In fact, opposed to our proposed verification technique, none of the above works explicitly consider a notion of dense real time.

Approaches towards information flow in process algebras have been enhanced with a notion of time by Focardi et al. [FGM03], extended by Huang et al. [HPLP09].[2] Similarly, Rafnsson et al. [RJB17] also verify the absence of timing channels in compositions of interacting processes. Whereas the above works are all based on a discrete time model, Roscoe and Huang [RH13] extend process algebras with both discrete time and dense real time. Similar to our work, the authors of the latter approach propose verification techniques that take an off-the-shelf model checking tool as a basis. However, these techniques are either restricted to the case of discrete time, or approximate the dense real-time behavior in a discrete fashion.

Whereas all aforementioned works differ profoundly from our approach in the underlying model of computation, Köpf and Basin [KB06] address the field of synchronous, state-based systems. A commonality shared with our work is that the authors apply the idea of self-composition (cf. Section 5.8.1) to make use of an off-the-shelf model checker. Nevertheless, the approach is once more restricted to a discretized notion of time. In summary, none of the approaches discussed above provide a verification technique in which time is treated as a continuous, real-valued magnitude. As previously discussed in Section 4.9.3, the aforementioned related works are therefore limited in the detection of timing channels.

In contrast, there are also approaches that handle timing channels in real-time systems given as timed automata. Cassez [Cas09] takes into account a security property that turns out to be undecidable. Accordingly, the author is not concerned with concrete verification techniques. Whereas Benattar et al. [BCLR15] synthesize controllers that enforce security of timed automata, Nielson et al. [NNV17] derive secure timed automata from language-based specifications. In [VNN18], the same authors propose an algorithm that imposes security restrictions on a given automaton. As distinct from the above approaches, André and Sun [AS19] use off-the-shelf tools to synthesize timing parameters of secure timed automata. Similar to our work, the authors apply the principle of self-composition and reduce the problem to a reachability check. However, according to the constructive nature of the above approaches, they are not primarily concerned with verification.

[2] The titles of [SA09; FGM03; HPLP09] refer to *real time*, despite underlying discrete notions of time.

Lanotte et al. [LMT10] provide a theoretical framework to investigate information flow properties of timed automata. Being defined on the basis of bisimulation, these properties are conceptually similar to the one underlying our work (cf. Definition 4.1 on page 73). In [LMT02], the same authors consider a more specific privacy property, which is amenable to off-the-shelf model checking tools by applying the idea of self-composition [Gor+04, p. 173]. Nevertheless, the more general properties investigated in [LMT10] are not provided with applied verification techniques. In contrast, Barbuti and Tesei [BT03] reduce the verification of the information flow security to a reachability check. Similar to our work, the authors apply Uppaal as an off-the-shelf tool [Tes04, Section 5.7]. However, the verified property only guarantees that the set of reachable locations does not depend on critical information [Tes04, p. 121].[3] This approximation affects both soundness and completeness. On the one hand, the observable events of an automaton might not depend on critical information, even if the reachability of locations does. On the other hand, if the reachability does not depend on critical information, the observable events might do so nevertheless. Thus, in the general case, the proposed technique fails to give accurate security guarantees.

In summary, the related works discussed above do either not provide (accurate) verification techniques that apply off-the-shelf tools, or the techniques they provide are restricted to a discrete notion of time. Accordingly, as illustrated by Fig. 5.9, the novelty of our contribution is that we verify the information flow security both under consideration of dense real-time behavior and with the help of an off-the-shelf technique. Only recently, verification approaches with a similar combination of characteristics have been developed for the larger class of timed hyperproperties [HZJ19; Hei18]. As independently proposed in our work, both approaches make use of self-composition as a means to verify hyperproperties of timed automata. Unlike our technique, Ho et al. [HZJ19] mainly investigate the theoretical decidability of the verification problem. In contrast, the approach by Heinen [Hei18] resembles our work more closely because it is based on a practical application of Uppaal as an off-the-shelf verification tool.

5.9 Summary

We presented a tool-supported verification technique for the information flow security of real-time systems given in the form of timed automata. On the basis of the component-based security policies developed in Chapter 4, our technique helps check the behavior of MechatronicUML components against information flows such as timing channels. Thereby, we complement MechatronicUML's compositional verification approach for functional properties with a corresponding technique to verify security properties. Our approach takes the refinement check by Heinzemann et al. [HBDS15] as a basis.

We evaluated the accuracy of our technique by conducting a security-related extension of the community case study CoCoME. On the basis of predefined information flow policies, we demonstrated that the obtained verification results satisfy the expectations assessed manually on the basis of expert opinion. However, our study also revealed a scalability bottleneck of our technique, which takes effect if large amounts of information are classified as critical.

[3] We refer the reader to Jaume et al. [JATM13] for a discussion of the security trade-off involved by this approach.

Our verification technique enables software engineers in the context of MechatronicUML to check automatically whether the behavior of a component adheres to a given security policy. In the light of the composability attested in Section 4.7, the security of a system is deducible from the verification results of its bottom-level subcomponents. Instead of analyzing the real-time behavior by means of proprietary algorithms that need to be implemented from scratch, we enable engineers to benefit from the maturity of an off-the-shelf verification tool.

6

Imperative Refinement of Declarative Model Transformations

In this thesis, transformations of models have been previously used to translate flow policies into component-based security policies (cf. Section 4.5), or to analyze timed automata for security properties (cf. Section 5.5). As described in Section 2.1.2, such model transformations are traditionally distinguished between declarative and imperative [MG06]. Using declarative transformations, developers define patterns of model elements that relate to each other. These patterns declare *what* source and target elements are transformed into each other when executing a transformation. In case of imperative transformations, developers specify sequences of algorithmic instructions. Thereby, they control explicitly *how* source elements are transformed into target elements during execution.

Today, developers may choose between dedicated transformation languages from both categories [KG17]. However, as suggested by empirical investigations [Heb+18; BCG19b], the advantages of special-purpose transformation languages over general-purpose programming languages are not perceived as significant by developers. In this context, we address the problem that using transformation languages to encode the logic of transformations often requires large-sized, verbose transformation definitions. Neither declarative nor imperative languages provide a silver bullet for this *verbosity* and therefore fail to reduce the development effort for model transformations. On the one hand, declarative languages save developers from the need to encode explicitly how transformation rules are to be applied, implicitly delegating the rule application to the execution engine. However, in practice, the transformation logic often involves complex alternatives between rules, or exceptions to the basic rules. Encoding such alternatives or exceptions enlarges the overall size of the rule set, affecting both maintainability and performance [Str+16]. On the other hand, since imperative rules are applied explicitly, they enable a more flexible encoding of alternatives or exceptions. As a drawback, imperative languages require developers to implement large amounts of repetitive *boilerplate* instructions by hand. These instructions are needed to link the elements created by different rules and thereby produce a coherent target model. Taken as a whole, developers are in a dilemma because they must cope with the verbosity of transformations one way or the other, enlarging either the size of the rule set or the amount of boilerplate instructions. The goal of this chapter is therefore to combine the best of both worlds, reducing the verbosity of transformations by integrating the benefits of declarative and imperative languages.

State of Research. Hybrid transformation languages like the Atlas Transformation Language (ATL) combine a declarative application of rules with an imperative rule implementation [JABK08]. However, although rules are applied automatically, developers must still implement boilerplate instructions manually to ensure that the target elements of a rule are linked with further elements created by other rules. To save this manual effort, approaches towards *metamodel matching* aim to infer transformation rules from the underlying source and target metamodels [LFH14]. However, this approach assumes that rules can be inferred fully automatically, without enabling developers to preconfigure the inference upfront with manually defined rules. In contrast, declarative approaches as presented in [FB16] aim to link target elements semi-automatically on the basis of predefined rules, but without the need to encode the transformation logic in full detail. However, by focusing on declarative rules, this approach does not enable a flexible, imperative encoding of alternative or exceptional logic.

Contributions. To benefit from the strengths of both declarative and imperative transformations, this chapter contributes a hybrid transformation approach that enables a declarative transformation to be refined with imperative instructions. Initially, we enable developers to give a transformation definition in terms of a *transformation model* [Béz+06], encompassing simple declarative *type mappings* that reflect recurrent relations between classifiers (cf. Section 2.1.1) from the source and target metamodels. To automate the linking of target models, we present a heuristic inference engine that augments the type mappings with additional *feature mappings*, defining how to translate features of source classifiers into features of target classifiers. As described in Section 2.1.1, such features link the individual model elements with each other and thereby ensure coherence of models. By linking target elements with the help of heuristics, we prevent developers from the burden of handcrafting repetitive boilerplate instructions. Since our engine matches corresponding features automatically by analyzing the manual type mappings, we adopt a semi-automated approach. To execute a transformation, the inferred mapping models can be interpreted using a *framework* [Joh97] that we conceptualize in this chapter as well. If the transformation logic involves alternative or exceptional parts that cannot be encoded using our mapping models (or cannot be inferred correctly), the framework enables developers to refine the execution manually with imperative instructions. The implementation of the framework is solely based on general-purpose language facilities, which we identify explicitly to check state-of-the-art imperative transformation languages against them. Thereby, we analyze the feasibility of implementing our framework using particular host languages. In combination, our contributions ensure conciseness of transformation definitions because neither the simple logic of linking elements, nor complex alternatives or exceptions to this logic need to be encoded verbosely.

Novelty. Compared to the existing works on automating the definition of model transformations, our contributions are novel by ensuring manual control of the automation. First, unlike metamodel matching, we enable developers to preconfigure the inference of feature mappings upfront with a given set of type mappings. Second, unlike the approach from [FB16], we support an imperative refinement of declarative mappings in cases when the logic of a transformation cannot be fully encoded in a declarative fashion. In summary, we thereby support manual intervention by transformation developers, enabling them to benefit from a reduced verbosity even if the generation of a transformation definition cannot be fully automated.

Publication. We published an early version of our execution framework in a conference paper [*GSB17], restricting our approach to refinements of endogenous transformations that are based on the *implicit copy* design pattern [LK14]. Conceptually, since such transformations use a common source and target metamodel, each classifier is implicitly mapped to itself. We generalized our approach to exogenous transformations in a bachelor thesis [Bud18], taking into account explicit mappings between classifiers from different metamodels. Building upon these results, we presented our heuristic inference engine in a workshop paper [*GB19].

Outline. The remainder of this chapter is structured as follows. We initially summarize our scientific contributions in Section 6.1. In Section 6.2, we give an overview on our work and introduce the concept of mapping models in Section 6.3. Section 6.4 presents our heuristic inference engine, before we propose the execution framework for mapping models in Section 6.5. Subsequently, we conduct case studies to evaluate our approach in Section 6.6. We discuss known limitations in Section 6.7 as well as related work in Section 6.8. Finally, Section 6.9 sums up this chapter.

6.1 Scientific Contributions

To overcome the verbosity of model transformations, the hybrid approach presented in this chapter comprises the following contributions:

- We introduce a concept of mapping models, enabling developers to define a model transformation in terms of type mappings between source and target classifiers.

- We propose an automated inference engine, enabling developers to augment these type mappings with feature mappings, which are used to link the elements of models resulting from a transformation.

- We present a framework for the automated execution of the augmented mapping models, encompassing a generic execution algorithm that can be refined using imperative transformation instructions.

- We identify the required language facilities underlying our execution framework and check state-of-the-art imperative transformation languages against these requirements.

- We conduct five case studies in which we showcase the ability of our approach to reduce the verbosity of model transformations accurately and effectively.

6.2 Overview

Since we address the general case of exogenous transformations (cf. Section 2.1.2), we motivate our contributions on the basis of *language translation* as the most widespread category of transformation intents [Bat+16, p. 179]. As a concrete example, we refer to the translation from active structures used in MBSE into component architectures used in CBSE (cf. Section 4.5), illustrating the source and target metamodels in Fig. 6.1.

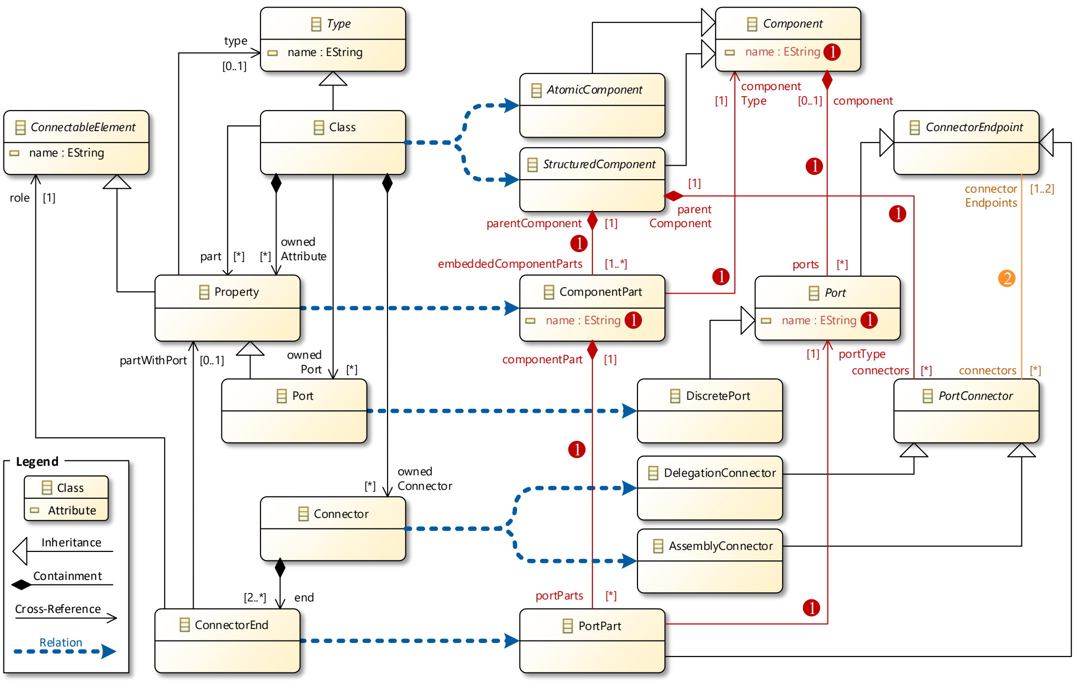


Figure 6.1: Relations between classes for the translation from UML to MechatronicUML.

According to an integration of Consens into SysML [KDHM13], we assume that an active structure is given as a UML class diagram [OMG17]. Thus, on the left, we show a simplified excerpt from the UML metamodel, representing each system element as a Class. The nesting of elements is realized with the help of a Property, which refers to a Type. In our context, this Type corresponds to another Class as a nested element. A Port is a specific Property that enables a Class to be linked by means of a Connector, thereby representing relations between elements such as information flows. To this end, a ConnectorEnd represents the coupling between a Connector and a Port as a specific form of ConnectableElement.

On the right, we show a simplified excerpt from the MechatronicUML metamodel, reflecting its underlying component model (cf. Section 2.3.2). Accordingly, we distinguish between bottom-level components represented by AtomicComponent and higher-level composite components represented by StructuredComponent. In both cases, a Port enables the interaction of a Component. Since we restrict our example to discrete components, DiscretePort is the only relevant type of port. A ComponentPart refers to another Component, which thereby acts as a subcomponent. Furthermore, a PortConnector binds two instances of a ConnectorEndpoint, representing either a Port of a Component or a PortPart of a ComponentPart. On this basis, dedicated types of connectors are used to distinguish between delegations and assemblies. Whereas a DelegationConnector connects a Port to a PortPart, an AssemblyConnector connects one PortPart to another. In the following, we use the example transformation to define our addressed problem in Section 6.2.1. Next, in Section 6.2.2, we derive requirements for our solution, which we propose in Section 6.2.3.

6.2.1 Problem Definition

To define a model transformation, developers must indicate which source and target classes relate to each other. For example, declarative approaches like triple graph grammars [ALS16] relate the classes by means of a *correspondence metamodel*, whereas imperative languages like QVTo [OMG16] reflect such relations in the signatures of operations. In our example from Fig. 6.1, a Class either relates to an AtomicComponent or a StructuredComponent, depending on whether it comprises at least one Property as a part. Accordingly, such a Property bears relation to a ComponentPart. A UML Port is related to a DiscretePort, and a UML Connector relates either to a DelegationConnector or an AssemblyConnector. Finally, a ConnectorEnd is related to a PortPart, provided that it refers to a Property as a partWithPort.

As shown by numerous scientific works [DGC17; FB16; Gue+13; BVJM13; Wim+10; LHBJ06], relations between source and target classes can be used to establish transformation rules. Given a set of source elements, the relations define the target elements that need to be created, thereby mapping source to target models. Nevertheless, developing fully functional transformations often leads to verbose transformation definitions because developers must cope with two remaining challenges that we describe in the following. As a point of reference, we will refer to the approach by Freund and Braune [FB16], which uses mappings from source to target models for a similar purpose as in our work.

Boilerplate Initialization of Features

To form a coherent model, target elements must be linked with each other by initializing their features (cf. Section 2.1.1). The initialization applies to attributes (e.g., name in Fig. 6.1), as well as references including containments (e.g., between Component and Port) or cross-references (e.g., between ComponentPart and Component). We address the problem that initializing the features marked by ❶ in Fig. 6.1 is unnecessarily tedious. Due to the relations between source and target classes, the correct initialization of the target elements is partially obvious from the initialization of the given source elements. For instance, since a Property represents a part of a Class, it is obvious from the given relations that the related ComponentPart must be one of the embeddedComponentParts of the related StructuredComponent. Similarly, a Component must obviously act as the componentType of a ComponentPart whenever the related Class serves as the type of the related Property. As another example, the name of a ComponentPart can obviously be carried over from the name of the related Property one-on-one.

In imperative languages like QVTo, initializing the above target features requires developers to handcraft large amounts of repetitive boilerplate instructions in a verbose fashion. Declarative approaches like the one by Freund and Braune [FB16] reduce this verbosity because they deduce the initialization of a coherent target model automatically. Nevertheless, as a limitation pointed out by the authors, their approach only ensures that a target model is syntactically valid with respect to its metamodel [FB16, p. 293]. In contrast, the approach does not aim to match target features with corresponding source features. As a drawback, a portion of the features marked by ❶ might be left uninitialized, potentially losing information carried by the source model.

Increased Size of Rule Sets

In most cases, not all features of the target metamodel enable an automated initialization as described above. Often, this inability is caused by *structural heterogeneity* [Wim+10] in comparison to the source metamodel. For example, when transforming a UML Connector into a DelegationConnector of MechatronicUML, the connectorEndpoints (marked by ② in Fig. 6.1) must be initialized with a DiscretePort. However, compared to other features, this initialization is less obvious because the source Connector does not directly refer to a Port. Therefore, this Port in the source model is not directly navigable and cannot be used to establish a relation to the DiscretePort in the target model. Thereby, the source metamodel differs from the target metamodel, in which each DelegationConnector refers directly to the respective DiscretePort. Due to this structural heterogeneity, the connectorEndpoints constitute an exception to the default transformation logic. Unlike the majority of features, the initialization of this reference is hard to deduce from the given relations between source and target classes. As can be seen from this example, there are still situations in which transformation definitions must be extended manually, either to make exceptions to the basic transformation rules or to choose explicitly between alternative rules.

We address the problem that, in such situations, purely declarative approaches require transformation developers to encode exceptional or alternative initializations either by increasing the size of individual transformation rules, or by increasing the number of rules in the rule set [Str+16]. In both cases, the overall size of the rule set increases, thereby inducing verbosity of the transformation definition. In the example, the endpoints of an AssemblyConnector or DelegationConnector must be initialized on a case-by-case basis, e.g., by establishing two separate rules. Additionally, in case of a delegation, the respective rule must be increased itself because it requires the developer to (i) iterate over each end of the source Connector, (ii) select a ConnectorEnd without a partWithport, and (iii) navigate to the role and thereby obtain a Port that relates to the DiscretePort in the target model. In case of declarative transformation languages, increasing the overall size of the rule set implies not only higher maintenance efforts, but may also cause performance bottlenecks during the automated rule application [Str+16]. Accordingly, both maintainability [AB11; KGBH10; RNHR13] and performance [GTB18; ABKP11] are considered as crucial quality factors for model transformations.

To decrease the size of the rule set and the associated verbosity, *reuse* of model transformations is a prominent research topic [Bru+20; CFSS16; Kus+15]. For example, declarative approaches have been extended with mechanisms such as *rule inheritance* [Wim+12] or *variability-based transformation* [Str+18]. However, to overcome the aforementioned verbosity issues of declarative approaches, controlling the application of rules more explicitly by means of imperative instructions is attractive to developers [Shi19]. For example, as described above, initializing the endpoints of a DelegationConnector involves operations such as iteration, selection, or navigation of model elements. As another shortcoming pointed out by Freund and Braune [FB16, p. 293], their approach does not enable developers to express such complex operations by means of imperative instructions.

6.2.2 Requirements

From the example problem defined in Section 6.2.1, we derive a set of requirements (R1–R3) that our contributions must satisfy:

(R1) Accuracy: When deducing definitions of model transformations automatically, corresponding source and target features must be matched as accurately as possible.

(R2) Reduced Verbosity: Deducing a transformation definition automatically should reduce its verbosity in comparison to a manual, purely imperative definition.

(R3) Imperative Logic: To enable exceptional or alternative initializations of features, transformation definitions must support the encoding of imperative transformation logic.

The subsequent requirements (R4 and R5) do not directly derive from our example problem and will be justified below:

(R4) Executability: To enable the automated transformation of source into target models, the logic encoded by a transformation definition must be executable [CH06] on the basis of a proper transformation engine.

(R5) General-Purpose Language Facilities: To ensure a wide-ranging applicability of our approach, implementing the above transformation engine should require only general-purpose facilities available in state-of-the-art imperative transformation languages.

6.2.3 Solution Approach

We address the requirements specified in Section 6.2.2 by proposing a hybrid, semi-automated approach that reduces the verbosity of model transformations by combining the strengths of both declarative and imperative transformation languages. First, to overcome the boilerplate instructions required by imperative languages to initialize features, we build upon a declarative approach and provide developers with means to deduce such initializations automatically. Second, to integrate alternative or exceptional transformation logic, we enable developers to refine these declarations with imperative instructions.

We illustrate the transformation process in Fig. 6.2. In the initial activity Define Type Mappings, developers create a transformation model encompassing declarative *type mappings* between classifiers from the source and target metamodels. These mapping models will be described in Section 6.3. The goal of the subsequent activity Infer Feature Mappings is to reduce the verbosity as demanded by requirement R2. To this end, we provide developers with a heuristic *inference engine* that augments the predefined type mappings with additional *feature mappings*, rendering boilerplate instructions unnecessary. To account for the accuracy demanded by requirement R1, we aim to map source features to their corresponding target features, thereby preventing a loss of the information included in the source model. In a technical sense, our inference engine is itself a model transformation because it produces an augmented mapping model. We will describe the engine in Section 6.4.

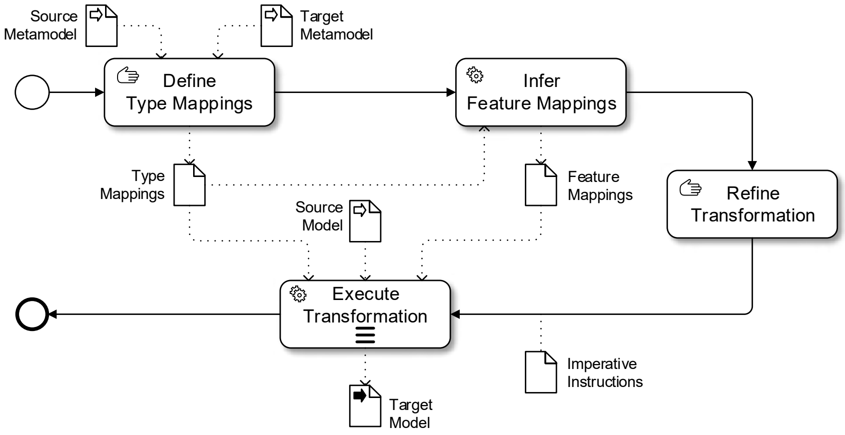


Figure 6.2: Overview of the proposed transformation process.

On the basis of the inferred feature mappings, we satisfy requirement R4 by providing an execution framework that acts as a transformation engine for mapping models. To this end, our framework implements an algorithm for the transformation of source into target models. Again, the execution algorithm is a model transformation itself, which is used to operationalize a mapping model and thereby acts as an interpreter. To meet requirement R3, our framework provides capabilities to refine the execution with *imperative instructions*. In the activity Refine Transformation, such refinements are specified manually by the developer using an imperative transformation language. Finally, in the activity Execute Transformation, the execution framework can be applied multiple times to transform a *source model* into a *target model*, taking into account the declarative type and feature mappings as well as the imperative transformation instructions. We describe our execution framework in Section 6.5.

6.3 Mapping Models

Our approach is based on a declarative *transformation model* [Béz+06] that is used to relate the source types of a transformation and their corresponding target types, thereby declaring a transformation definition. Since such a model comprises numerous type-level mappings between the source and target metamodels, it is referred to as *mapping model*. We define the structure of mapping models by means of a metamodel shown in Fig. 6.3. All classes highlighted in gray originate from the Ecore metamodel depicted in Fig. 2.1 on page 11, which we take as a basis for our approach. As can be seen, each mapping model is constituted by a MappingRoot, in which all mappings are directly or indirectly contained. In particular, a MappingRoot comprises a number of ClassifierMapping elements. We use ClassifierMapping as the abstract superclass of all kinds of type mappings supported in our approach. To ensure identifiability if necessary, each ClassifierMapping may be given an optional name. In the following, we first address the different kinds of type mappings in Section 6.3.1. Thereafter, in Section 6.3.2, we focus more specifically on feature mappings.

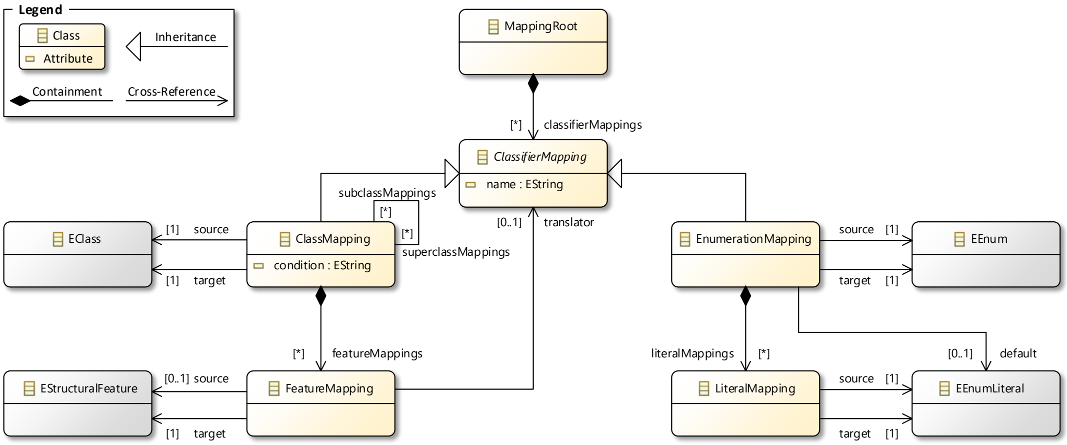


Figure 6.3: Metamodel for mapping models comprising type and feature mappings.

6.3.1 Type Mappings

The most basic form of ClassifierMapping is a mapping between a source and a target class. Such mappings are represented by a ClassMapping whereas source and target each correspond to an EClass. A class mapping establishes a transformation rule, according to which a new element of the target class is to be created for each existing element of the source class.

Classes are not limited in the number of mappings they are allowed to participate in. Accordingly, various types of correspondences can be modeled, differing in how often each class acts as source or target of a mapping. Wimmer et al. [Wim+10] distinguish such correspondences between *copying* (1:1), *partitioning* (1:n), *merging* (n:1), and *generating* (0:1) of elements. Both 1:1 and 1:n mappings are supported natively by our approach, enabling an element of a source class to be either copied to an element of a target class, or partitioned to multiple elements of different target classes. In addition, we also support n:1 mappings from distinct source classes to the same target class. However, please note that the source elements are not going to be merged into the same target element. Instead, such mappings describe numerous 1:1 correspondences, copying source elements with distinct classes to numerous target elements with a shared class. Besides merging, our approach also disregards 0:1 correspondences, which require target elements to be generated from scratch. Instead, both merging and generating of elements are typical operations that require a transformation to be refined imperatively (cf. Section 6.5.2).

As an example, Fig. 6.4 reflects the class mappings from Section 6.2.1 between the UML metamodel on the left and the MechatronicUML metamodel on the right. Accordingly, the example includes both 1:1 mappings (e.g., from Property to ComponentPart) and 1:n mappings (e.g., from Class to AtomicComponent and StructuredComponent). In the remainder of this section, we will first introduce additional modeling elements to control the mapping of model elements more precisely by means of context conditions or subtyping. Finally, we will briefly address the mapping of enumerations.

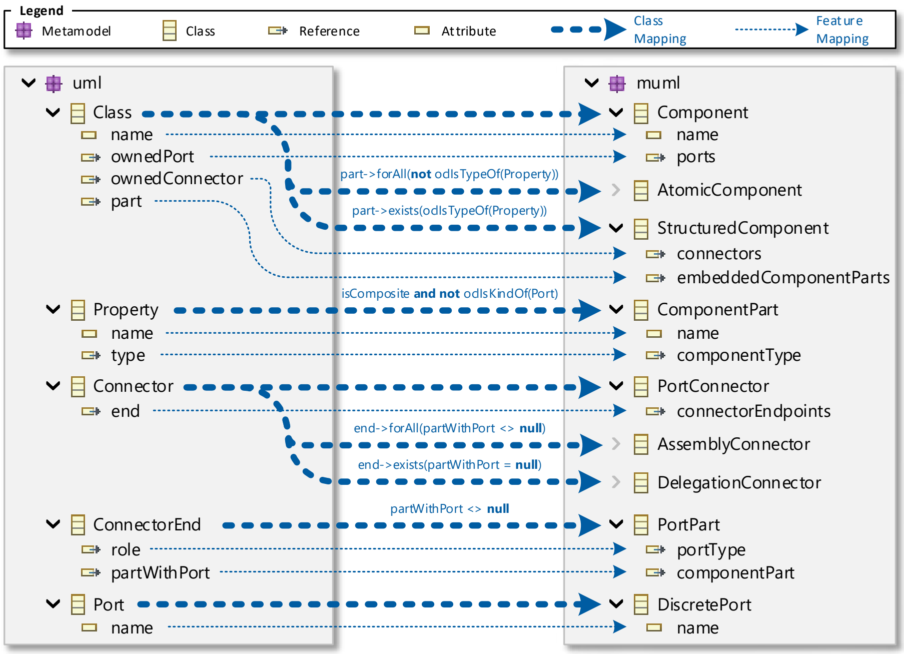


Figure 6.4: Mapping model for the translation from UML to MECHATRONICUML.

Context Conditions

Class mappings are not necessarily unconditional. Under certain circumstances, a target element should only be created if the source element is in a specific context (constituted by its attribute values or references to other elements). For example, the 1:n mapping of a Class in Fig. 6.4 does not represent a partitioning in the proper sense because the source element must be mapped *either* to an AtomicComponent or to a StructuredComponent, depending on its context. We therefore enable developers to attach such context conditions to class mappings, thereby restricting the circumstances under which the existing source elements are mapped to newly created target elements. This enables developers to control the creation of target elements more precisely. Accordingly, in the metamodel shown in Fig. 6.3, a ClassMapping can be equipped with a condition as an EString given in a constraint language. To check these conditions during the execution of our mapping models (cf. Section 6.5), we require the string to be a valid expression of the OCL [OMG14].

At the center of Fig. 6.4, we show the context conditions required by our example in terms of OCL expressions. For example, alternative conditions are used to choose between the two mutually exclusive mappings of a Class, depending on whether there exists a part that is of type Property. Furthermore, such a Property is only mapped to a ComponentPart if its isComposite attribute is true and it is not itself a Port. Another pair of alternative conditions is used to map a Connector either to an AssemblyConnector or a DelegationConnector, depending on whether there exists at least one end that refers to a partWithPort. Accordingly, a final condition requires that a ConnectorEnd refers to a partWithPort in order to be mapped to a PortPart.

Subtyping

As described in Section 2.1.1, metamodels support *subtyping* [LW94] of classes, thereby enabling features of a shared superclass to be inherited by multiple subclasses. For example, the MechatronicUML metamodel depicted in Fig. 6.1 uses subtyping relations on the basis of Component, PortConnector and ConnectorEndpoint as abstract superclasses. To account for such subtyping relations, a class mapping may represent the *superclass mapping* of multiple *subclass mappings* that share common properties. Accordingly, the example in Fig. 6.4 includes additional class mappings for two of the aforementioned superclasses, mapping Class to Component and Connector to PortConnector.

In the metamodel from Fig. 6.3, we illustrate these relationships by means of a bidirectional reference that enables a ClassMapping to indicate multiple subclassMappings and, conversely, superclassMappings. In this context, well-formedness of a mapping model requires that the source and target classes of each superclass mapping are *substitutable* by those of the corresponding subclass mapping. For example, in Fig. 6.4, the mapping between Class and Component acts as the superclass mapping of the two subclass mappings from Class to AtomicComponent and Class to StructuredComponent. This relationship is well-formed because a Component is substitutable by its subtypes AtomicComponent and StructuredComponent.

Opposed to the behavior of a default class mapping, superclass mappings do not create an element of their own (possibly abstract) target class. Instead, the responsibility for the creation of the target element is delegated from the superclass mapping to a subclass mapping, which creates an element of its associated subclass. The selection of this delegated subclass mapping depends on the given source element. To select a subclass mapping, the following two requirements must be satisfied. First, the source class of the subclass mapping must be substitutable by the class of the given source element. Second, the source element must also fulfill a context condition that may be attached to the subclass mapping (see above). In Fig. 6.4, the conditions attached to the two subclass mappings for Class are mutually exclusive. Thus, the choice of the delegated subclass mapping is unambiguous. However, in the general case, more than one subclass mapping might be selectable. To resolve such ambiguities, we leave the concrete selection procedure to the upcoming Section 6.5, where we describe the execution of our mapping models.

Conversely, whenever a subclass mapping is responsible for the creation of a target element, it is also obliged to account for the properties of a corresponding superclass mapping. For example, such properties might specify how to initialize additional features that the subclass inherits from the respective superclass. As described in Section 2.1.1, metamodels support *multiple inheritance*. A ClassMapping may therefore refer to multiple superclassMappings, as reflected by the metamodel in Fig. 6.3. Accordingly, on creation of a target element by a subclass mapping, the properties of each indicated superclass mapping must be taken into account as well. This is also true for all transitive superclass mappings, which must be taken into account recursively.

Enumerations

To initialize attributes that refer to enumerations as data types (cf. Section 2.1.1), our mapping models support *enumeration mappings* as well. Such mappings establish relations between corresponding source and target enumerations from the respective metamodels. In Fig. 6.3, we therefore introduce an EnumerationMapping as a second kind of ClassifierMapping. Such a mapping refers to a source and a target enumeration of type EEnum, and comprises a number of LiteralMapping elements. Each LiteralMapping maps a source literal to a target literal, whereas both source and target correspond to an EEnumLiteral from the corresponding source or target enumeration. However, instead of mapping all literals of the source enumeration by means of explicit literal mappings, an enumeration mapping may indicate a default value as a distinguished EEnumLiteral from the target enumeration. Thus, a source literal without an explicit literal mapping is implicitly mapped to this default target literal.

6.3.2 Feature Mappings

Besides the creation of target elements, a model transformation is also responsible for initializing their features. To this end, we enhance our class mappings by introducing dedicated *feature mappings*, which define how the features of target elements must be initialized. As shown in Fig. 6.3, a FeatureMapping describes a mapping from an optional source feature to a mandatory target feature. Both correspond to an EStructuralFeature, which is either an attribute or a reference (cf. Section 2.1.1). Feature mappings are accommodated by class mappings to control the initialization of the created target elements. To initialize its associated target feature, a feature mapping defines how to obtain an initialization value from the source element. In this context, a source feature indicates that its values should be mapped to the target model, serving as initialization values for the respective target feature. In the following, we refer to the values mapped from the source to the target model as *source values*. In Fig. 6.4, we augment all class mappings with corresponding feature mappings. For example, in the context of the mapping between Class and Component, the source value of the name attribute is mapped to the same-named target attribute. Moreover, the source values of the ownedPort reference are mapped to the ports reference. Note that, for bidirectional references such as ports, it is sufficient to initialize them in one direction only, whereas the corresponding opposite references like component will be implicitly initialized as well.

In the context of a feature mapping, only source values of primitive data types (cf. Section 2.1.1) can be mapped from the source to the target model directly. By contrast, in case of non-primitive types such as enumerations or classes, the source values must first be translated into *target values* that conform to the target feature. In these cases, a feature mapping must refer back to a ClassifierMapping that acts as a translator (cf. Fig. 6.3). Thus, if the types of source and target features are enumerations, the translator must be an EnumerationMapping between them. Otherwise, if source and target are references, the translator is a ClassMapping that is able to translate the source elements into conforming target elements. To reduce the complexity of Fig. 6.4, the translators are omitted. As an example, the feature mapping between ownedPort and ports uses the class mapping from Port to DiscretePort as its translator.

Under certain circumstances, the source value of a feature mapping, which must be mapped to the target model, does not originate from any specific source feature. For this reason, the source reference of a FeatureMapping is optional in Fig. 6.3. Whenever no source feature is given, the source value of the feature mapping is assumed to be the source element of the class mapping itself. This is useful in case of 1:n mappings that are used for partitioning a source element into multiple target elements (cf. Section 6.3.1). In this case, a feature mapping without an indicated source feature can be used to initialize references between two distinct target elements, for which no corresponding references exist inside the source metamodel because both target elements correspond to the same source element.

A feature mapping is only well-formed if its features satisfy the criterion of *compatibility*. For a source feature to be compatible with a target feature, they must be either both references or both attributes. If the data type of an attribute is primitive (cf. Section 2.1.1), compatibility requires the type of the other attribute to be identical. By contrast, in case of references or non-primitive attributes, the source values must be translatable into the type of the target feature. Thus, in both cases, a classifier mapping must be indicated as a translator. On the one hand, if source and target features are attributes and their types are enumerations, the translator must be a proper enumeration mapping from the source enumeration to the target enumeration. On the other hand, if source and target features are references, their types are classes. Accordingly, the translator must be a proper class mapping with respect to the subtyping relations inside the source and target metamodels. In particular, the mapping's source class must be *contravariant* and, therefore, substitutable by the class of the source reference. Thus, the mapping is guaranteed to be applicable to each source element. Conversely, the mapping's target class must be *covariant* and, therefore, able to substitute the class of the target reference. Thereby, each target element is guaranteed to conform to the target feature.

The compatibility of features also applies to their multiplicities (cf. Section 2.1.1). On the one hand, if the source feature accommodates multiple values, it is a *many* feature and therefore only compatible with another *many* feature as its associated target. On the other hand, if the source is a *single* feature that accommodates no more than one value, both *single* or *many* features can serve as compatible targets. A more detailed, numeric consideration of compatible multiplicities is left to future works.

6.4 Inference Engine

As described in Section 6.2.3, we use the manually specified type mappings to automatically infer additional feature mappings, which are needed to initialize a target model. This inference is based on the decision tree that we illustrate in Fig. 6.5. In the context of a mapping model, we apply the inference to each class mapping and to each feature of the target class that can be initialized manually (instead of deriving its values automatically from other sources). The decision-making process leads to one of five final decisions marked by ① to ⑤. To account for the accuracy demanded by requirement R1 in Section 6.2.2, the underlying heuristics of the process aim to match each target feature with a corresponding set of source features, from which the values for the initialization will be derived.

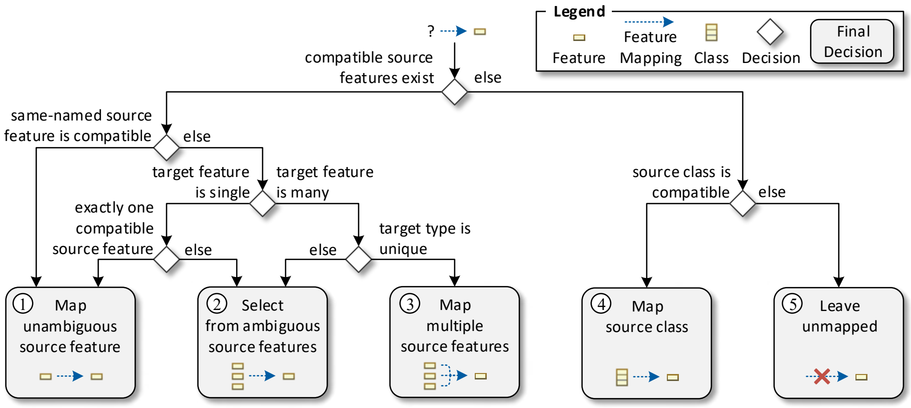

Legend
Feature
Feature Mapping
Class
Decision
Final Decision
?
compatible source features exist
else
same-named source feature is compatible
else
target feature is single
target feature is many
exactly one compatible source feature
else
else
target type is unique
source class is compatible
else
① Map unambiguous source feature
② Select from ambiguous source features
③ Map multiple source features
④ Map source class
⑤ Leave unmapped

Figure 6.5: Decision tree for the inference of feature mappings from type mappings [*GB19].

As can bee seen from the tree in Fig. 6.5, the initial decision made by our engine is to check whether there is a source feature that is compatible (cf. Section 6.3.2) with the target feature at hand. In this case, ambiguities between multiple compatible source features are still possible. Handling such ambiguities is the subject of the left-hand subtree. “Our initial tiebreaker is the name of features. On the basis of our observation that features with a corresponding information content are often given the same names, we check for the existence of a same-named source feature. If such a feature is compatible, an unambiguous mapping between the same-named features is created according to the final decision ①” [*GB19]. For example, this decision is taken to map each of the name attributes in Fig. 6.4 to its same-named counterpart. Otherwise, if there is no same-named source feature that is compatible, our decision tree takes into account the target feature’s multiplicity. As described below, we proceed depending on whether the target is a *single* or *many* feature (cf. Section 2.1.1).

Single Features. If a target feature is restricted to a single value, no more than one feature mapping must be created. If only one source feature is compatible, we again arrive at final decision ① and map the respective features unambiguously. In Fig. 6.4, we thereby infer the mapping between the type of a Property and the componentType of a ComponentPart. In contrast, if multiple source features are compatible, we remove the ambiguity by making a selection as per final decision ②, creating a single mapping for one selected source feature. In our decision tree, “we abstract from this selection, which is based on the order of features and takes into account how many feature mappings for a certain source feature have already been inferred before. In particular, we give preference to those source features that have been mapped least often. We select from these features in the order specified by their source metamodel because we also traverse and assign the target features according to their order. Thereby, we make use of our observation that the relative order of corresponding features is often consistent between source and target metamodels” [*GB19]. For example, in Fig. 6.4, initializing the componentPart of a PortPart requires a selection between the partWithPort and definingEnd of a ConnectorEnd. Whereas the latter is omitted from Fig. 6.4 because it is not mapped, both references are of type Property and therefore equally compatible.

Many Features. "In contrast, if the target is a *many* feature, it supports multiple co-existing feature mappings with different source features. However, mapping numerous compatible source features to the same target is not always the right decision to avoid information loss. Instead, there are cases in which numerous source features need to be distributed among different target features. Thus, as an additional tiebreaker in case of a *many* feature, we take into account if the type of the target feature is unique among all the *many* features of the target class. If the type is unique, then the target feature represents the only compatible candidate to map the source features to. Hence, in final decision ③, we create multiple feature mappings, one for each compatible source feature" [*GB19]. For example, in Fig. 6.4, this decision is used to initialize the ports, connectors, portParts, and connectorEndpoints references. However, in all of these cases, the number of compatible source features is restricted to one, such that only one mapping is created for each of the aforementioned target features. On the contrary, provided that the target feature's type is not unique, it is possible to distribute the source features among different target features. In this case, we again arrive at final decision ②, creating only one feature mapping by selecting from the compatible source features.

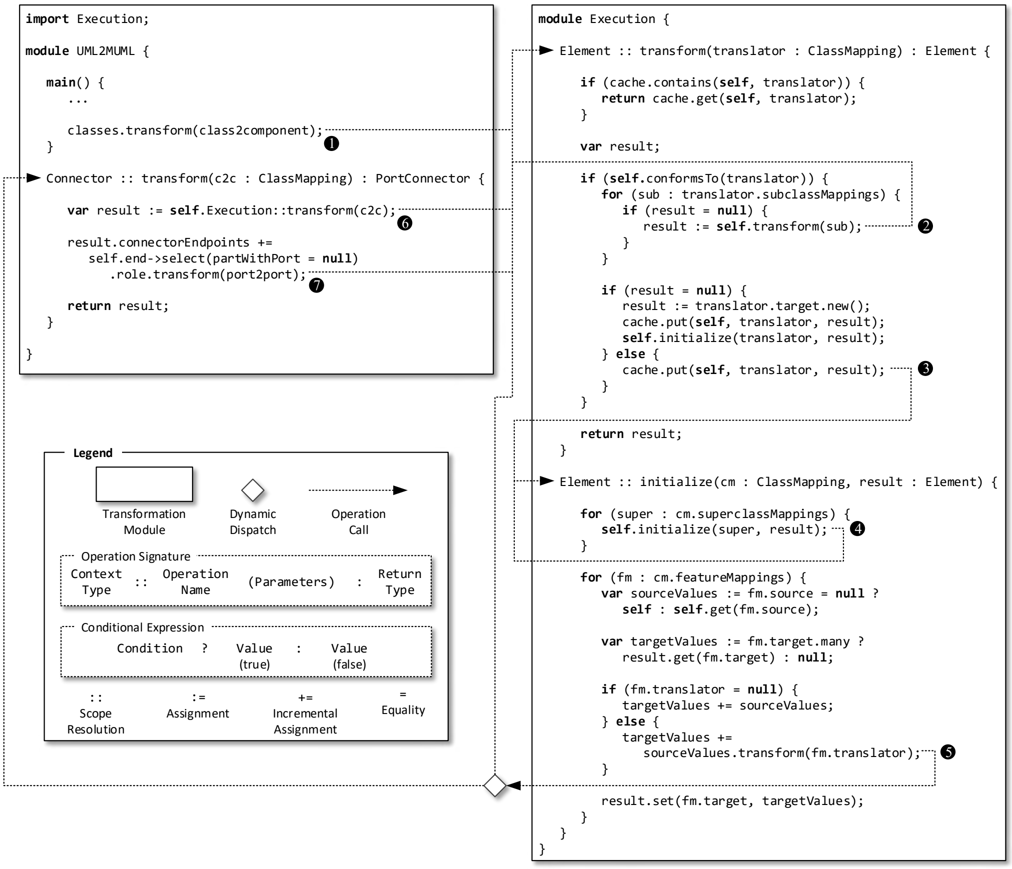
Incompatibility of Features. It is possible that none of the source features ensure compatibility with a target feature at hand. As an example from Fig. 6.4, the portType of a PortPart cannot be mapped automatically because the role of the source ConnectorEnd is of type ConnectableElement, for which there exists no mapping to Port as the required target class. In such situations, our fallback procedure is to create a feature mapping in which the optional source feature is omitted (cf. Section 6.3.2). However, this decision requires that the source class is translatable into some target class that conforms to the type of the target feature. Thus, if there exists a class mapping that could act as a translator, we arrive at final decision ④ and create a feature mapping that refers to the source class itself instead of a particular source feature. In the scope of our example, there are no translators that could be used to initialize the portType or any other uninitialized reference. Accordingly, these target features are left unmapped as per final decision ⑤.

6.5 Execution Framework

To ensure the executability demanded by requirement R4 from Section 6.2.2, we propose a *framework* [Joh97; GHJV94] for the automated execution of mapping models, as well as their manual refinement. At the core of this framework, we propose an execution algorithm that generalizes our previous work [*GSB17] on reducing the verbosity of endogenous transformations, which are based on identical source and target metamodels. In this special case, each class is implicitly mapped to itself, thereby enabling transformations to be designed according to the *implicit copy* pattern [LK14]. By contrast, we adapt the underlying concept to the general case of exogenous transformations with explicit mappings between classes from different metamodels. Section 6.5.1 introduces our execution algorithm, before we address its refinement with imperative instructions in Section 6.5.2. Finally, in Section 6.5.3, we analyze the language facilities that are needed to implement our framework.

6.5.1 Execution Algorithm

Figure 6.6 illustrates the interaction between transformation modules in the context of our execution framework. On the right, we depict a pseudo code implementation of our execution algorithm encapsulated by a module named Execution. The notation of the code is oriented towards the OCL [OMG14]. The main ingredient of our algorithm is an operation called transform, which translates a given source element into a target element. The translation depends solely on the existing source elements and will never be recursively affected by the created target elements. Benelallam et al. [BGTC18] refer to this property as *non-recursive rule application*. The transform operation is parametrized by a ClassMapping that serves as a translator (cf. Section 6.3.2). Both source and target element are typed by Element, which we assume to be the implicit superclass of arbitrary classes. Thus, Element acts both as the *context type* of transform (thereby typing the source element on which the operation is invoked by a caller) and its *return type* (thereby typing the target element returned to the caller). In Fig. 6.6, we use the scope resolution operator :: to separate the context type from the operation name, whereas : separates operations from their return types or variables from their types.



```
import Execution;

module UML2MUML {

   main() {
      ...

      classes.transform(class2component);
   }

   Connector :: transform(c2c : ClassMapping) : PortConnector {

      var result := self.Execution::transform(c2c);

      result.connectorEndpoints +=
         self.end->select(partWithPort = null)
            .role.transform(port2port);

      return result;
   }

}
```

```
module Execution {

   Element :: transform(translator : ClassMapping) : Element {

      if (cache.contains(self, translator)) {
         return cache.get(self, translator);
      }

      var result;

      if (self.conformsTo(translator)) {
         for (sub : translator.subclassMappings) {
            if (result = null) {
               result := self.transform(sub);
            }
         }

         if (result = null) {
            result := translator.target.new();
            cache.put(self, translator, result);
            self.initialize(translator, result);
         } else {
            cache.put(self, translator, result);
         }
      }

      return result;
   }

   Element :: initialize(cm : ClassMapping, result : Element) {

      for (super : cm.superclassMappings) {
         self.initialize(super, result);
      }

      for (fm : cm.featureMappings) {
         var sourceValues := fm.source = null ?
            self : self.get(fm.source);

         var targetValues := fm.target.many ?
            result.get(fm.target) : null;

         if (fm.translator = null) {
            targetValues += sourceValues;
         } else {
            targetValues +=
               sourceValues.transform(fm.translator);
         }

         result.set(fm.target, targetValues);
      }
   }
}
```

Figure 6.6: Interaction between transformation modules involved in the execution framework.

The Execution module can be imported by imperative transformations, enabling them to invoke the interpretation of a mapping model. On the left-hand side of Fig. 6.6, we illustrate such a transformation named UML2MUML, reflecting our example introduced in Section 6.2. In this context, the root elements of the UML source model are assumed to be of type Class and stored in a variable called classes. Thus, from its main operation, the transformation triggers the interpreter by invoking the transform operation on each root element. The invocation is marked by ❶. By invoking transform, each Class is passed to the operation as its source element, which is to be translated into a target element of type Component. Accordingly, the variable called class2component is assumed to represent the class mapping between Class and Component from Fig. 6.4, which is passed to the transform operation as its translator.

In the following, we describe the procedure of transform when invoked on a source Class. Within the scope of the operation, the source element is denoted by self. To simplify the presentation in Fig. 6.6, we assume that features are either references or attributes with primitive data types. Thus, we omit the handling of enumerations (cf. Section 6.3.1), which do not occur in our example. In the mathematical sense, a mapping must not map an element more than once. We ensure this behavior by applying the principle of *memoization* [Mic68]. Thus, our algorithm initially consults a cache of mapped elements, recording each triple of source element, translator, and target element. On this basis, we assume that the contains operation indicates whether a source element has already been mapped before by a given class mapping. If self has already been mapped by the translator, our algorithm returns the cached target element, which is retrieved using the get operation. Otherwise, if self has not been mapped before, the transform operation must create and initialize a new target element, which is stored in the result variable. Below, we address the creation and initialization separately.

Creation

As a prerequisite for the creation of a target element, the source element must conform to the class mapping that serves as translator. To this end, the operation conformsTo checks (i) whether the class of self substitutes the source class of the translator and (ii) whether self fulfills an OCL constraint that might be attached to the translator as a context condition (cf. Section 6.3.1). If the source element is non-conforming, the returned value of the result variable is still null, thereby indicating that the element could not be translated successfully. In contrast, if the source element conforms to the class mapping, our algorithm first checks whether the creation of the target element must be delegated to a potential subclass mapping (cf. Section 6.3.1). Thus, we traverse all subclass mappings sub according to their order inside the translator. As long as no subclass mapping has initialized the result with a value other than null, we attempt to create a target element by invoking transform recursively with the respective sub as a translator. We mark this invocation by ❷. In our example, transform might be invoked twice for both subclass mappings from Class to AtomicComponent and StructuredComponent (cf. Fig. 6.4). However, as described above, the invocation will only return an element different from null if self conforms to sub. For example, assuming that the Class denoted by self contains at least one part of type Property, the context conditions imposed in Fig. 6.4 are only fulfilled for the subclass mapping to StructuredComponent, which is therefore responsible for the creation.

Once transform has been invoked recursively, the given translator between Class and StructuredComponent does not refer to any subclass mappings itself (cf. Fig. 6.4). Hence, in the further course of our algorithm, the result variable is still null because no subclass mapping could be used to create the target element. In such a case, the translator is itself responsible for the creation. To this end, we obtain the target class of the translator (StructuredComponent in our example) and create a new element of that class using the new operation. The created element is assigned to the result variable, and the triple of self, translator, and result is immediately stored in the cache using the put operation. Next, to initialize the features of the created result, we invoke a second operation named initialize on self, passing both translator and the result as arguments. We mark this invocation by ❸. By contrast, whenever the result is no longer null because it has been previously created by a subclass mapping, the translator is not responsible for the creation. In this case, the triple of self, translator, and the pre-existing result is simply cached without invoking initialize. Accordingly, target elements are initialized locally upon their creation, a property referred to as *locality* by Benelallam et al. [BGTC18].

Initialization

Initializing a target element requires the execution of any feature mappings indicated by the class mapping that is responsible for the creation. Inside the initialize operation, the target element is denoted by result. Initially, our algorithm takes into account potential superclass mappings of the given class mapping cm. For each super, we invoke initialize recursively, thereby executing the feature mappings accommodated by superclass mappings prior to those accommodated by subclass mappings. In our example, super corresponds to the aforementioned class mapping from Class to Component, which accommodates feature mappings for the initialization of the name and ports of the target Component (cf. Fig. 6.4). We mark the recursive invocation of initialize by ❹.

After all superclass mappings have been handled, the initialize operation iterates over the feature mappings accommodated by the class mapping cm itself. Thus, for each feature mapping fm, we first obtain the sourceValues for that mapping. As described in Section 6.3.2, the source values are subject to the condition whether the feature mapping fm indicates a source feature or not. We therefore use a conditional expression to ensure that, whenever the source feature is null, there is only a single source value that corresponds to the source element denoted by self. Conversely, if fm indicates a source feature, we use a reflective operation named get to obtain the values of the source element for that feature. Next, we use another conditional expression to account for the fact that multiple feature mappings with a shared target feature may co-exist, as described in Section 6.4. Thus, if the respective target feature is a many feature (cf. Section 2.1.1), we must assume that another feature mapping with the same target has been executed before. If so, the feature might already be initialized with pre-existing targetValues, which must be preserved. Therefore, we use the aforementioned get operation to save the pre-existing targetValues. In contrast, if target is not a many feature, no more than one feature mapping for each target must exist. Accordingly, we assume that there is no pre-existing target value and therefore set the targetValues to null. Benelallam et al. [BGTC18] refer to the above assumptions as *single assignment on target properties*.

On this basis, we need to extend the existing targetValues with new values resulting from the execution of fm. If the translator of fm is null (indicating that both source and target features are primitive attributes), the sourceValues are assumed to conform directly to the type of the target feature. Thus, in this case, we use the += operator to simply add the sourceValues to the existing targetValues in an incremental fashion. For example, this is the case when executing the feature mapping between the name of a source Class and the name of a target Component from Fig. 6.4. By contrast, if the translator of fm is not null, it must be used to translate the source values into conforming target values. We carry out the translation by invoking the transform operation recursively on each of the sourceValues. In particular, whenever source is a containment (cf. Section 2.1.1), the translated sourceValues correspond to elements that are hierarchically contained by self. Thereby, we traverse the source model in a descending fashion according to the *recursive descent* design pattern for model transformations [LK14]. As a parameter for the recursive invocation of transform, we pass the translator indicated by fm. This invocation is marked by ❺. As described above, invoking transform will either create a new target element with the aid of the translator, or obtain this target element from the cache if it has been created by the translator before. In both cases, no explicit navigation through the target model is necessary to obtain the target elements needed as values for the initialization of a feature. This property is also referred to as *forbidden target navigation* by Benelallam et al. [BGTC18]. In our example, the recursion takes place when executing the feature mappings used to initialize the ports, connectors, and embeddedComponentParts of a StructuredComponent (cf. Fig. 6.4). Again, we use the incremental += operator to append the translated values to the pre-existing targetValues. Finally, we use a reflective operation named set to initialize the target feature of the result with the obtained targetValues. After all feature mappings have been processed by initialize, the transform operation returns the initialized result to its caller.

6.5.2 Imperative Refinement

The algorithm presented in Section 6.5.1 is generic in the sense that it translates arbitrary source elements into target elements depending on a given mapping model. However, as described in Section 6.2.1, there are exceptional cases in which the logic of a transformation is not adequately encoded by the declared mappings, or cannot be encoded by means of declarative mappings at all. In the context of our example, the latter case occurs for a UML Connector that is mapped to a DelegationConnector in MechatronicUML. Unlike an AssemblyConnector, the connectorEndpoints must be initialized with a DiscretePort. However, from the context of the source Connector, the corresponding Port is not directly navigable. Therefore, it cannot be simply mapped to a target element of type DiscretePort, which is required for the initialization. Instead, this Port must be obtained by iterating over each end of the source Connector in order to select a ConnectorEnd without a partWithPort, which therefore enables the relevant Port to be accessed as its role. Since such a complex operation is hard to encode in a declarative fashion, requirement R3 from Section 6.2.2 demands an imperative encoding of transformation logic. To satisfy this requirement, our execution framework enables transformation developers to refine the execution of declarative mappings by means of individual imperative instructions.

In particular, developers may refine the execution on a per-type basis, redefining the transform operation for specific source classes (subclassing the generic class Element) that require exceptional logic during their translation into target classes. Thereby, we account for the fact that types represent suitable branch conditions in the scope of model transformations [Heb+18, p. 452]. For example, the UML2MUML transformation in Fig. 6.6 redefines transform for the Connector class. Thus, the c2c parameter of the operation is assumed to represent the class mapping between Connector and PortConnector from Fig. 6.4, therefore adjusting the operation's return type to PortConnector. We use this redefinition to encode the exceptional logic that is needed to initialize the endpoints of a delegation connector as described above. To select at runtime between the different definitions of transform, our approach is based on a *dynamic dispatch* of operations. Thus, whenever transform is called on a source element, the actual type of that element determines whether the generic default implementation or a type-specific refinement is invoked. Since transform is recursively called by the generic implementation (cf. Section 6.5.1), our framework underlies the *inversion of control* principle [GHJV94]. Thus, refinements are not invoked explicitly by the transformation developer but implicitly by the framework itself, thereby inverting the control flow.

For example, in the default case, the recursive call marked by ❺ in Fig. 6.6 invokes the generic definition of transform. However, when initializing the connectors of a target Component, the translated source values are of type Connector. Consequently, the operation call is dispatched dynamically to invoke the type-specific redefinition of transform inside the UML2MUML transformation. However, a redefinition may always delegate the translation to the generic definition inside the Execution module. Accordingly, in Fig. 6.6, the source Connector denoted by self is first translated into the target PortConnector as defined by the declarative mapping model. To this end, ❻ marks the invocation of the default implementation, reusing c2c as a translator. Thus, the resulting PortConnector will be partially initialized by executing the feature mapping between end and connectorEndpoints that is accommodated by c2c (cf. Fig. 6.4). After returning the control to the redefinition, the PortConnector is stored in the result variable. If the result is an AssemblyConnector, it is already fully initialized with the corresponding endpoints of type PortPart. In contrast, if the result is a DelegationConnector, it must still be post-processed to be initialized with an endpoint of type DiscretePort, thereby requiring a refinement of the transformation logic encoded by the mapping model. We specify this refinement in the further course of the redefinition, where we iterate over each end of self and use a select operation to filter out the ConnectorEnd with a partWithPort equal to null. Subsequently, we obtain the role of that ConnectorEnd, representing the Port that can finally be translated into the DiscretePort needed for the initialization. The invocation of this translation is marked by ❼, assuming that the required translator between Port and DiscretePort is accessible through a variable named port2port. Since there is no redefinition for Port, the translation is carried out by the generic definition of transform, before using the += operator to assign the obtained DiscretePort to the connectorEndpoints of the result. Note that, in case of an AssemblyConnector, the above assignment does not have any effect because each end does refer to a partWithPort. Thus, no explicit distinction between assemblies and delegations must be made inside our redefinition. Finally, the refined PortConnector denoted by result is returned to the generic implementation, which continues the execution of the given mapping model.

6.5.3 Language Facilities

In the previous sections, we presented our framework in the form of a pseudo code implementation, thereby providing a language-independent solution for the execution and refinement of mapping models. Nevertheless, to ensure a successful implementation, a programming language must offer specific facilities that our framework is based on. In [*GSB17], we therefore extrapolated these core language facilities from our conceptual approach. Thereby, we enable practitioners to analyze the feasibility of implementing the concept using concrete host languages. In particular, the following language facilities (LF1–LF3) are mandatory:

(LF1) Module Superimposition is a mechanism that enables transformation developers "to overlay several transformation definitions on top of each other and then execute them as one transformation" [WSD10, p. 286]. In our approach, this is essential for type-specific transformation modules (such as UML2MUML) to be superimposed upon the Execution module, thereby avoiding code duplication of the generic transform operation. Note that, in general, module superimposition does not require a transformation module to explicitly import another module, unless specific contents from that module are directly accessed [WSD10, p. 302]. However, since our approach is based on a direct call of transform (cf. Fig. 6.6), an import of the Execution module is required.

(LF2) Dynamic Dispatch of operations must be supported to select at runtime between the generic default implementation of the transform operation and type-specific refinements of itself. In particular, the selection must be made on the basis of the actual runtime class of the element upon which the operation is called. Thus, as a prerequisite, it must be possible to redefine one and the same operation for different context types, i.e., operations must be *polymorphic*. Note that, although operation *overriding* is considered as a central aspect of module superimposition [WSD10], our approach does not strictly depend on the ability to override the generic transform operation. Overriding is one possible way to ensure that operations are dispatched dynamically, provided that the implementation of classes can be changed. If so, the generic implementation inside the Element class can be overridden inside specific subclasses. However, in the general case of unchangeable classes, dynamic dispatch can also be achieved by other, more broadly applicable mechanisms such as the *visitor* pattern [GHJV94].

(LF3) Reflection [Smi82] is an indispensable facility that must be provided by the underlying environment used for metamodeling. This requirement arises from the fact that the algorithm presented in Section 6.5.1 invokes multiple reflective operations either on particular classes from the source or target metamodels, or on the model elements as instantiations of these classes. In particular, the algorithm creates new elements of a given target class by means of a reflective new operation. Furthermore, the algorithm requires each model element to provide reflective access to its values for specific features, which is the purpose of the get operation. Conversely, our algorithm also requires a reflective mechanism such as the set operation, which is used to initialize the values of elements for specific features. The application of reflection to model transformations has been previously investigated by Kurtev [Kur10].

Table 6.1: Comparative overview of facilities offered by transformation languages.

Language Facility		QVTo	ATL	K3	ETL	Xtend
(LF1)	Module Superimposition	✓	✓	✓	✓	✓
(LF2)	Dynamic Dispatch	✓	✓	✓	✓	✓
(LF3)	Reflection	✓	✓	✓	✓	✓

As demanded by requirement R5 from Section 6.2.2, LF1–LF3 are general-purpose facilities available in state-of-the-art imperative languages. To support this claim, we conduct a feasibility analysis in which we check existing transformation languages against the facilities. From the languages surveyed in [KG17], we include only those that support imperative instructions, thereby excluding purely declarative ones. As can be seen from Table 6.1, we take into account the imperative QVTo [OMG16] and the hybrid ATL [JABK08] as the most widespread, de-facto standards for model transformations [Bat+16; BCG19b]. Furthermore, we consider Kermeta 3 (K3)[1] and the Epsilon Transformation Language (ETL) [KPP08] as further imperative languages. In addition to the above special-purpose languages, we also address Xtend[2] as a general-purpose language that "lends itself well to creating model transformations" [Heb+18, p. 447]. Below, we discuss each of the facilities separately.

With respect to LF1, module superimposition has been explicitly built into ATL [WSD10]. However, the other languages are equipped with alternative mechanisms that can be used to import language modules into each other, thereby enabling different transformations to share the generic default implementation of our execution algorithm. Accordingly, LF1 is offered by all of the considered languages, as shown in Table 6.1. The facility LF2 is supported natively by QVTo, ATL, and ETL. All of these languages enable operations to be defined in the context of arbitrary classes. At runtime, the invoked definition is selected depending on the actual class of the element that an operation is called upon. In contrast, Xtend explicitly supports *multiple dispatch*. Therefore, the language enables the context element to be treated as a default operation parameter. On this basis, operations are invoked dynamically according to the runtime class of the elements passed as a parameter. Since K3 is based on Xtend, it makes use of the very same mechanism to support dynamic dispatch. In summary, each of the considered languages provides LF2, as indicated by Table 6.1. Regarding LF3, ATL is the only language with a reflective application programming interface (API) for accessing or initializing specific features, or for invoking specific operations of model elements. However, all other languages provide access to the reflective API offered by EMF as the underlying metamodeling environment. Therefore, in Table 6.1, we label LF3 as provided by each of the languages. In summary, as of today, each facility is properly offered by all of the languages taken into account. Accordingly, we conclude that our execution framework can generally be implemented in a wide range of host languages with imperative features.

[1] http://www.kermeta.org

[2] https://www.eclipse.org/xtend

6.6 Case Studies

In this section, we conduct case studies to evaluate our contributions with respect to the requirements specified in Section 6.2.2. Whereas the imperative logic (R3) and the executability (R4) are functional attributes covered by the execution framework presented in Section 6.5, the feasibility of implementing the framework using general-purpose language facilities (R5) has already been validated in Section 6.5.3. We therefore focus on the quantitative evaluation of R1 and R2 by addressing the following research questions (RQs):

RQ1: *How accurately can declarative feature mappings be inferred?*

RQ2: *How effectively can the verbosity of imperative transformations be reduced?*

In the following, we rely on the reporting guidelines for case studies by Runeson and Höst [RH09]. Accordingly, to illustrate the design of our studies, we first elaborate on the selection of cases in Section 6.6.1, before addressing the data collection in Section 6.6.2 and our analysis of this data in Section 6.6.3. Subsequently, we present the results obtained from the analysis in Section 6.6.4 and discuss threats to the validity of these results in Section 6.6.5.

6.6.1 Case Selection

Our studies refer to five different transformation cases. In the following, we describe the selected cases and the underlying selection criteria. First, we address a transformation in which models of a media library are translated between different syntactic representations. The origin of this case is the work by Freund and Braune [FB16], which served as a reference point in Section 6.2.1. By selecting this case, we aim to confirm that our contributions alleviate the identified verbosity problems. In the following, this case is referred to as *Lib2Lib*. The second selected case is another syntactic translation of participants involved in an educational institution [FV09]. We refer to this case as *Edu2Edu*.

In the above cases, the source metamodels closely match with the target metamodels because both represent the same semantic content, differing in its syntactic representation only. To ensure that our studies are less biased by such a close match between source and target metamodels, the next selected case involves metamodels that do not bear a direct semantic relationship. This case is referred to as *Class2ER*, addressing a translation between class diagrams and entity-relationship (ER) diagrams [Wim+10]. Since this case involves a higher structural heterogeneity (cf. Section 6.2.1) compared to the previous ones, it entails an advanced complexity of the transformation logic to be encoded.

Although the three above cases have been selected from the scientific literature, each of them has the characteristics of a toy problem. Thus, to increase the relevance of our studies, the two residual cases are transformations that have been practically applied in the scope of this thesis. On the one hand, the fourth selected case is the translation of active structures given as UML class diagrams into MechatronicUML component architectures, as described in Section 6.2. The reason for selecting this case, which we refer to as *UML2MUML*, is that the underlying source and target metamodels have not been synthesized ad hoc. Instead, they are real-world metamodels that are practically relevant even outside the studied transformation scenario. Thereby, we aim to reduce the risk that the obtained results could be biased by synthesizing metamodels, as is the case with the prior toy problems. On the other hand, the fifth selected case is the automata construction from Section 5.5. Unlike the above exogenous scenarios, this transformation is endogenous because it uses Uppaal timed automata as a common source and target metamodel. With respect to RQ1, this case only acts as a trivial *sanity check* because each feature must be unambiguously mapped to itself as a same-named counterpart (cf. Fig. 6.5). This case is referred to as *TA2TA*.

6.6.2 Data Collection

Building on the analysis of language facilities from Section 6.5.3, we select QVTo [OMG16] as a host language to provide an implementation of our execution framework, which is given in Appendix B. Moreover, we also provide an imperative reference transformation written in QVTo for each case. These reference transformations act as a benchmark (i) for the transformation logic underlying the individual cases and (ii) for the size of the resulting transformation definitions encoding this logic. In addition, we also create a mapping model for each case, whereas the respective class mappings are extrapolated from the imperative operations inside our reference transformations. Whereas the mapping models for *Lib2Lib* and *Class2ER* are based on plain class mappings, the residual cases involve superclass and subclass mappings (cf. Section 6.3.1). Furthermore, the *UML2MUML* case involves conditional mappings, which are restricted by OCL constraints (cf. Section 6.3.1).

In the next step, as a point of reference for our inference engine, we enhance the mapping models manually by adding the respective feature mappings. Again, we extract these mappings from the imperative operations in our reference transformations. However, in general, not all parts of the transformation logic can be defined by means of class and feature mappings. Hence, on the basis of our execution framework implemented in QVTo, we extend the reference mapping models with imperative refinements as described in Section 6.5.2. Thereby, we create hybrid transformation definitions, which are intended to reflect the same transformation logic as the imperative reference transformations. Please note that the *Edu2Edu* case is special as it does not require such an imperative refinement at all and can therefore be defined solely in term of declarative mappings. Nevertheless, like all other cases, it still requires imperative glue code to trigger the execution of the mapping model (cf. Section 6.5.1). To confirm that the hybrid transformation definitions are in fact equivalent to the imperative reference transformations, we execute both of them with a test input model and match the resulting output models against each other.

Subsequently, we take the manual class mappings as a basis for the application of our inference engine (cf. Section 6.4), using its heuristics to infer the feature mappings for each case automatically. We assess the accuracy of the produced results by comparing the inferred mappings against the mappings defined manually in the prior step. Furthermore, to assess the effective reduction of the imperative instructions, we measure the number of source lines of code (SLOC) that were needed by the imperative reference transformations and by the corresponding imperative refinements of the individual cases.

6.6.3 Analysis

In the following, we describe how we analyze the collected data to answer our research questions. To analyze the accuracy of our inference engine, we compare the feature mappings inferred automatically against the reference mappings declared manually. “We regard an inferred feature mapping as a *true positive* if the same mapping is also present inside the reference mapping model. A *false positive* is an inferred mapping that is not part of the reference mappings. On this basis, we measure precision and recall in order to quantify how many of the inferred mappings are actually correct, and how many required mappings could be correctly inferred” [*GB19]. Furthermore, to analyze the effective reduction of the imperative verbosity, we compare the size of each reference transformation against its corresponding refinement and calculate the portion of SLOC that could have been saved.

6.6.4 Results

Table 6.2 summarizes the obtained results of the data collection and analysis. The left part shows the amount of class and feature mappings declared as part of the reference mapping models. The center part of the table illustrates the inference of feature mappings. On the right, we give an overview on the amount of imperative instructions. Moreover, we also indicate the equivalence of the execution results between the purely imperative reference transformations on the one hand, and our hybrid transformations on the other hand.

According to our findings, all feature mappings required in case of *Lib2Lib* and *Edu2Edu* could be successfully inferred, leading to an optimal value of the recall for both cases. Unsurprisingly, the recall of the *TA2TA* case was optimal as well due to the identity of the source and target metamodel. The recall of the two residual cases was suboptimal, however, the values of 89% and 75% still suggest that large amounts of the manual work could potentially be saved. For the reasons stated above, its is also unsurprising that the *TA2TA* case achieved the highest possible precision. Whereas the precision measured for the *Edu2Edu* case was optimal as well, the residual cases each showed a number of false positives. For the *Lib2Lib* and *Class2ER* cases, this number was small enough to conclude that the vast majority of mappings was inferred correctly with a precision of 83% and respectively 89%. As opposed to these results, the precision achieved for the *UML2MUML* case was only 41%. Accordingly, more than half of the mappings inferred by the automated engine would need to be removed or adjusted manually be the transformation developer.

Table 6.2: Results of the conducted case studies [adapted from *GB19].

	Mappings		Inference				Instructions			
Case	**Class**	**Feature**	**True Positives**	**False Positives**	**Precision**	**Recall**	**Reference SLOC**	**Refinement SLOC**	**Reduction**	**Equivalent Execution**
Lib2Lib	5	10	10	2	83%	100%	28	15	46%	✓
Edu2Edu	4	10	10	0	100%	100%	27	11	59%	✓
Class2ER	5	9	8	1	89%	89%	40	29	28%	✓
UML2MUML	11	12	9	13	41%	75%	52	34	35%	✓
TA2TA	81	110	110	0	100%	100%	1278	839	34%	✓

With respect to RQ1, the recall did not fall below 75% in any of the cases that were studied. This result suggests "that the accuracy of our approach is essentially profitable because the majority of required feature mappings could be automatically inferred. However, as indicated by the precision, the actual benefit for transformation developers is apparently depending on the characteristics of the concrete source and target metamodels at hand. In particular, transformations involving large, industrial-scale metamodels like UML provide manifold opportunities for mapping non-corresponding, yet compatible features in an incorrect way, leading to a loss of precision" [*GB19]. In cases like *UML2MUML*, developers are therefore required to make an extra effort because the results of the inference must be post-processed and double-checked for false positive mappings. Future works should aim to reduce this number of mappings being inferred incorrectly. For example, this could be achieved by excluding untranslatable features from the inference engine explicitly, or by limiting the set of class mappings that can act as a translator for particular features.

In terms of imperative instructions, the findings in Table 6.2 suggest that the transformation definitions could be effectively reduced in each case. Since *Edu2Edu* does not require an imperative refinement at all, this case benefits from the highest reduction by 59%, whereas its remaining eleven SLOC are glue code that is needed to invoke the interpretation of the mapping model. Similarly, since *Lib2Lib* is based on metamodels with a close resemblance as well, its imperative verbosity could be reduced by 46%. In contrast, *Class2ER*, *UML2MUML* and *TA2TA* only achieved reductions between 28% and 35% due to the fact that more imperative instructions are needed to overcome a greater structural heterogeneity (cf. Section 6.2.1) compared to the other cases. Nevertheless, with respect to RQ2, our studies showed that we were able to reduce the effective verbosity of the imperative transformation definitions to a considerable amount. However, note that the effective reduction indicated in Table 6.2 does not necessarily imply an improved efficiency in terms of the overall development effort. The reason is that reducing the amount of manually specified instructions may come at a price of inaccurate feature mappings inferred by our automated approach, as it was shown in the context of RQ1. This inaccuracy may burden developers with an extra effort for correcting the inferred mappings, which is beyond the scope of RQ2.

6.6.5 Validity

As previously described in Section 5.6.5, we distinguish between various types of validity [RH09]. First, with respect to *construct validity*, the results for RQ2 are threatened by our choice of SLOC, which might not be a representative metric to measure the verbosity of imperative transformations at all. Whereas our studies are based on the assumption that the single lines are similarly verbose, the actual verbosity might vary from line to line, requiring developers to put a different amount of effort into each single line. Furthermore, it is important to point out that the imperative verbosity does not fully reflect the mental development effort. First, in addition to the manual encoding of the imperative refinements, developers must also double-check the inferred feature mappings, which increases their cognitive load. Second, inaccurate inference results as detected for RQ1 must be manually corrected. Both issues imply an extra effort that is not taken into account by RQ2.

“The *internal validity* of our studies is affected by the manual specification of reference transformations. Instead, the results of our studies could be less biased when using predefined sets of transformation definitions like the ATL transformation zoo [Kus+13] or a dedicated benchmark for model matching [WL13] as a ground truth. Furthermore, the results of our studies might not be representative for other cases, thereby affecting the *external validity*. In particular, the library case [FB16] was known to us prior to the development of our solution [. . .]. Therefore, our approach could have been tailored specifically to this case, whereas more general characteristics of other cases might have been disregarded” [*GB19].

To promote the *reliability* of our results, we enable their replication by making the collected data publicly available [*Ger20b]. Our data set includes the mapping and example models of all cases, the QVTo-based implementations of our inference engine and the execution framework, as well as the imperative reference transformations and refinements for each case.

6.7 Limitations

Correspondences. As described in Section 6.3.1, our mapping models are currently restricted to 1:1 and 1:n correspondences, thereby enabling a transformation to copy or partition the affected model elements [Wim+10]. In contrast, further types of correspondences like n:1 or 0:1 are not supported. Accordingly, mapping models cannot be used to specify that elements must be merged or generated from scratch. Enabling the definition of advanced correspondence types could further reduce the amount of imperative transformation instructions. However, on the downside, it is more challenging to infer these advanced correspondences accurately, as compared to the basic correspondences underlying our current approach.

Accuracy. Our inference engine presented in Section 6.4 could be improved with respect to the accuracy of the inferred mappings. First, to prevent target features from being left unmapped (cf. Fig. 6.5), a workaround is to extend the search space for compatible source features. This could be achieved by navigating from the source element of a class mapping to other, referenced source elements that provide a compatible source feature. In particular, this form of source navigation could also be used to define the aforementioned n:1 correspondences because it enables features of multiple source elements to be mapped to features of a single target element. Second, the accuracy of the inferred feature mappings could also be improved by considering the numeric multiplicities of source and target features as an additional criterion for their compatibility. Third, as already described in Section 6.6.4, particular features could be excluded from the inference explicitly, thereby avoiding the inference of false positive mappings. For the same purpose, the scope in which individual class mappings may act as translator could be limited to particular features, thereby preventing other features from being translated wrongly. Furthermore, additional techniques to infer more accurate transformations include *machine learning*, as recently done in [BCG19a], and *search-based MDE* [BSA17] as a means to trade off alternative inference results against each other. Finally, another different approach towards improving the accuracy is to reduce the inference of metamodel mappings to a more general matching problem, e.g., from well-researched fields like *schema matching* or *ontology matching* [IV11].

Confidence. Given the fact that the inference results are potentially inaccurate (see above), another limitation is that our engine does not inform transformation developers about the degree of accuracy. Thus, developers are uncertain about the *confidence* they should have in the inferred mappings. The indication of confidence in model transformation rules was recently pioneered in [BBMV18], whereas *Bayesian inference* has been used as one concrete method to reason about uncertainty in the context of models [HP14]. In future works, such an approach could be an initial step towards *explainability* [Köh+19] of the inference. Explanations of the inference results could help balance the manual effort that developers must make to double-check their accuracy, as discussed in Section 6.6.4.

Language Integration. Due to the hybrid nature of our approach presented in this chapter, the transformation definitions are divided into a declarative and an imperative part. Therefore, clarity and comprehensibility of transformation definitions are affected. A possible future improvement is to integrate our inference engine directly into an imperative transformation language. Using such an integration, it could be sufficient for transformation developers to declare transformation rules including source and target types only, whereas the declared rules are implemented automatically by generating the initialization instructions for the respective target features. Thereby, users of imperative languages would no longer be forced to involve a declarative approach, thereby increasing the applicability of our inference engine.

Interpretation. The execution algorithm for mapping models proposed in Section 6.5.1 is designed as an interpreter that is based on a single generic operation. A limitation induced by this design decision is that refinements can only be specified per source type on the basis of dynamic dispatch, as described in Section 6.5.2. Thus, in case of 1:n mappings with a common source type, a potential distinction between the different target types must be made explicitly as part of the refinement, leading to additional amounts of imperative instructions. In contrast, an alternative approach towards the execution is to synthesize an imperative transformation from a declarative mapping model. This kind of synthesis is also referred to as a *higher-order transformation (HOT)* because the source and target artifacts are themselves transformation definitions [Tis+09]. In our case, a synthesized transformation definition could provide one imperative operation for each declarative type mapping. Concerning the limitation to per-type refinements, these individual operations have a beneficial effect because they can be overridden independently of each other. Thereby, a HOT-based execution enables a more fine-grained specification of refinements on a per-mapping basis, without requiring additional instructions to distinguish between different target types.

Co-Evolution. Despite the fact that we address exogenous model transformations in general (cf. Section 6.2), our approach can also be used with more specific transformation intents such as *migration* [Lúc+16]. For example, *co-evolution* refers to the approach of migrating a model to an evolved version of its metamodel [HKB17]. In case of a single evolutionary step, source and target metamodels are often broadly similar. Hence, whereas our inference engine is currently restricted to feature mappings, the similarity of source and target metamodels might even enable an accurate inference of type mappings, e.g., between all those classifiers that remain unchanged during the evolution. We have prototyped the application of our execution framework to this scenario in a bachelor thesis [Som18].

6.8 Related Work

In literature, the most general approach for reducing the verbosity of model transformations is *reuse*. We refer the reader to [Bru+20; CFSS16; Kus+15] for an overview of the field. In this section, we focus on approaches that are more closely related to our work because they rely on mapping models as transformation definitions (cf. Section 6.8.1), or generate transformation definitions automatically (cf. Section 6.8.2).

6.8.1 Mapping Models

Lopes et al. [LHBJ06], Guerra et al. [Gue+13], as well as Bollati et al. [BVJM13] all use mappings between metamodels to define model transformations. Similar to our approach, these works also enable the specification of context conditions to filter the application of a mapping. Both [LHBJ06] and [BVJM13] use a HOT to generate an executable transformation definition from the mapping models. Thereby, these works differ from our approach of interpreting the mapping models by means of an execution algorithm. Diskin et al. [DGC17] propose a formal approach in which so-called *meta-traceability links* between the elements of source and target metamodels serve as a mapping model. To execute the transformations, the authors apply formal operations from the area of category theory. In contrast to our work, all of the above approaches require a manual definition of mapping models, whereas a reduced development effort by means of an automated inference is beyond their scope.

Wimmer et al. [Wim+10] provide developers with a set of operators to define mappings between the elements of source and target metamodels. The proposed mapping operators reflect recurrent correspondences and thus can be reused as building blocks of custom transformations. Again, unlike our work, the authors use a HOT to generate an executable transformation definition. The general intent of the approach is to reduce the development effort for model transformations and is therefore similar to the goal of our work. Nevertheless, developers must tailor the provided operators to their metamodels at hand. Thus, opposed to our approach of inferring mappings automatically, features still need to be mapped manually.

In contrast to the approach of executing mapping models by means of a HOT, Freund and Braune [FB16] use an execution algorithm as an interpreter, thereby resembling our work. Their algorithm ensures coherence of target models, even if the links between target elements have only been partially defined by the developer. In this respect, the goal of the algorithm is similar to the one of our inference engine. As a conceptual difference, our work separates the concerns of inference and execution, whereas the proposed algorithm combines both responsibilities. In general, the presented approach is affected by two crucial limitations, both of which we address in our work. First, the proposed algorithm only ensures that the syntactical constraints of the target metamodel like multiplicities are satisfied. As opposed to this, we match corresponding source and target features to preserve the semantic content of models during a transformation. Second, the authors do not enable developers to enrich mapping models with imperative instructions, which might be required to correct or complete the decisions made by the execution algorithm. In comparison, we adopt a hybrid approach by supporting an imperative refinement of declarative mapping models.

If declarative mappings are insufficient to fully encode the transformation logic, the use of a HOT enables developers to customize the generated transformations, e.g., at the level of imperative instructions. Nevertheless, none of the above works address such an imperative refinement on a systematic basis, as we do in this chapter. Furthermore, only Freund and Braune [FB16] use automated decision making to reduce the number of mappings created manually. However, these decisions are made on the fly during execution and are not permanently reflected in the mapping models. Therefore, in Section 6.8.2, we will focus more specifically on automated approaches generating permanent transformation definitions.

6.8.2 Model Transformation Generation

A survey of approaches to the generation of model transformations is given in [BDB16]. In general, such approaches can be classified in two major categories, generating transformation rules either *by models* or *by metamodels*. We illustrate this categorization in Fig. 6.7. For example, a subfield of *search-based MDE* [BSA17] is *model-driven optimization*, using models to encode optimization problems, which are solved by means of search techniques [Joh+19]. One possible approach is a *rule-based encoding*, which intersects with the field of model transformation generation *by models* because solutions are offered as sequences of transformation rules that must be applied to optimize a given model. Since such optimizations are endogenous transformations, the approach differs from our work with a more general focus on exogenous translations.

Another field in the first category is *model transformation by example* [Kap+12; Var06], which uses exemplary pairs of source and target models to infer transformation rules. On the one hand, in case of a *demonstration-based* approach as adopted by Kehrer et al. [KAH17], a transformation is exemplified by changing a model manually. These exemplary changes are recorded and used to infer generalized transformation rules automatically. Unlike our work, this approach primarily addresses endogenous transformations. On the other hand, in case of a *correspondence-based* approach, examples are given pairwise in terms of pre-existing source and target models, thereby extending the scope to exogenous translations. Corresponding elements of the example models must be indicated manually and are used to infer generalized correspondences between the respective metamodels [Kap+12, p. 206]. Although the inferred correspondences resemble our mapping models, corresponding features must still be mapped manually at the level of the example models. In contrast, our inference engine aims to map corresponding features automatically on the basis of manually specified correspondences between types. As the main difference between our approach and model transformation by example, we do not require the existence or creation of concrete example models.

Previous works on model transformation by example are increasingly based on search techniques [MSS18; GKHE18; BS16; KSBB12], thereby intersecting with the aforementioned field of search-based MDE. Recently, this *cognification* [CCBG18] has been pushed forward by Burgueño et al. [BCG19a] who use neural networks to learn transformations from sets of example source and target models. As an advantage over our work, this approach does not require developers to specify any correspondences manually. Nevertheless, on the downside, a sufficiently large number of example models must be available as a training set.

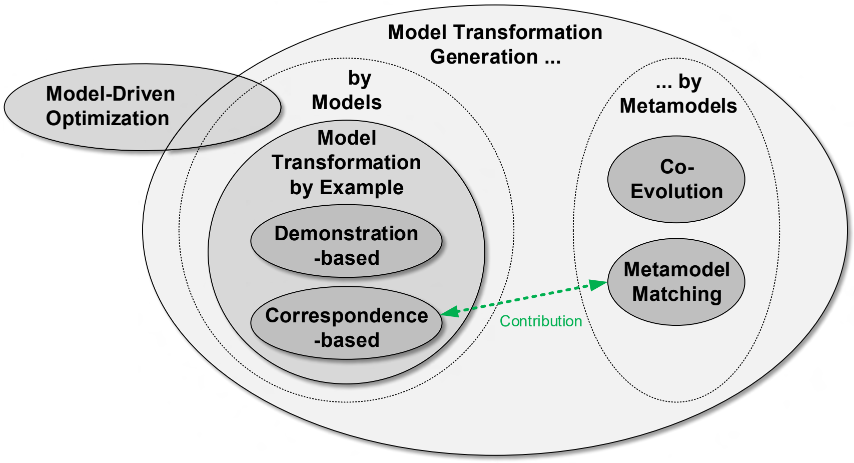


Figure 6.7: Overview of related works from the area of model transformation generation.

The second category depicted in Fig. 6.7 includes approaches towards model transformation generation *by metamodels*. Since the inference engine presented in Section 6.4 is based on mappings between metamodels, our work falls into this category as well. As another approach, Kehrer et al. [KTRK16] derive transformations for the editing of models from the respective metamodels. These transformations ensure that the edited models are consistent with the constraints of the metamodel such as multiplicities. Again, since editing transformations are endogenous, they differ from the exogenous translations addressed in our work.

Co-evolution [EKSS18] is another field in which model transformations are generated *by metamodels*. As surveyed in [HKB17], approaches like the one presented in [KSW19] aim to compensate for evolving metamodels by co-evolving the associated models. Recently, related works also address the co-evolution of metamodels and model transformations [RIKD18; KKE18; KSW18]. Both approaches derive rules for the migration of artifacts from the changes between different versions of a metamodel. In contrast, we address completely exogenous transformations without an evolutionary relationship between metamodels.

Finally, model transformation generation *by metamodels* also subsumes the area of *metamodel matching* [LFH14]. This approach closely relates to our work because mapping models are inferred by matching similar metamodels against each other. “To this end, the authors take not only structural but also linguistic similarity measures into account. Thus, their works differ from our approach which primarily analyzes the compatibility of features, using same names only as a tiebreaker. Metamodel matching increases the level of automation compared to our work because we require developers to manually indicate a set of correspondences between types. However, the matching approach is limited to metamodels with a certain degree of similarity” [*GB19], which enables a fully automated inference of accurate matches. In contrast, our semi-automated approach ensures that the inferred feature mappings are compatible with respect to the type mappings defined manually, even if the underlying metamodels are dissimilar due to structural heterogeneity (cf. Section 6.2.1).

Unlike our work, approaches from the two aforementioned fields of co-evolution and metamodel matching have been combined with techniques from search-based MDE [KSW19; KSW18; Kes+14]. In contrast, our novel contribution depicted in Fig. 6.7 is that we pre-configure the automated inference (as known from metamodel matching) with manual correspondences[3] (as used in the area of correspondence-based model transformation by example). Thereby, we enable transformation developers to generate model transformations semi-automatically, even in cases where the dissimilarity of metamodels does not allow for a fully automated generation. For future works, we propose an in-depth comparison of our semi-automated approach against the existing, more highly automated works on model transformation generation. The goal of such a comparison is to trade off the accuracy of the generated transformations against the remaining manual effort that developers must make.

6.9 Summary

In this chapter, we presented a hybrid model transformation approach that addresses the verbosity problems raised by existing transformation languages. First, on the basis of transformation models comprising declarative type mappings, we proposed an engine for the automated inference of additional feature mappings. Compared to imperative languages, we thereby reduce the amount of boilerplate instructions that developers must specify manually to initialize the features of models. Second, we presented a framework for the execution of these mapping models, which enables developers to refine a transformation with imperative instructions. Compared to declarative languages, we thereby prevent alternative or exceptional transformation logic from being encoded by large-sized sets of transformation rules. In addition, we explicitly identified the underlying language facilities that imperative languages must provide to serve as a host language for the implementation of our execution framework.

We conducted five case studies, in which we analyzed the accuracy of the declarative mappings inferred automatically and the reduction of the verbosity as measured by the imperative instructions specified manually. Our findings suggest that the majority of mappings can actually be inferred, thereby reducing the verbosity to a considerable degree. However, our accuracy assessment also showed that the precision of the inference decreases in case of transformations between dissimilar metamodels with a higher structural heterogeneity.

Our semi-automated approach is beneficial for transformation developers because it enables a reduced verbosity even in case of heterogeneous metamodels, for which transformations cannot be inferred fully automatically. First, developers can pre-configure the inference with type mappings, thereby ensuring syntactic validity of the inferred feature mappings. Second, developers may refine the mappings with imperative instructions, thereby accounting for transformation logic that requires an imperative style of encoding. Finally, the identified language facilities enable practitioners to analyze the feasibility of implementing our execution framework in concrete imperative host languages. By conducting such a feasibility analysis, we observed that a range of common transformation languages qualify as a host.

[3] Lafi et al. [LFH14] report that some approaches for metamodel matching enable a specification of initial mappings. However, to the best of our knowledge, these mappings do not take effect on the matching results.

7

Conclusion

We conclude this thesis by summarizing our scientific contributions with a focus on the obtained results and their benefits from an engineering viewpoint. We give this summary in Section 7.1. Finally, in Section 7.2, we briefly discuss future research perspectives suggested by our contributions.

7.1 Summary of Contributions

Due to their high degree of interconnection, next-generation CPSs must satisfy particular security requirements to achieve protection goals like confidentiality or integrity of the exchanged information. Therefore, dedicated security countermeasures must be adopted during the engineering to make systems *secure by design*. One possible countermeasure is to control the information flow security [Man11], using dedicated formal methods to analyze systems with respect to confidentiality and integrity requirements. Nevertheless, a major unresolved problem is that the theory of information flow security suffers from a limited scope of applicability in engineering practice, especially when targeting complex, discipline-spanning application domains like CPSs. In this thesis, we offered a solution to this problem by integrating formal methods for information flow security into a model-driven design method for CPSs, including Consens for the discipline-spanning systems engineering [Ana+14a] and MechatronicUML for the discipline-specific software engineering [*Bec+14].

Our first contribution is a specification technique for security policies as an extension of Consens, thereby front-loading the security requirements engineering to the initial, discipline-spanning stage of systems engineering. To prevent an underspecification of security requirements, we turned two of the partial models of Consens into flow policies known from the theory of information flow [Man03]. The resulting security policies enable systems engineers to document unauthorized information flows of a system under development and to refine the documented flows when decomposing a system into subsystems. Our contribution includes a validity check for these refinements, which formally assures engineers that an initial, system-level security policy is refined at the subsystem-level such that the specified requirements are preserved. We evaluated our contribution on the basis of a systematic quality framework for security methodologies [UFF18]. The evaluation identified the early consideration of security aspects and their formal assurance as main benefits of our work.

As our second contribution, we presented a set of architectural well-formedness rules for component-based security policies in the scope of the MechatronicUML component model. The proposed rules guide software architects both during the translation of flow policies into component-based policies and during the refinement of these policies on decomposition of components into subcomponents. In particular, a well-formed refinement ensures that the security restrictions imposed by the security policy of a component are correctly enforced by the policies of the constituent subcomponents. Thus, refining security policies according to our rules enables architects to reason locally about the security of components, preventing the emergence of global information flows when individual components are assembled. We thereby tackle the well-known composability problem of information flow security [Man02], providing architects with a guarantee that an architecture is globally secure whenever it is composed of locally secure components. We gave formal evidence for this composability while taking into account the asynchronous communication and the real-time behavior underlying MechatronicUML, thereby evaluating the soundness of our contribution.

The third contribution made in this thesis is an automated verification technique for the information flow security of MechatronicUML components, which are real-time systems whose behavior is given in the form of timed automata [AD94]. To verify that a component adheres to its predefined security policy, we applied the concept of self-composition [BDR11] to check whether the component behavior is properly refined by a variant of itself with restricted access to security-critical information. We thereby reduce the verification to a refinement check [HBDS15], which enables us to apply the off-the-shelf model checker Uppaal instead of implementing proprietary verification algorithms. The proposed verification technique is timing-sensitive [KWH11] by taking the specified real-time behavior of components into account. As a result, software engineers benefit from the ability of our technique to detect timing channels [BGN17]. If such channels remain undetected, they enable attackers draw security-critical conclusions from the timing of messages passed by a component. Furthermore, in combination with the composability demonstrated as part of the second contribution, the proposed technique enables a compositional verification in which the security of an overall architecture follows directly from the verified security of single components. We successfully evaluated the accuracy of our verification technique using a security-oriented extension of the community case study CoCoME for component-based systems [GH17].

Finally, at the base level of MDE, our fourth contribution tackles the verbosity of model transformations, which served as an enabling technology for the translation and verification of security policies within the scope of the previous contributions. To reduce this verbosity and the associated effort to develop transformations using existing transformation languages [Heb+18], we relied on the concept of declarative transformation models [Béz+06]. On this basis, we proposed an inference engine that enables developers to semi-automate the definition of model transformations, using a basic transformation model to infer additional repetitive boilerplate declarations. In addition, we proposed a framework that executes the inferred transformation models automatically, but also enables developers to refine the execution manually with imperative, type-specific instructions. We pointed out the core language facilities required to implement our framework and conducted five case studies to evaluate the reduced verbosity as well as the accuracy of the inference engine.

In summary, we extended the model-driven design method with techniques for the specification, refinement, and verification of security policies. Thereby, we enable a consistent handling of security aspects across systems and software engineering. Whereas the design method was previously restricted to the safety of CPSs, we integrated a complementary focus on security engineering, thereby promoting the overall *resilience* of systems [TSHL16].

7.2 Future Perspectives

The contributions summarized in Section 7.1 give rise to various perspectives of future research, which have already been discussed by chapter (cf. Sections 3.7, 4.8, 5.7 and 6.7). In the following, we recap the most crosscutting challenges from a more general viewpoint.

Security Requirements Engineering: Whereas our specification technique from Chapter 3 enables the early *documentation* of security requirements and the *validation* of their refinements, other requirements engineering activities are beyond the current scope of our work. For example, this limitation applies to the *elicitation* of security requirements. Therefore, as also pointed out by the evaluation conducted in Section 3.6, an upstream assessment of security threats is a promising future extension, which could enable engineers to elicit security requirements in a systematic way. However, as already discussed in Section 3.7, performing a threat analysis at the level of MBSE is challenging due to the abstraction of platform-specific implementation details, which cannot be taken into account for the early elicitation of concrete threats. Furthermore, since our contributions do not address the software requirements engineering in the context of MechatronicUML [Hol19; Hol+16b], the security requirements specified at the level of Consens are not explicitly handled during this phase. Thus, future research activities should enable software requirements engineers to analyze upfront whether a requirements specification including security requirements is consistent and realizable during the downstream software design.

Adaptivity: The contributions made in this thesis do not enable engineers to declassify information [SS09]. Accordingly, our works do not address security policies that are adaptive to changing security situations, as recently envisaged by Bennaceur et al. [Ben+19]. This is a major limitation due to the focus of MechatronicUML on *self-adaptation* of systems, which is achieved by structural reconfiguration of the underlying component architectures [HBV19]. As already described in Section 4.8, future works must use adaptive security policies to sense changes in the security situation and trigger reconfigurations that are necessary to prevent policy violations. As recently proposed by Khakpour et al. [KSNW19], security policies must also be used to decide whether a specific reconfiguration is secure or must be suppressed because it puts the system at risk. A crucial research challenge is to account for the adaption of security policies consistently across all phases of the engineering process. For example, the adaptivity must also be considered by systems engineers during the policy specification (cf. Chapter 3), or by software engineers during the verification of policies (cf. Chapter 5).

Evolution: Our contributions are limited in the sense that the underlying models are static, describing a system under development in delivery condition. Thereby, we do not take into consideration that systems evolve over time after they have first been delivered, such that their security must be maintained [Fel+14; Jür+19]. With respect to security, considering the evolution of systems is an indispensable obligation because the capabilities of attackers are expected to evolve as well, thereby establishing new attack vectors. Thus, security is not a steady state, but must be re-established by taking countermeasures in response to evolution of the attack surface. Since the MechatronicUML component model already supports multiplicities as a means to enable structural reconfiguration (cf. Section 4.8), this variability could provide the basis for the secure evolution of architectures as a future research topic.

Cognification: Like many other research fields, MDE is currently subject to an increasing *cognification* [CCBG18], integrating techniques from areas like artificial intelligence. Several contributions of this thesis have potential for being cognified as well. For example, future works could use the well-formedness rules proposed in Section 4.6 to search for optimal refinements of security policies automatically. Similar to a proposal by Scandariato et al. [SHF18], such an approach could be used to optimize the security arrangements made by software architects, enabling *design space exploration* on the basis of search-based MDE [BSA17]. For example, optimizing refinements could ease the integration of third-party or legacy components with fixed or even unknown security policies, minimize the verification effort by reducing the number of policies that must be actively verified, or enforce the principle of least privilege by preventing policies from being more permissive than necessary. Another example is a cognification of the inference engine proposed in Section 6.4. In future works, more accurate transformation rules could potentially be inferred with the help of search techniques [cf. MSS18; GKHE18; BS16; KSBB12] or machine learning [cf. BCG19a].

Resilience: Whereas Consens and MechatronicUML previously addressed the safety of systems under development, our contributions extend the design method with an approach for security engineering. We thereby account for the fact that CPSs must be equally *resilient* to threats from both categories [TSHL16], especially because security incidents can have safety-critical consequences. A limitation of our contributions is that safety and security aspects are still handled separately, instead of considering a possible interplay between both properties in detail. By contrast, the cross-fertilization between safety and security engineering is a prominent research topic [BW18; PB13], which should be taken into account by future works.

Layered Security: Controlling the information flow security is only one of many possible measures that can be adopted when securing software systems. However, since the effectiveness of the individual security measures is limited to specific kinds of attacks, no measure provides a silver bullet for overall security on its own. Instead, the approach of taking multiple complementary security measures, which are arranged in layers, is known as *layered security*.

For example, due to its restrictiveness, information flow control can only rule out such flows that do not affect the system's functionality. In contrast, a certain degree of information flow might be indispensable to satisfy the functional requirements of a system. Thus, such flows can only be ruled out by information flow control if it is possible to declassify certain critical information (see above). Nevertheless, declassified information must still be secured as effectively as possible. Future works must therefore trade off information flow control against more permissive measures such as *taint tracking* [SAB10], as done by Balliu et al. [BSS17]. Furthermore, since there are cases in which the information exchange cannot be effectively restricted at all, complementary measures such as *authentication* and *encryption* must be adopted to secure the exchanged information against tampering or disclosure. In this respect, the discipline-spanning domain of CPSs is particularly challenging because information security might also be compromised by exploiting continuous signals (cf. Section 4.8) or even physical side channels (cf. Section 3.7). Engineers must therefore protect such forms of interaction by suitable measures as well. In addition, since the engineering of CPSs also involves the physical layer of a system under development, physical protection measures such as door locks must also be put in place by engineers.

A future research challenge is therefore to integrate other proactive security measures besides information flow control into the design models. Such an integration would enable engineers to check the interplay of various security measures by conducting comprehensive security analyses. For example, Tuma et al. [TSB19] recently integrated a notion of encryption into a lightweight information flow analysis. Compared to our approach of analyzing the information flow only, a comprehensive security analysis could enable engineers to reason about a system's level of protection more significantly. In this context, future works must also analyze possible repercussions of security measures on other quality properties like performance [WR10; SLCS12].

Multiple security layers are of particular importance whenever a single layer is successfully compromised by an attacker. In this situation, security measures must compensate for other, compromised measures to prevent or at least delay a successful attack. Since each measure represents an individual line of defense, this approach is also referred to as *defense in depth* [Sty04]. For example, information flow security assumes that the direct access to information is limited to specific actors (cf. Section 2.4.1). If attackers succeed to invalidate these assumptions, the level of security guaranteed by information flow control is suspended. Thus, future works must also put in place reactive security measures such as *intrusion detection* [SEH19] to alleviate the impact of attacks.

Platform-Specific Deployment: According to our model-driven approach, the platform-independent design models emphasized in this thesis must be used to derive executable implementation artifacts, which can be deployed to a specific execution platform. A general challenge for the deployment is to preserve properties that have been verified at the model level. To this end, none of the assumptions made during the platform-independent design must be falsified by the platform-specific implementation.

In particular, to preserve verified security properties on deployment, design-level security measures must be properly implemented. For example, Peldszus et al. recently addressed this challenge in the context of data flow analyses [Pel+19]. To implement the information flow control as a security measure adopted in our work, fundamental assumptions that were made about the accessibility of information by specific actors (cf. Section 2.4.1) must not be falsified at the implementation level. Accordingly, *access control* is a means to implement information flow security [Man03, p. 23] such that the provided security guarantees are preserved on deployment.

In the context of MDE, future works must enable engineers to synthesize proper access control implementations automatically and thereby preserve information flow security on deployment without manual intervention. Accordingly, the synthesis must ensure that the design is properly refined by the implementation. A challenging research task is therefore to deal with the *refinement paradox* [Ros95] as already described in Section 5.7. According to this phenomenon, secure information flow is not guaranteed to be preserved in case of a proper refinement [Jac89]. Hence, synthesizing secure implementations is a future research challenge that must be treated with special care.

A

CoCoME Security Policies

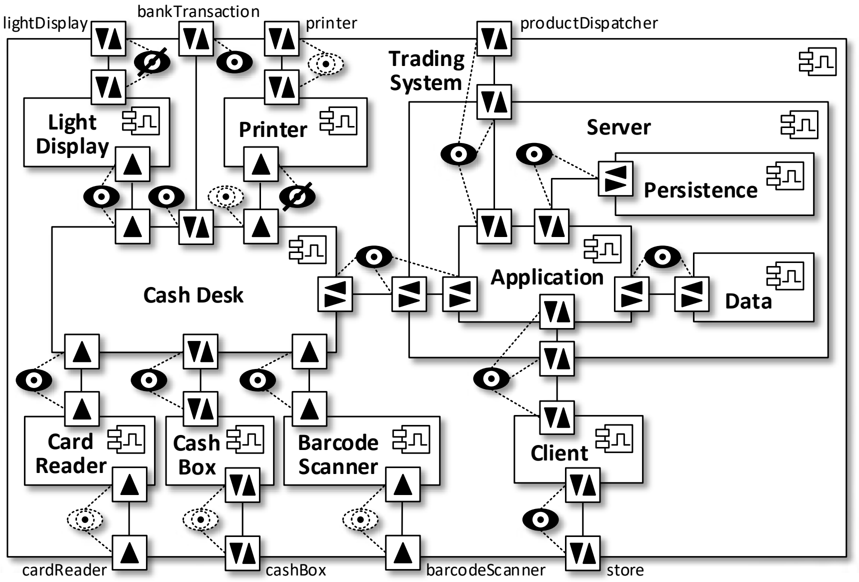


Figure A.1: Security policy for the lightDisplay port.

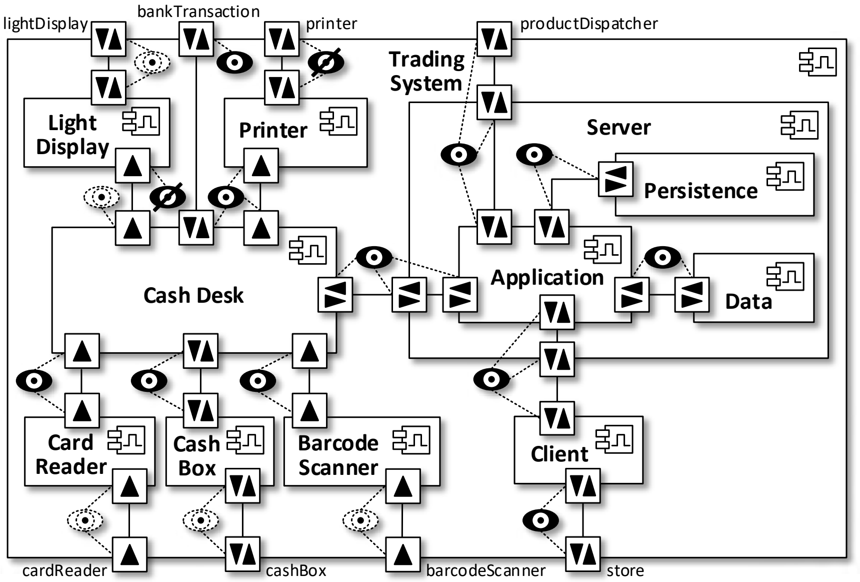


Figure A.2: Security policy for the printer port.

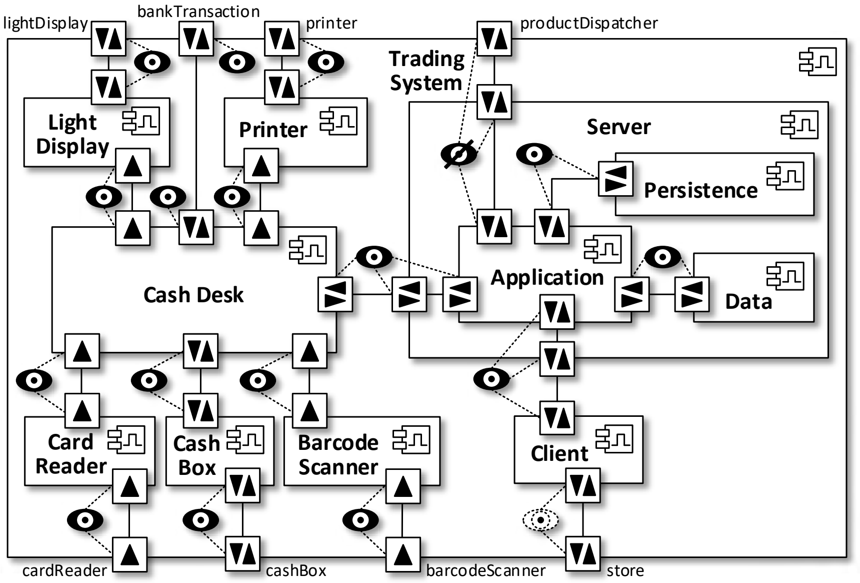


Figure A.3: Security policy for the productDispatcher port.

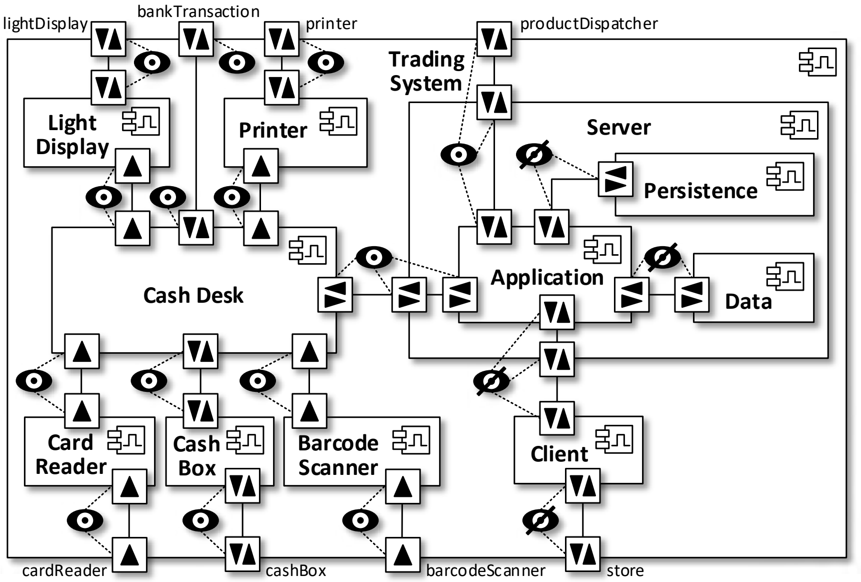


Figure A.4: Security policy for the store port.

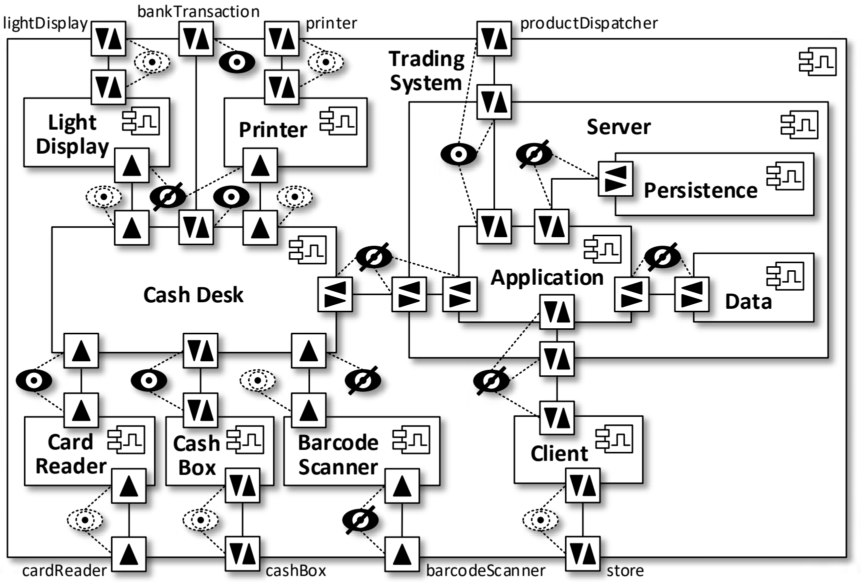


Figure A.5: Security policy for the barcodeScanner port.

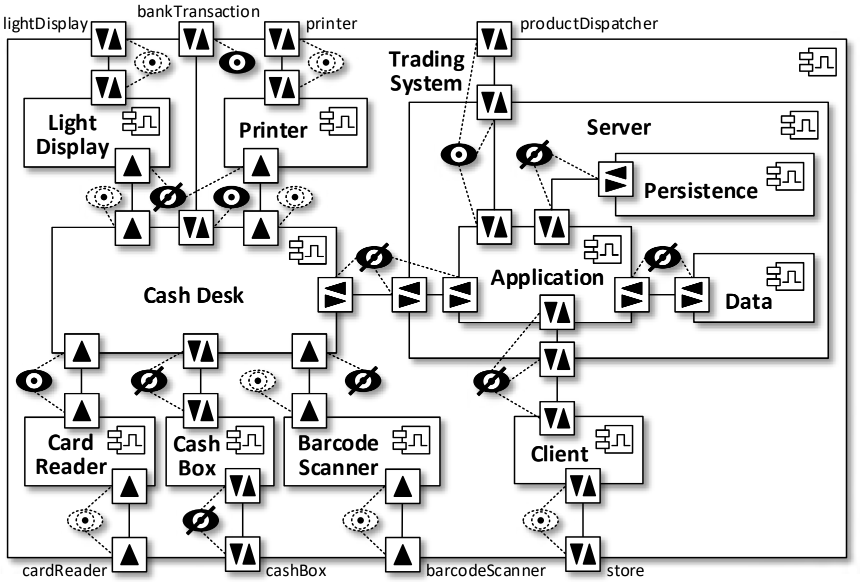


Figure A.6: Security policy for the cashBox port.

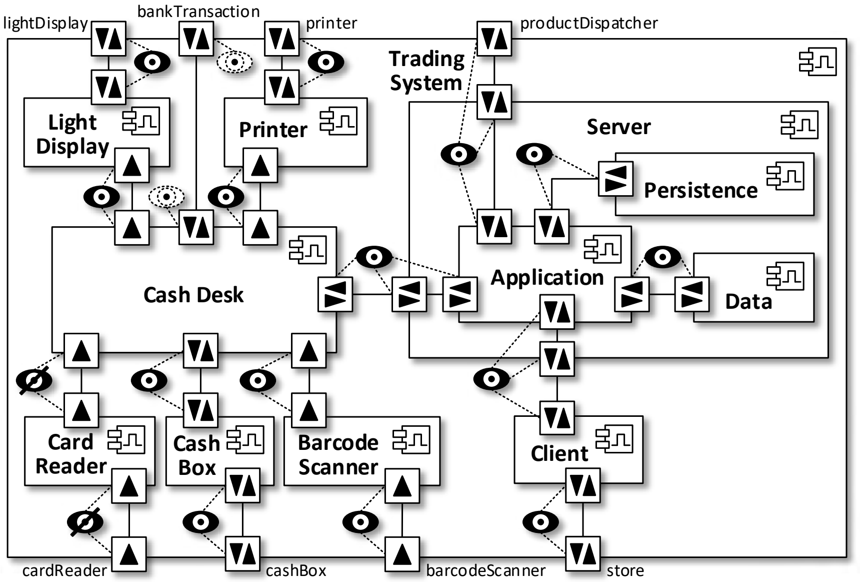


Figure A.7: Security policy for the cardReader port.

B

Implemented Execution Framework

Listing B.1: QVTo-based implementation of the execution framework from Section 6.5.

```
modeltype mappings uses 'http://www.upb.de/2020/Mappings';
modeltype ecore uses 'http://www.eclipse.org/emf/2002/Ecore';

library Execution;

mapping Element :: transform(translator : ClassMapping) : Element
when {self.conformsTo(translator)} {

    init {
        if (result.oclIsUndefined()) {
            translator.subclassMappings->forEach(sub) {
                if (result.oclIsUndefined()) {
                    result := self.map transform(sub);
                }
            };

            if (result.oclIsUndefined()) {
                result := translator.target.ePackage.
                    eFactoryInstance.create(translator.target);
            }
        }
    }

    self.map initialize(translator, result);
}

query Element :: conformsTo(cm : ClassMapping) : Boolean {

    var c = self.oclAsType(EObject).eClass();

    return (c = cm.source or c.eAllSuperTypes->includes(cm.source))
        and self.ocl(cm.condition);
}
```

```
blackbox query OclAny :: ocl(constraint : String) : Boolean;

mapping Element :: initialize(cm : ClassMapping, reslt : Element) {

    cm.superclassMappings->forEach(super) {
        self.map initialize(super, reslt);
    };

    cm.featureMappings->forEach(fm) {

        var sourceValues : Sequence(OclAny);

        if (fm.source.oclIsUndefined()) {
            sourceValues += self;
        }
        else {
            sourceValues += self![EObject].eGet(fm.source);
        };


        var targetValues : Sequence(OclAny);

        if (fm.target.many) {
            targetValues += reslt![EObject].eGet(fm.target);
        };

        if (fm.target.oclIsKindOf(EAttribute)) {
            targetValues += sourceValues;
        }
        else {
            targetValues += sourceValues[Element].map transform(fm.
                translator);
        };


        if (fm.target.many) {
            targetValues := targetValues->excluding(null);
            reslt![EObject].eSet(fm.target, targetValues);
        }
        else {
            var targetValue = targetValues->any(true);
            reslt![EObject].eSet(fm.target, targetValue);
        }
    }
}
```

Publications and Contributions

In the following, I point out my personal contribution to all scientific publications that were authored or co-authored by me and served as a basis for this thesis.

- Christopher Gerking. “Traceability of Information Flow Requirements in Cyber-Physical Systems Engineering”. In: *Doctoral Symposium at MoDELS 2016*. MoDELS-DS 2016 (Saint-Malo, France, Oct. 2, 2016). Proceedings of the Doctoral Symposium at the 19th ACM/IEEE International Conference of Model-Driven Engineering Languages and Systems 2016 (MoDELS 2016). Ed. by Shiva Nejati and Rick Salay. CEUR Workshop Proceedings 1735. ISSN: 1613-0073. 2016. URN: urn:nbn:de:0074-1735-1 [*Ger16].

  I am the sole author of this publication, which was conceptualized and written by myself under the supervision of Wilhelm Schäfer and Eric Bodden. This work motivates and outlines two of the main contributions made in this thesis and therefore serves as the basis for Chapters 3 and 5.

- Christopher Gerking, David Schubert, and Ingo Budde. “Reducing the Verbosity of Imperative Model Refinements by Using General-Purpose Language Facilities”. In: *Theory and Practice of Model Transformation*. 10th International Conference. ICMT 2017 (Marburg, Germany, July 17–18, 2017). Proceedings. Ed. by Esther Guerra and Mark G. J. van den Brand. Lecture Notes in Computer Science 10374. Springer, 2017, pp. 19–34. ISBN: 978-3-319-61472-4. DOI: 10.1007/978-3-319-61473-1_2 [*GSB17].

  This publication includes a premature version of our execution framework proposed in Chapter 6, restricted to endogenous model transformations. I am the main author of this paper, whereas the technical realization has been prototyped by Ingo Budde. I researched the related work, extrapolated the language facilities from the prototype, and compared existing languages against these facilities. Whereas the proof of concept has been documented by David Schubert in cooperation with me, I wrote the residual parts of the paper myself.

- Johannes Geismann, Christopher Gerking, and Eric Bodden. “Towards ensuring security by design in cyber-physical systems engineering processes”. In: *Proceedings of the 2018 International Conference on Software and System Process*. ICSSP 2018 (Gothenburg, Sweden, May 26–27, 2018). Ed. by Marco Kuhrmann, Rory V. O’Connor, Dan Houston, and Regina Hebig. ACM, 2018, pp. 123–127. ISBN: 978-1-4503-6459-1. DOI: 10.1145/3202710.3203159 [*GGB18].

  Besides other security countermeasures, this paper briefly integrates our contributions from Chapters 3 to 5 into an engineering process for CPSs. The paper has been conceptualized by Johannes Geismann and was co-authored by me. In particular, I wrote all parts relating to the platform-independent engineering, as well as the introduction and conclusion. Furthermore, I researched and documented the related work in cooperation with Johannes Geismann. The writing of the paper has been supervised by Eric Bodden.

- Christopher Gerking. "Specification of Information Flow Security Policies in Model-Based Systems Engineering". In: *Software Technologies. Applications and Foundations*. STAF 2018 Collocated Workshops (Toulouse, France, June 25–29, 2018). Revised Selected Papers. Ed. by Manuel Mazzara, Iulian Ober, and Gwen Salaün. Lecture Notes in Computer Science 11176. Springer, 2018, pp. 617–632. ISBN: 978-3-030-04770-2. DOI: 10.1007/978-3-030-04771-9_47 [*Ger18].

  I am the sole author of this publication, which presents our specification technique for security policies proposed in Chapter 3 of this thesis. The paper has been conceptualized and written by myself under the supervision of Eric Bodden.

- Christopher Gerking, David Schubert, and Eric Bodden. "Model Checking the Information Flow Security of Real-Time Systems". In: *Engineering Secure Software and Systems*. 10th International Symposium. ESSoS 2018 (Paris, France, June 26–27, 2018). Proceedings. Ed. by Mathias Payer, Awais Rashid, and Jose M. Such. Lecture Notes in Computer Science 10953. Springer, 2018, pp. 27–43. ISBN: 978-3-319-94495-1. DOI: 10.1007/978-3-319-94496-8_3 [*GSB18].

  This paper presents the verification technique proposed in Chapter 5 of this thesis. I am the main author of this publication. The contents have been conceptualized and written by me in cooperation with David Schubert and Eric Bodden. In particular, I researched the related work and gave the proof of concept. Advised by David Schubert, I also constructed the test automata as the main technical contribution of this paper. David Schubert documented parts of the fundamentals and the automata construction, whereas the writing of the introduction has been supported by Eric Bodden. All residual parts of the paper were written by me.

- Christopher Gerking and David Schubert. "Towards Preserving Information Flow Security on Architectural Composition of Cyber-Physical Systems". In: *Software Architecture*. 12th European Conference on Software Architecture. ECSA 2018 (Madrid, Spain, Sept. 24–28, 2018). Proceedings. Ed. by Carlos E. Cuesta, David Garlan, and Jennifer Pérez. Lecture Notes in Computer Science 11048. Springer, 2018, pp. 147–155. ISBN: 978-3-030-00760-7. DOI: 10.1007/978-3-030-00761-4_10 [*GS18].

  This paper sketches our well-formedness rules for component-based security policies, which have been proposed as best practices in Chapter 4 of this thesis. As the main author, I conceptualized the paper and also researched the related work. Whereas the description of the underlying component model was given by David Schubert as a co-author, the residual parts of the paper were written by me.

- Christopher Gerking and David Schubert. “Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures”. In: *IEEE International Conference on Software Architecture*. ICSA 2019 (Hamburg, Germany, Mar. 25–29, 2019). Proceedings. IEEE, 2019, pp. 61–70. ISBN: 978-1-7281-0528-4. DOI: 10.1109/ICSA.2019.00015 [*GS19].

  This paper elaborates and evaluates our well-formedness rules for component-based security policies proposed in Chapter 4. I am the lead author of this paper, which has been co-authored by David Schubert. I conceptualized our contribution, researched related work, and conducted the case study. David Schubert contributed to this publication in an advisory capacity and by documenting background information on MechatronicUML. All residual parts were written by me.

- Christopher Gerking and Ingo Budde. “Heuristic Inference of Model Transformation Definitions from Type Mappings”. In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion*. MODELS-C 2019 (Munich, Germany, Sept. 15–20, 2019). Proceedings. Ed. by Loli Burgueño et al. IEEE, 2019, pp. 182–188. ISBN: 978-1-7281-5125-0. DOI: 10.1109/MODELS-C.2019.00031 [*GB19].

  This publication presents the inference engine for transformation models proposed in Chapter 6 of this thesis. The published results are based on the bachelor thesis of Ingo Budde [Bud18], who also contributed to this paper as a co-author. As a supervisor of the thesis, I revised and streamlined the initial outcomes for the publication in this paper. The case studies were conducted by Ingo Budde in cooperation with me. As the main author, I wrote all parts of the paper myself.

Bibliography

The bibliography is subdivided into published works authored or co-authored by me, theses supervised by me, scientific literature by other authors, and standard specifications.

Published Works

[*Bec+14] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Wilhelm Schäfer, Matthias Meyer, and Uwe Pohlmann. "The MechatronicUML method. Model-driven software engineering of self-adaptive mechatronic systems". In: *36th International Conference on Software Engineering*. ICSE Companion 2014 (Hyderabad, India, May 31–June 7, 2014). Proceedings. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 614–615. isbn: 978-1-4503-2768-8. doi: 10.1145/2591062.2591142 (cit. on pp. 18, 145).

[*Dzi+14] Stefan Dziwok, Christopher Gerking, Steffen Becker, Sebastian Thiele, Christian Heinzemann, and Uwe Pohlmann. "A tool suite for the model-driven software engineering of cyber-physical systems". In: *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*. FSE 2014 (Hong Kong, China, Nov. 16–22, 2014). Proceedings. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 715–718. isbn: 978-1-4503-3056-5. doi: 10.1145/2635868.2661665 (cit. on p. 18).

[*Dzi+16] Stefan Dziwok, Uwe Pohlmann, Goran Piskachev, David Schubert, Sebastian Thiele, and Christopher Gerking. *The MechatronicUML Design Method. Process and Language for Platform-Independent Modeling*. Tech. rep. tr-ri-16-352. Version 1.0. Software Engineering Department, Fraunhofer IEM and Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn. Dec. 2016. url: https://www.hni.uni-paderborn.de/pub/9478 (cit. on pp. 18, 22, 105).

[*GB19] Christopher Gerking and Ingo Budde. "Heuristic Inference of Model Transformation Definitions from Type Mappings". In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion*. MODELS-C 2019 (Munich, Germany, Sept. 15–20, 2019). Proceedings. Ed. by Loli Burgueño et al. IEEE, 2019, pp. 182–188. isbn: 978-1-7281-5125-0. doi: 10.1109/MODELS-C.2019.00031 (cit. on pp. 8, 115, 126, 127, 137–139, 143, 159).

[*GBS20] Christopher Gerking, Eric Bodden, and Wilhelm Schäfer. “Industrial Security by Design. Nachverfolgbare Informationssicherheit für Cyber-Physische Produktionssysteme”. German. In: *Handbuch Gestaltung digitaler und vernetzter Arbeitswelten*. Ed. by Günter W. Maier, Gregor Engels, and Eckhard Steffen. Springer, 2020, pp. 139–162. isbn: 978-3-662-52898-3. doi: 10.1007/978-3-662-52979-9_8. Repr. from *Handbuch Gestaltung digitaler und vernetzter Arbeitswelten*. Springer Reference Psychologie. 2017, pp. 1–24. isbn: 978-3-662-52903-4. doi: 10.1007/978-3-662-52903-4_8-1.

[*Ger13] Christopher Gerking. “Transparent Uppaal-based Verification of MechatronicUML Models”. MA thesis. University of Paderborn, May 2013. url: https://www.hni.uni-paderborn.de/pub/7908 (cit. on pp. 23, 25).

[*Ger16] Christopher Gerking. “Traceability of Information Flow Requirements in Cyber-Physical Systems Engineering”. In: *Doctoral Symposium at MoDELS 2016*. MoDELS-DS 2016 (Saint-Malo, France, Oct. 2, 2016). Proceedings of the Doctoral Symposium at the 19th ACM/IEEE International Conference of Model-Driven Engineering Languages and Systems 2016 (MoDELS 2016). Ed. by Shiva Nejati and Rick Salay. CEUR Workshop Proceedings 1735. issn: 1613-0073. 2016. urn: urn:nbn:de:0074-1735-1 (cit. on pp. 4, 35, 87, 157).

[*Ger18] Christopher Gerking. “Specification of Information Flow Security Policies in Model-Based Systems Engineering”. In: *Software Technologies. Applications and Foundations*. STAF 2018 Collocated Workshops (Toulouse, France, June 25–29, 2018). Revised Selected Papers. Ed. by Manuel Mazzara, Iulian Ober, and Gwen Salaün. Lecture Notes in Computer Science 11176. Springer, 2018, pp. 617–632. isbn: 978-3-030-04770-2. doi: 10.1007/978-3-030-04771-9_47 (cit. on pp. 7, 30, 35, 40, 42, 43, 57, 158).

[*Ger20a] Christopher Gerking. *A Verification Technique for Real-Time Information Flow Security*. Data set. Zenodo, 2020. doi: 10.5281/zenodo.3715313 (cit. on p. 105).

[*Ger20b] Christopher Gerking. *Imperative Refinement of Declarative Model Transformations*. In collab. with Ingo Budde. Data set. Zenodo, 2020. doi: 10.5281/zenodo.3678348 (cit. on p. 139).

[*GGB18] Johannes Geismann, Christopher Gerking, and Eric Bodden. “Towards ensuring security by design in cyber-physical systems engineering processes”. In: *Proceedings of the 2018 International Conference on Software and System Process*. ICSSP 2018 (Gothenburg, Sweden, May 26–27, 2018). Ed. by Marco Kuhrmann, Rory V. O’Connor, Dan Houston, and Regina Hebig. ACM, 2018, pp. 123–127. isbn: 978-1-4503-6459-1. doi: 10.1145/3202710.3203159 (cit. on pp. 35, 157).

[*GH14] Christopher Gerking and Christian Heinzemann. “Solving the Movie Database Case with QVTo”. In: *Transformation Tool Contest*. TTC 2014 (York, UK, July 25, 2014). Proceedings of the 7th Transformation Tool Contest. Ed. by Louis M. Rose, Christian Krause, and Tassilo Horn. CEUR Workshop Proceedings 1305. ISSN: 1613-0073. 2014, pp. 98–102. URN: urn:nbn:de:0074-1305-7.

[*GLS15] Christopher Gerking, Jan Ladleif, and Wilhelm Schäfer. “Model-driven test case design for model-to-model semantics preservation”. In: *6th International Workshop on Automating Test Case Design, Selection and Evaluation*. A-TEST 2015 (Bergamo, Italy, Aug. 30–31, 2015). Proceedings. Ed. by Tanja E. J. Vos, Sigrid Eldh, and Wishnu Prasetya. ACM, 2015, pp. 1–7. ISBN: 978-1-4503-3813-4. DOI: 10.1145/2804322.2804323.

[*GS18] Christopher Gerking and David Schubert. “Towards Preserving Information Flow Security on Architectural Composition of Cyber-Physical Systems”. In: *Software Architecture*. 12th European Conference on Software Architecture. ECSA 2018 (Madrid, Spain, Sept. 24–28, 2018). Proceedings. Ed. by Carlos E. Cuesta, David Garlan, and Jennifer Pérez. Lecture Notes in Computer Science 11048. Springer, 2018, pp. 147–155. ISBN: 978-3-030-00760-7. DOI: 10.1007/978-3-030-00761-4_10 (cit. on pp. 7, 57, 68, 158).

[*GS19] Christopher Gerking and David Schubert. “Component-Based Refinement and Verification of Information-Flow Security Policies for Cyber-Physical Microservice Architectures”. In: *IEEE International Conference on Software Architecture*. ICSA 2019 (Hamburg, Germany, Mar. 25–29, 2019). Proceedings. IEEE, 2019, pp. 61–70. ISBN: 978-1-7281-0528-4. DOI: 10.1109/ICSA.2019.00015 (cit. on pp. 7, 57, 67, 68, 87, 91, 99, 100, 102, 159).

[*GSB17] Christopher Gerking, David Schubert, and Ingo Budde. “Reducing the Verbosity of Imperative Model Refinements by Using General-Purpose Language Facilities”. In: *Theory and Practice of Model Transformation*. 10th International Conference. ICMT 2017 (Marburg, Germany, July 17–18, 2017). Proceedings. Ed. by Esther Guerra and Mark G. J. van den Brand. Lecture Notes in Computer Science 10374. Springer, 2017, pp. 19–34. ISBN: 978-3-319-61472-4. DOI: 10.1007/978-3-319-61473-1_2 (cit. on pp. 8, 115, 127, 133, 157).

[*GSB18] Christopher Gerking, David Schubert, and Eric Bodden. “Model Checking the Information Flow Security of Real-Time Systems”. In: *Engineering Secure Software and Systems*. 10th International Symposium. ESSoS 2018 (Paris, France, June 26–27, 2018). Proceedings. Ed. by Mathias Payer, Awais Rashid, and Jose M. Such. Lecture Notes in Computer Science 10953. Springer, 2018, pp. 27–43. ISBN: 978-3-319-94495-1. DOI: 10.1007/978-3-319-94496-8_3 (cit. on pp. 8, 87, 89, 91, 93, 158).

[*GSDH15] Christopher Gerking, Wilhelm Schäfer, Stefan Dziwok, and Christian Heinzemann. “Domain-Specific Model Checking for Cyber-Physical Systems”. In: *Model-Driven Engineering, Verification and Validation*. MoDeVVa 2015 (Ottawa, Canada, Sept. 29, 2015). Proceedings of the 12th Workshop on Model-Driven Engineering, Verification and Validation. Ed. by Michalis Famelis, Daniel Ratiu, Martina Seidl, and Gehan M. K. Selim. CEUR Workshop Proceedings 1514. ISSN: 1613-0073. 2015, pp. 18–27. URN: urn:nbn:de:0074-1514-4 (cit. on pp. 19, 26).

[*PHMG14] Uwe Pohlmann, Jörg Holtmann, Matthias Meyer, and Christopher Gerking. “Generating Modelica Models from Software Specifications for the Simulation of Cyber-Physical Systems”. In: *40th EUROMICRO Conference on Software Engineering and Advanced Applications*. EUROMICRO-SEAA 2014 (Verona, Italy, Aug. 27–29, 2014). Proceedings. Ed. by Rick Rabiser and Richard Torkar. IEEE Computer Society, 2014, pp. 191–198. ISBN: 978-1-4799-5795-8. DOI: 10.1109/SEAA.2014.18.

[*Sch+17] Stefano Schivo, Bugra M. Yildiz, Enno Ruijters, Christopher Gerking, Rajesh Kumar, Stefan Dziwok, Arend Rensink, and Mariëlle Stoelinga. “How to Efficiently Build a Front-End Tool for Uppaal. A Model-Driven Approach”. In: *Dependable Software Engineering. Theories, Tools, and Applications*. Third International Symposium. SETTA 2017 (Changsha, China, Oct. 23–25, 2017). Proceedings. Ed. by Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang. Lecture Notes in Computer Science 10606. Springer, 2017, pp. 319–336. ISBN: 978-3-319-69482-5. DOI: 10.1007/978-3-319-69483-2_19 (cit. on p. 103).

[*SHG16] David Schubert, Christian Heinzemann, and Christopher Gerking. “Towards Safe Execution of Reconfigurations in Cyber-Physical Systems”. In: *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE 2016 (Venice, Italy, Apr. 5–8, 2016). Proceedings. IEEE Computer Society, 2016, pp. 33–38. ISBN: 978-1-5090-2569-5. DOI: 10.1109/CBSE.2016.10.

Supervised Theses

[Bru15] Christopher Brune. “Domain-Specific Property Specification for Timed Model Checking”. MA thesis. University of Paderborn, Feb. 2015.

[Bud18] Ingo Budde. “Verfeinerung deklarativer Mappingmodelle durch imperative Modelltransformationen”. German. BA thesis. Paderborn University, Oct. 2018 (cit. on pp. 115, 159).

[Cor18] Lukas Corona. “Modellbasierte Spezifikation deklassifizierbarer Sicherheitsrichtlinien für cyber-physische Systeme”. German. BA thesis. Paderborn University, July 2018 (cit. on p. 107).

[Lad14] Jan Ladleif. “Testbasierte Qualitätssicherung für Modellanalysen im Kontext von MechatronicUML”. German. BA thesis. University of Paderborn, Nov. 2014.

[Man17] Jonas Manuel. “A Modular Simulation Architecture for the Operational Semantics of MechatronicUML”. MA thesis. Paderborn University, Dec. 2017.

[Som18] Poul Henry Sommer. “Imperative Verfeinerung von Modell-Metamodell-Koevolutionstransformationen”. German. BA thesis. Paderborn University, Nov. 2018 (cit. on p. 140).

Scientific Literature

[AB11] Marcel van Amstel and Mark G. J. van den Brand. “Model Transformation Analysis. Staying Ahead of the Maintenance Nightmare”. In: *Theory and Practice of Model Transformations*. 4th International Conference. ICMT 2011 (Zürich, Switzerland, June 27–28, 2011). Proceedings. Ed. by Jordi Cabot and Eelco Visser. Lecture Notes in Computer Science 6707. Springer, 2011, pp. 108–122. isbn: 978-3-642-21731-9. doi: 10.1007/978-3-642-21732-6_8 (cit. on p. 118).

[ABKP11] Marcel van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. “Performance in Model Transformations. Experiments with ATL and QVT”. In: *Theory and Practice of Model Transformations*. 4th International Conference. ICMT 2011 (Zürich, Switzerland, June 27–28, 2011). Proceedings. Ed. by Jordi Cabot and Eelco Visser. Lecture Notes in Computer Science 6707. Springer, 2011, pp. 198–212. isbn: 978-3-642-21731-9. doi: 10.1007/978-3-642-21732-6_14 (cit. on p. 118).

[ABL98] Luca Aceto, Augusto Burgueño, and Kim Guldstrand Larsen. “Model Checking via Reachability Testing for Timed Automata”. In: *Tools and Algorithms for Construction and Analysis of Systems*. 4th International Conference. TACAS ’98 (Lisbon, Portugal, Mar. 28–Apr. 4, 1998). Proceedings. Ed. by Bernhard Steffen. Lecture Notes in Computer Science 1384. Springer, 1998, pp. 263–280. isbn: 3-540-64356-7. doi: 10.1007/BFb0054177 (cit. on pp. 86, 90, 105).

[AD94] Rajeev Alur and David L. Dill. “A Theory of Timed Automata”. In: *Theoretical Computer Science* 126.2 (Apr. 1994), pp. 183–235. issn: 0304-3975. doi: 10.1016/0304-3975(94)90010-8 (cit. on pp. 5, 23, 71, 146).

[AFW06] Khaled Alghathbar, Csilla Farkas, and Duminda Wijesekera. “Securing UML Information Flow Using FlowUML”. In: *Journal of Research and Practice in Information Technology* 38.1 (Feb. 2006), pp. 111–120. issn: 1443-458X. url: https://acs.org.au/content/dam/acs/50-years/journals/jrpit/JRPIT38.1.111.pdf (cit. on p. 50).

[Aga00] Johan Agat. “Transforming Out Timing Leaks”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2000 (Boston, Massachusetts, USA, Jan. 19–21, 2000). Ed. by Mark N. Wegman and Thomas W. Reps. ACM, 2000, pp. 40–53. isbn: 1-58113-125-9. doi: 10.1145/325694.325702 (cit. on p. 110).

[AGD18] Deniz Akdur, Vahid Garousi, and Onur Demirörs. “A survey on modeling and model-driven engineering practices in the embedded software industry”. In: *Journal of Systems Architecture. Embedded Software Design* 91 (Nov. 2018), pp. 62–82. issn: 1383-7621. doi: 10.1016/j.sysarc.2018.09.007 (cit. on p. 1).

[AIS18] Rasim M. Alguliyev, Yadigar Imamverdiyev, and Lyudmila Sukhostat. “Cyber-physical systems and their security issues”. In: *Computers in Industry* 100 (Sept. 2018), pp. 212–223. issn: 0166-3615. doi: 10.1016/j.compind.2018.04.017 (cit. on p. 1).

[AK05] Colin Atkinson and Thomas Kühne. “A Generalized Notion of Platforms for Model-Driven Development”. In: *Model-Driven Software Development*. Ed. by Sami Beydeda, Matthias Book, and Volker Gruhn. Springer, 2005, pp. 119–136. isbn: 978-3-540-25613-7. doi: 10.1007/3-540-28554-7_6 (cit. on p. 9).

[AL93] Martín Abadi and Leslie Lamport. “Composing Specifications”. In: *ACM Transactions on Programming Languages and Systems* 15.1 (Jan. 1993), pp. 73–132. issn: 0164-0925. doi: 10.1145/151646.151649 (cit. on p. 32).

[ALS16] Anthony Anjorin, Erhan Leblebici, and Andy Schürr. “20 Years of Triple Graph Grammars. A Roadmap for Future Research”. In: *Electronic Communications of the EASST* 73 (2016): *Graph Computation Models. Selected Revised Papers from GCM 2015*, pp. 1–20. issn: 1863-2122. doi: 10.14279/tuj.eceasst.73.1031 (cit. on p. 117).

[AM17] Yosef Ashibani and Qusay H. Mahmoud. “Cyber physical systems security. Analysis, challenges and solutions”. In: *Computers & Security* 68 (July 2017), pp. 81–97. issn: 0167-4048. doi: 10.1016/j.cose.2017.04.005 (cit. on p. 1).

[Ana+14a] Harald Anacker, Christian Brenner, Rafał Dorociak, Roman Dumitrescu, Jürgen Gausemeier, Peter Iwanek, Wilhelm Schäfer, and Mareen Vaßholz. “Methods for the Domain-Spanning Conceptual Design”. In: *Design Methodology for Intelligent Technical Systems. Develop Intelligent Technical Systems of the Future*. Ed. by Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer. Lecture Notes in Mechanical Engineering. Springer, 2014. Chap. 4, pp. 117–182. isbn: 978-3-642-45434-9. doi: 10.1007/978-3-642-45435-6_4 (cit. on pp. 14, 16, 145).

[Ana+14b] Harald Anacker et al. “Methods for the Design and Development”. In: *Design Methodology for Intelligent Technical Systems. Develop Intelligent Technical Systems of the Future*. Ed. by Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer. Lecture Notes in Mechanical Engineering. Springer, 2014. Chap. 5, pp. 183–350. isbn: 978-3-642-45434-9. doi: 10.1007/978-3-642-45435-6_5 (cit. on pp. 18, 62).

[APN17] Faeq Alrimawi, Liliana Pasquale, and Bashar Nuseibeh. “Software Engineering Challenges for Investigating Cyber-Physical Incidents”. In: *2017 IEEE/ACM 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems*. SEsCPS 2017 (Buenos Aires, Argentina, May 21, 2017). Proceedings. IEEE, 2017, pp. 34–40. isbn: 978-1-5386-4043-2. doi: 10.1109/SEsCPS.2017.9 (cit. on p. 1).

[AR16] Ludovic Apvrille and Yves Roudier. “Designing Safe and Secure Embedded and Cyber-Physical Systems with SysML-Sec”. In: *Model-Driven Engineering and Software Development*. Third International Conference. MODELSWARD 2015 (Angers, France, Feb. 9–11, 2015). Revised Selected Papers. Ed. by Philippe Desfray, Joaquim Filipe, Slimane Hammoudi, and Luís Ferreira Pires. Communications in Computer and Information Science 580. Springer, 2016, pp. 293–308. isbn: 978-3-319-27868-1. doi: 10.1007/978-3-319-27869-8_17 (cit. on pp. 34, 52).

[AS19] Étienne André and Jun Sun. “Parametric Timed Model Checking for Guaranteeing Timed Opacity”. In: *Automated Technology for Verification and Analysis*. 17th International Symposium. ATVA 2019 (Taipei, Taiwan, Oct. 28–31, 2019). Proceedings. Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Lecture Notes in Computer Science 11781. Springer, 2019, pp. 115–130. isbn: 978-3-030-31783-6. doi: 10.1007/978-3-030-31784-3_7 (cit. on pp. 86, 108, 110).

[AS85] Bowen Alpern and Fred B. Schneider. “Defining Liveness”. In: *Information Processing Letters* 21.4 (Oct. 1985), pp. 181–185. issn: 0020-0190. doi: 10.1016/0020-0190(85)90056-0 (cit. on pp. 6, 19, 32, 85).

[ASL02] Ross J. Anderson, Frank Stajano, and Jong-Hyeon Lee. “Security policies”. In: *Advances in Computers*. Vol. 55. Ed. by Marvin V. Zelkowitz. Elsevier, 2002, pp. 185–235. isbn: 978-0-12-012155-7. doi: 10.1016/S0065-2458(01)80030-9 (cit. on pp. 2, 29).

[ASSS13] Luciane Telinski Wiedermann Agner, Inali Wisniewski Soares, Paulo Cézar Stadzisz, and Jean Marcelo Simão. “A Brazilian survey on UML and model-driven practices for embedded software development”. In: *Journal of Systems and Software* 86.4 (Apr. 2013): *Software Engineering in Brazil. Retrospective and Prospective Views*. Ed. by Alessandro Garcia, pp. 997–1005. issn: 0164-1212. doi: 10.1016/j.jss.2012.11.023 (cit. on p. 1).

[Bat+16] Edouard Batot, Houari A. Sahraoui, Eugene Syriani, Paul Molins, and Wael Sboui. “Systematic Mapping Study of Model Transformations for Concrete Problems”. In: *4th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD 2016 (Rome, Italy, Feb. 19–21, 2016). Proceedings. Ed. by Slimane Hammoudi, Luís Ferreira Pires, Bran Selic, and Philippe Desfray. SciTePress, 2016, pp. 176–183. ISBN: 978-989-758-168-7. DOI: 10.5220/0005657301760183 (cit. on pp. 115, 134).

[Bau+17] Thomas Bauereiß et al. *RIFL 1.1. A Common Specification Language for Information-Flow Requirements*. Tech. rep. TUD-CS-2017-0225. Technische Universität Darmstadt. Aug. 2017. DOI: 10.5445/IR/1000092713 (cit. on p. 50).

[BBCW19] Hugo Brunelière, Erik Burger, Jordi Cabot, and Manuel Wimmer. “A feature-based survey of model view approaches”. In: *Software and Systems Modeling* 18.3 (June 2019): *Theme Section on model-based design of Cyber-Physical Systems; MODELS 2016 Special Section; EMMSAD 2017 Special Section*, pp. 1931–1952. ISSN: 1619-1366. DOI: 10.1007/s10270-017-0622-9 (cit. on p. 14).

[BBMV18] Loli Burgueño, Manuel F. Bertoa, Nathalie Moreno, and Antonio Vallecillo. “Expressing Confidence in Models and in Model Transformation Elements”. In: *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS’18 (Copenhagen, Denmark, Oct. 14–19, 2018). Proceedings. Ed. by Andrzej Wąsowski, Richard F. Paige, and Øystein Haugen. ACM, 2018, pp. 57–66. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239394 (cit. on p. 140).

[BCE11] David A. Basin, Manuel Clavel, and Marina Egea. “A decade of model-driven security”. In: *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*. SACMAT’11 (Innsbruck, Austria, June 15–17, 2011). Ed. by Ruth Breu, Jason Crampton, and Jorge Lobo. ACM, 2011, pp. 1–10. ISBN: 978-1-4503-0688-1. DOI: 10.1145/1998441.1998443 (cit. on p. 4).

[BCG19a] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. “An LSTM-Based Neural Network Architecture for Model Transformations”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*. MODELS 2019 (Munich, Germany, Sept. 15–20, 2019). Proceedings. Ed. by Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño. IEEE, 2019, pp. 294–299. ISBN: 978-1-7281-2536-7. DOI: 10.1109/MODELS.2019.00013 (cit. on pp. 139, 142, 148).

[BCG19b] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. “The Future of Model Transformation Languages. An Open Community Discussion”. In: *Journal of Object Technology* 18.3, Art. No. 7 (July 2019): *The 12th International Conference on Model Transformations*. Ed. by Anthony Anjorin and Regina

Hebig, pp. 1–11. issn: 1660-1769. doi: 10.5381/jot.2019.18.3.a7 (cit. on pp. 113, 134).

[BCLR15] Gilles Benattar, Franck Cassez, Didier Lime, and Olivier H. Roux. "Control and synthesis of non-interferent timed systems". In: *International Journal of Control* 88.2 (2015), pp. 217–236. issn: 0020-7179. doi: 10.1080/00207179.2014.944356 (cit. on pp. 86, 108, 110).

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. 2nd ed. Synthesis Lectures on Software Engineering. Morgan & Claypool, 2017. isbn: 978-1-62705-708-0. doi: 10.2200/S00751ED2V01Y201701SWE004 (cit. on pp. 1, 9).

[BDB16] Karima Berramla, El Abbassia Deba, and Djilali Benhamamouch. "Model Transformation Generation. A Survey of the State-of-the-Art". In: *International Conference on Information Technology for Organizations Development*. IT4OD (Fez, Morocco, Mar. 30–Apr. 1, 2016). IEEE, 2016. isbn: 978-1-4673-7689-1. doi: 10.1109/IT4OD.2016.7479253 (cit. on p. 142).

[BDE13] David Broman, Patricia Derler, and John C. Eidson. "Temporal Issues in Cyber-Physical Systems". In: *Journal of the Indian Institute of Science. A Multidisciplinary Reviews Journal* 93.3 (July–Sept. 2013): *Cyber Physical Systems*. Ed. by Bharadwaj Amrutur, pp. 389–402. issn: 0970-4140. url: http://journal.library.iisc.ernet.in/index.php/iisc/article/view/1686 (cit. on p. 5).

[BDL04] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. "A Tutorial on Uppaal". In: *Formal Methods for the Design of Real-Time Systems*. International School on Formal Methods for the Design of Computer, Communication and Software Systems. SFM-RT 2004 (Bertinoro, Italy, Sept. 13–18, 2004). Revised Lectures. Ed. by Marco Bernardo and Flavio Corradini. Lecture Notes in Computer Science 3185: Tutorial. Springer, 2004, pp. 200–236. isbn: 3-540-23068-8. doi: 10.1007/978-3-540-30080-9_7 (cit. on p. 26).

[BDL06] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. "Model driven security. From UML models to access control infrastructures". In: *ACM Transactions on Software Engineering and Methodology* 15.1 (Jan. 2006), pp. 39–91. issn: 1049-331X. doi: 10.1145/1125808.1125810 (cit. on p. 4).

[BDR11] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21.6 (Dec. 2011): *Programming Language Interference and Dependence*, pp. 1207–1252. issn: 0960-1295. doi: 10.1017/S0960129511000193 (cit. on pp. 86, 89, 90, 109, 146).

[Bel+14] Nicolas Belloir, Vanea Chiprianov, Manzoor Ahmad, Manuel Munier, Laurent Gallon, and Jean-Michel Bruel. "Using Relax Operators into an MDE Security Requirement Elicitation Process for Systems of Systems". In: *Proceedings of the ECSA 2014 Workshops & Tool Demos Track*. European Conference on Software Architecture. ECSAW '14 (Vienna, Austria, Aug. 25–29, 2014). Ed. by Danny Weyns. ACM, 2014, Art. No. 32, pp. 1–4. isbn: 978-1-4503-2778-7. doi: 10.1145/2642803.2642835 (cit. on p. 52).

[Ben+19] Amel Bennaceur et al. "Modelling and analysing resilient cyber-physical systems". In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2019 (Montreal, Canada, May 25–26, 2019). Proceedings. Ed. by Marin Litoiu, Siobhán Clarke, and Kenji Tei. ACM, 2019, pp. 70–76. isbn: 978-1-7281-3368-3. doi: 10.1109/SEAMS.2019.00018 (cit. on pp. 79, 147).

[Ben+96] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. "Uppaal. A Tool Suite for Automatic Verification of Real-Time Systems". In: *Hybrid Systems*. Vol. III: *Verification and Control. Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems* (New Brunswick, New Jersey, USA, Oct. 22–25, 1995). Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Lecture Notes in Computer Science 1066. Springer, 1996, pp. 232–243. isbn: 3-540-61155-X. doi: 10.1007/BFb0020949 (cit. on pp. 6, 24, 75).

[Béz+06] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. "Model Transformations? Transformation Models!" In: *Model Driven Engineering Languages and Systems*. 9th International Conference. MoDELS 2006 (Genova, Italy, Oct. 1–6, 2006). Proceedings. Ed. by Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio. Lecture Notes in Computer Science 4199. Springer, 2006, pp. 440–453. isbn: 3-540-45772-0. doi: 10.1007/11880240_31 (cit. on pp. 8, 114, 120, 146).

[BFPR02] Annalisa Bossi, Riccardo Focardi, Carla Piazza, and Sabina Rossi. "Transforming Processes to Check and Ensure Information Flow Security". In: *Algebraic Methodology and Software Technology*. 9th International Conference. AMAST 2002 (Saint-Gilles-les-Bains, Réunion Island, France, Sept. 9–13, 2002). Proceedings. Ed. by Hélène Kirchner and Christophe Ringeissen. Lecture Notes in Computer Science 2422. Springer, 2002, pp. 271–286. isbn: 3-540-44144-1. doi: 10.1007/3-540-45719-4_19 (cit. on pp. 94, 98, 108).

[BGN17] Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. "A Survey of Timing Channels and Countermeasures". In: *ACM Computing Surveys* 50.1 (Apr. 2017), 6:1–6:39. issn: 0360-0300. doi: 10.1145/3023872 (cit. on pp. 6, 29, 146).

[BGTC18] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. “Distributing relational model transformation on MapReduce”. In: *Journal of Systems and Software* 142 (Aug. 2018), pp. 1–20. issn: 0164-1212. doi: 10.1016/j.jss.2018.04.014 (cit. on pp. 128, 130, 131).

[BKP20] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. “A survey of modeling language specification techniques”. In: *Information Systems* 87, Art. No. 101425 (Jan. 2020), pp. 1–20. issn: 0306-4379. doi: 10.1016/j.is.2019.101425 (cit. on p. 10).

[Bod18] Eric Bodden. “State of the systems security”. In: *2018 ACM/IEEE 40th International Conference on Software Engineering*. ICSE-Companion 2018 (Gothenburg, Sweden, May 27–June 3, 2018). Companion Proceeedings. Ed. by Michel R. V. Chaudron, Ivica Crnković, Marsha Chechik, and Mark Harman. ACM, 2018, pp. 550–551. isbn: 978-1-4503-5663-3. doi: 10.1145/3183440.3183462 (cit. on p. 4).

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981. isbn: 0-13-822122-7 (cit. on pp. 45, 47).

[Bos+04] Annalisa Bossi, Riccardo Focardi, Damiano Macedonio, Carla Piazza, and Sabina Rossi. “Unwinding in Information Flow Security”. In: *Electronic Notes in Theoretical Computer Science* 99 (Aug. 2004): *Proceedings of the MEFISTO Project 2003. Formal Methods for Security and Time*. MEFISTO 2003. Ed. by Mario Bravetti and Roberto Gorrieri, pp. 127–154. issn: 1571-0661. doi: 10.1016/j.entcs.2004.02.006 (cit. on p. 108).

[BPKN18] Aimee Borda, Liliana Pasquale, Vasileios Koutavas, and Bashar Nuseibeh. “Compositional verification of self-adaptive cyber-physical systems”. In: *2018 ACM/IEEE 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2018 (Gothenburg, Sweden, May 28–29, 2018). Proceedings. Ed. by Jesper Andersson and Danny Weyns. ACM, 2018, pp. 1–11. isbn: 978-1-4503-5715-9. doi: 10.1145/3194133.3194146 (cit. on pp. 56, 81).

[BPS18] Iulia Bastys, Frank Piessens, and Andrei Sabelfeld. “Prudent Design Principles for Information Flow Control”. In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*. PLAS’18 (Toronto, Canada, Oct. 19, 2018). Ed. by Mário S. Alvim and Stéphanie Delaune. ACM, 2018, pp. 17–23. isbn: 978-1-4503-5993-1. doi: 10.1145/3264820.3264824 (cit. on pp. 48, 59).

[Bre10] Christian Brenner. “Analyse von mechatronischen Systemen mittels Testautomaten”. German. MA thesis. University of Paderborn, Aug. 2010. url: https://www.hni.uni-paderborn.de/pub/7318 (cit. on p. 105).

[Bru+20] Jean-Michel Bruel et al. "Comparing and classifying model transformation reuse approaches across metamodels". In: *Software and Systems Modeling* 19.2 (Mar. 2020), pp. 441–465. ISSN: 1619-1366. DOI: 10.1007/s10270-019-00762-9 (cit. on pp. 118, 141).

[BS16] Islem Baki and Houari A. Sahraoui. "Multi-Step Learning and Adaptive Search for Learning Complex Model Transformations from Examples". In: *ACM Transactions on Software Engineering and Methodology* 25.3, Art. No. 20 (Aug. 2016), pp. 1–37. ISSN: 1049-331X. DOI: 10.1145/2904904 (cit. on pp. 142, 148).

[BSA17] Ilhem Boussaïd, Patrick Siarry, and Mohamed Ahmed-Nacer. "A survey on search-based model-driven engineering". In: *Automated Software Engineering. An International Journal* 24.2 (June 2017), pp. 233–294. ISSN: 0928-8910. DOI: 10.1007/s10515-017-0215-4 (cit. on pp. 139, 142, 148).

[BSS17] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. "We Are Family. Relating Information-Flow Trackers". In: *Computer Security. ESORICS 2017.* Proceedings. Part I. 22nd European Symposium on Research in Computer Security (Oslo, Norway, Sept. 11–15, 2017). Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Lecture Notes in Computer Science 10492. Springer, 2017, pp. 124–145. ISBN: 978-3-319-66401-9. DOI: 10.1007/978-3-319-66402-6_9 (cit. on p. 149).

[BSYJ17] Alexander van den Berghe, Riccardo Scandariato, Koen Yskout, and Wouter Joosen. "Design notations for secure software. A systematic literature review". In: *Software and Systems Modeling* 16.3 (July 2017), pp. 809–831. ISSN: 1619-1366. DOI: 10.1007/s10270-015-0486-9 (cit. on p. 4).

[BT03] Roberto Barbuti and Luca Tesei. "A Decidable Notion of Timed Non-Interference". In: *Fundamenta Informaticae* 54.2–3 (2003): *Concurrency Specification and Programming. CS&P'2002*. Part 1, pp. 137–150. ISSN: 0169-2968. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi54-2-3-03 (cit. on pp. 86, 111).

[Bur02] Sven Burmester. "Generierung von Java Real-Time Code für zeitbehaftete UML Modelle". German. Diploma thesis. University of Paderborn, Sept. 2002 (cit. on p. 22).

[BVJM13] Verónica Andrea Bollati, Juan Manuel Vara, Álvaro Jiménez, and Esperanza Marcos. "Applying MDE to the (semi-)automatic development of model transformations". In: *Information and Software Technology* 55.4 (Apr. 2013), pp. 699–718. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.11.004 (cit. on pp. 117, 141).

[BW18] Jürgen Beyerer and Petra Winzer, eds. *Beiträge zu einer Systemtheorie Sicherheit*. German. acatech DISKUSSION. Herbert Utz, 2018. ISBN: 978-3-8316-4624-1. URL: https://www.acatech.de/publikation/beitraege-zu-einer-systemtheorie-sicherheit (cit. on p. 148).

[BY04] Johan Bengtsson and Wang Yi. "Timed Automata. Semantics, Algorithms and Tools". In: *Lectures on Concurrency and Petri Nets. Advances in Petri Nets*. 4th Advanced Course on Petri Nets. ACPN 2003 (Eichstätt, Germany, Sept. 15–26, 2003). Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Lecture Notes in Computer Science 3098: Tutorial. Springer, 2004, pp. 87–124. ISBN: 3-540-22261-8. DOI: 10.1007/978-3-540-27755-2_3 (cit. on pp. 23, 24).

[Cad11] Tom Caddy. *Side-Channel Attacks*. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. 2nd ed. Springer, 2011, p. 1204. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_227 (cit. on pp. 28, 49).

[Cas09] Franck Cassez. "The Dark Side of Timed Opacity". In: *Advances in Information Security and Assurance*. Third International Conference and Workshops. ISA 2009 (Seoul, South Korea, June 25–27, 2009). Proceedings. Ed. by Jong Hyuk Park, Hsiao-Hwa Chen, Mohammed Atiquzzaman, Changhoon Lee, Tai-Hoon Kim, and Sang-Soo Yeo. Lecture Notes in Computer Science 5576. Springer, 2009, pp. 21–30. ISBN: 978-3-642-02616-4. DOI: 10.1007/978-3-642-02617-1_3 (cit. on pp. 82, 86, 110).

[CCBG18] Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard. "Cognifying Model-Driven Software Engineering". In: *Software Technologies. Applications and Foundations*. STAF 2017 Collocated Workshops (Marburg, Germany, July 17–21, 2017). Revised Selected Papers. Ed. by Martina Seidl and Steffen Zschaler. Lecture Notes in Computer Science 10748. Springer, 2018, pp. 154–160. ISBN: 978-3-319-74729-3. DOI: 10.1007/978-3-319-74730-9_13 (cit. on pp. 142, 148).

[CCP19] Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. "Multi-view approaches for software and system modelling. A systematic literature review". In: *Software and Systems Modeling* 18.6 (Dec. 2019), pp. 3207–3233. ISSN: 1619-1366. DOI: 10.1007/s10270-018-00713-w (cit. on p. 14).

[Čer93] Kārlis Čerāns. "Decidability of Bisimulation Equivalences for Parallel Timer Processes". In: *Computer Aided Verification*. Fourth International Workshop. CAV '92 (Montreal, Canada, June 29–July 1, 1992). Proceedings. Ed. by Gregor von Bochmann and David K. Probst. Lecture Notes in Computer Science 663. Springer, 1993, pp. 302–315. ISBN: 3-540-56496-9. DOI: 10.1007/3-540-56496-9_24 (cit. on pp. 85, 105, 109).

[CFSS16] Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. "Perspectives of Model Transformation Reuse". In: *Integrated Formal Methods*. 12th International Conference. IFM 2016 (Reykjavík, Iceland, June 1–5, 2016). Proceedings. Ed. by Erika Ábrahám and Marieke Huisman. Lecture Notes in Computer Science 9681. Springer, 2016, pp. 28–44. isbn: 978-3-319-33692-3. doi: 10.1007/978-3-319-33693-0_3 (cit. on pp. 118, 141).

[CH06] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–646. issn: 0018-8670. doi: 10.1147/sj.453.0621 (cit. on pp. 6, 12, 13, 119).

[Che+09] Betty H. C. Cheng et al. "Software Engineering for Self-Adaptive Systems. A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems* (Dagstuhl Castle, Jan. 13–18, 2008). Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee. Lecture Notes in Computer Science 5525: State-of-the-Art Survey. Springer, 2009, pp. 1–26. isbn: 978-3-642-02160-2. doi: 10.1007/978-3-642-02161-9_1 (cit. on p. 14).

[CHQ16] Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. "On the diversity of asynchronous communication". In: *Formal Aspects of Computing. Applicable Formal Methods* 28.5 (Sept. 2016): *Papers on Formal Engineering Methods including Extended Versions of papers presented at ICFEM 2014. Part 2*, pp. 847–879. issn: 0934-5043. doi: 10.1007/s00165-016-0379-x (cit. on pp. 5, 18).

[CILN02] Robert Crook, Darrel C. Ince, Luncheng Lin, and Bashar Nuseibeh. "Security Requirements Engineering. When Anti-Requirements Hit the Fan". In: *IEEE Joint International Conference on Requirements Engineering*. RE 2002 (Essen, Germany, Sept. 9–13, 2002). Proceedings. IEEE Computer Society, 2002, pp. 203–205. isbn: 0-7695-1465-0. doi: 10.1109/ICRE.2002.1048527 (cit. on p. 4).

[CM15] Stephen Chong and Ron van der Meyden. "Using Architecture to Reason about Information Security". In: *ACM Transactions on Information and System Security* 18.2, Art. No. 8 (Dec. 2015), pp. 1–30. issn: 1094-9224. doi: 10.1145/2829949 (cit. on pp. 32, 56, 82).

[CMMS16] Ivica Crnković, Ivano Malavolta, Henry Muccini, and Mohammad Sharaf. "On the Use of Component-Based Principles and Practices for Architecting Cyber-Physical Systems". In: *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE 2016 (Venice, Italy, Apr. 5–8, 2016). Proceedings. IEEE Computer Society, 2016, pp. 23–32. isbn: 978-1-5090-2569-5. doi: 10.1109/CBSE.2016.9 (cit. on pp. 5, 55).

[Coh77] Ellis S. Cohen. “Information Transmission in Computational Systems”. In: *Proceedings of Sixth ACM Symposium on Operating Systems Principles*. SOSP 1977 (West Lafayette, Indiana, USA, Nov. 16–18, 1977). Ed. by Saul Rosen and Peter J. Denning. ACM, 1977, pp. 133–139. doi: 10.1145/800214.806556 (cit. on p. 29).

[CPS17] Anupam Chattopadhyay, Alok Prakash, and Muhammad Shafique. “Secure Cyber-Physical Systems. Current trends, tools and open research problems”. In: *Proceedings of the 2017 Design, Automation & Test in Europe*. DATE 2017 (Lausanne, Switzerland, Mar. 27–31, 2017). Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 1104–1109. isbn: 978-3-9815370-8-6. doi: 10.23919/DATE.2017.7927154 (cit. on p. 1).

[CS10] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties”. In: *Journal of Computer Security* 18.6 (2010): *21st IEEE Computer Security Foundations Symposium. CSF’08*. Ed. by Andrei Sabelfeld, pp. 1157–1210. issn: 0926-227X. doi: 10.3233/JCS-2009-0393 (cit. on pp. 31, 32, 85).

[CSVC11] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron. “A Classification Framework for Software Component Models”. In: *IEEE Transactions on Software Engineering* 37.5 (Sept.–Oct. 2011), pp. 593–615. issn: 0098-5589. doi: 10.1109/TSE.2010.83 (cit. on pp. 18, 20, 21, 55, 57, 58).

[DBHT12] Stefan Dziwok, Kathrin Bröker, Christian Heinzemann, and Matthias Tichy. *A Catalog of Real-Time Coordination Patterns for Advanced Mechatronic Systems*. Tech. rep. tr-ri-12-319. University of Paderborn. Feb. 2012. url: https://www.hni.uni-paderborn.de/pub/6888 (cit. on pp. 25, 27).

[DD77] Dorothy E. Denning and Peter J. Denning. “Certification of Programs for Secure Information Flow”. In: *Communications of the ACM* 20.7 (July 1977), pp. 504–513. issn: 0001-0782. doi: 10.1145/359636.359712 (cit. on p. 28).

[Del+14] Michael Dellnitz et al. “The Paradigm of Self-optimization”. In: *Design Methodology for Intelligent Technical Systems. Develop Intelligent Technical Systems of the Future*. Ed. by Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer. Lecture Notes in Mechanical Engineering. Springer, 2014. Chap. 1, pp. 1–25. isbn: 978-3-642-45434-9. doi: 10.1007/978-3-642-45435-6_1 (cit. on p. 15).

[DGB14] Stefan Dziwok, Sebastian Goschin, and Steffen Becker. “Specifying Intra-Component Dependencies for Synthesizing Component Behaviors”. In: *Model-Driven Engineering for Component-Based Software Systems*. ModComp 2014 (Valencia, Spain, Sept. 29, 2014). Proceedings of the 1st International Workshop on Model-Driven Engineering for Component-Based Software Systems. Ed. by Federico Ciccozzi, Massimo Tivoli, and Jan Carlson. CEUR Workshop Proceedings 1281. issn: 1613-0073. 2014, pp. 16–25. urn: urn:nbn:de:0074-1281-7 (cit. on pp. 27, 107).

[DGC17] Zinovy Diskin, Abel Gómez, and Jordi Cabot. “Traceability Mappings as a Fundamental Instrument in Model Transformations”. In: *Fundamental Approaches to Software Engineering*. 20th International Conference. FASE 2017 (Uppsala, Sweden, Apr. 26–28, 2017). Proceedings. Ed. by Marieke Huisman and Julia Rubin. Lecture Notes in Computer Science 10202: Advanced Research in Computing and Software Science. Springer, 2017, pp. 247–263. ISBN: 978-3-662-54493-8. DOI: 10.1007/978-3-662-54494-5_14 (cit. on pp. 117, 141).

[DHRS11] Deepak D’Souza, Raveendra Holla, K. R. Raghavendra, and Barbara Sprick. “Model-checking trace-based information flow properties”. In: *Journal of Computer Security* 19.1 (2011), pp. 101–138. ISSN: 0926-227X. DOI: 10.3233/JCS-2010-0400 (cit. on pp. 71, 86, 109).

[DHS05] Ádám Darvas, Reiner Hähnle, and David Sands. “A Theorem Proving Approach to Analysis of Secure Information Flow”. In: *Security in Pervasive Computing*. Second International Conference. SPC 2005 (Boppard, Germany, Apr. 6–8, 2005). Proceedings. Ed. by Dieter Hutter and Markus Ullmann. Lecture Notes in Computer Science 3450. Springer, 2005, pp. 193–209. ISBN: 3-540-25521-4. DOI: 10.1007/978-3-540-32004-3_20 (cit. on pp. 86, 109).

[Dim+12] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. “Model Checking Information Flow in Reactive Systems”. In: *Verification, Model Checking, and Abstract Interpretation*. 13th International Conference. VMCAI 2012 (Philadelphia, Pennsylvania, USA, Jan. 22–24, 2012). Proceedings. Ed. by Viktor Kuncak and Andrey Rybalchenko. Lecture Notes in Computer Science 7148. Springer, 2012, pp. 169–185. ISBN: 978-3-642-27939-3. DOI: 10.1007/978-3-642-27940-9_12 (cit. on pp. 86, 109).

[DKV00] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages. An Annotated Bibliography”. In: *ACM SIGPLAN Notices* 35.6 (June 2000). Ed. by Jay Fenwick and Cindy Norris, pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035 (cit. on pp. 1, 10).

[DKW08] Vijay D’Silva, Daniel Kröning, and Georg Weissenbacher. “A Survey of Automated Techniques for Formal Software Verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410 (cit. on p. 2).

[DLS12] Patricia Derler, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli. “Modeling Cyber-Physical Systems”. In: *Proceedings of the IEEE* 100.1 (Jan. 2012): *Cyber-Physical Systems*. Ed. by Radha Poovendran, Krishna Sampigethaya, Sandeep Kumar S. Gupta, Insup Lee, K. Venkatesh Prasad, David Corman, and James L. Paunicka, pp. 13–28. ISSN: 0018-9219. DOI: 10.1109/JPROC.2011.2160929 (cit. on p. 1).

[Dra+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. “Microservices. Yesterday, Today, and Tomorrow”. In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Springer, 2017, pp. 195–216. ISBN: 978-3-319-67424-7. DOI: 10.1007/978-3-319-67425-4_12 (cit. on pp. 55, 79).

[DS00] Premkumar T. Devanbu and Stuart G. Stubblebine. “Software engineering for security. A roadmap”. In: *The Future of Software Engineering*. 22nd International Conference on Software Engineering. ICSE 2000 (Limerick, Ireland, June 4–11, 2000). Ed. by Anthony Finkelstein. ACM, 2000, pp. 227–239. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336559 (cit. on p. 1).

[Dwo11] Cynthia Dwork. *Differential Privacy*. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. 2nd ed. Springer, 2011, pp. 338–340. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_752 (cit. on p. 107).

[DZGL18] Pengfei Duan, Ying Zhou, Xufang Gong, and Bixin Li. “A Systematic Mapping Study on the Verification of Cyber-Physical Systems”. In: *IEEE Access* 6 (2018), pp. 59043–59064. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2872015 (cit. on p. 2).

[Dzi17] Stefan Dziwok. “Specification and verification for real-time coordination protocols of cyber-physical systems”. PhD thesis. Paderborn University, Sept. 2017. DOI: 10.17619/UNIPB/1-196 (cit. on pp. 19, 25–27, 78).

[EH10] Tobias Eckardt and Stefan Henkler. “Component Behavior Synthesis for Critical Systems”. In: *Architecting Critical Systems*. First International Symposium. ISARCS 2010 (Prague, Czech Republic, June 23–25, 2010). Proceedings. Ed. by Holger Giese. Lecture Notes in Computer Science 6150. Springer, 2010, pp. 52–71. ISBN: 978-3-642-13555-2. DOI: 10.1007/978-3-642-13556-9_4 (cit. on pp. 27, 107).

[EKSS18] Jürgen Etzlstorfer, Elisabeth Kapsammer, Wieland Schwinger, and Johannes Schönböck. “Surveying Co-evolution in Modeling Ecosystems”. In: *Model-Driven Engineering and Software Development*. 5th International Conference. MODELSWARD 2017 (Porto, Portugal, Feb. 19–21, 2017). Revised Selected Papers. Ed. by Luís Ferreira Pires, Slimane Hammoudi, and Bran Selic. Communications in Computer and Information Science 880. Springer, 2018, pp. 354–376. ISBN: 978-3-319-94763-1. DOI: 10.1007/978-3-319-94764-8_15 (cit. on p. 143).

[EYLL11] Golnaz Elahi, Eric S. K. Yu, Tong Li, and Lin Liu. “Security Requirements Engineering in the Wild. A Survey of Common Practices”. In: *35th Annual IEEE International Computer Software and Applications Conference*. Vol. I: *Main Conference*. COMPSAC 2011 (Munich, Germany, July 18–22, 2011). Proceedings. IEEE Computer Society, 2011, pp. 314–319. isbn: 978-0-7695-4439-7. doi: 10.1109/COMPSAC.2011.48 (cit. on p. 4).

[Fab+10] Benjamin Fabian, Seda F. Gürses, Maritta Heisel, Thomas Santen, and Holger Schmidt. “A comparison of security requirements engineering methods”. In: *Requirements Engineering* 15.1 (Mar. 2010): *Special Issue on RE’09. Security Requirements Engineering*. Ed. by Eric Dubois and Haralambos Mouratidis, pp. 7–40. issn: 0947-36020. doi: 10.1007/s00766-009-0092-x (cit. on p. 4).

[FB16] Matthias Freund and Annerose Braune. “A generic transformation algorithm to simplify the development of mapping models”. In: *19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS’16 (Saint-Malo, France, Oct. 2–7, 2016). Proceedings. Ed. by Benoit Baudry and Benoît Combemale. ACM, 2016, pp. 284–294. isbn: 978-1-4503-4321-3. doi: 10.1145/2976767.2976777 (cit. on pp. 114, 117, 118, 135, 139, 141, 142).

[Fel+14] Michael Felderer et al. “Evolution of Security Engineering Artifacts. A State of the Art Survey”. In: *International Journal of Secure Software Engineering* 5.4 (Oct.–Dec. 2014), pp. 48–98. issn: 1947-3036. doi: 10.4018/ijsse.2014100103 (cit. on p. 148).

[FG95] Riccardo Focardi and Roberto Gorrieri. “A Taxonomy of Security Properties for Process Algebras”. In: *Journal of Computer Security* 3.1 (1995). Ed. by Li Gong, pp. 5–34. issn: 0926-227X. doi: 10.3233/JCS-1994/1995-3103 (cit. on pp. 2, 29, 60, 73).

[FGM03] Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. “Real-time information flow analysis”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (Jan. 2003). Ed. by Li Gong, Joshua D. Guttman, Peter Y. A. Ryan, and Steve A. Schneider, pp. 20–35. issn: 0733-8716. doi: 10.1109/JSAC.2002.806122 (cit. on p. 110).

[FLM19] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. “Architecting with microservices. A systematic mapping study”. In: *Journal of Systems and Software* 150 (Apr. 2019), pp. 77–97. issn: 0164-1212. doi: 10.1016/j.jss.2019.01.001 (cit. on p. 55).

[FLV14] John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. “From Embedded to Cyber-Physical Systems. Challenges and Future Directions”. In: *Collaborative Design for Embedded Systems. Co-modelling and Co-simulation*. Ed. by John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. Springer, 2014.

Chap. 14, pp. 293–303. isbn: 978-3-642-54117-9. doi: 10.1007/978-3-642-54118-6_14 (cit. on p. 1).

[FPN20] Daniel Frassinelli, Sohyeon Park, and Stefan Nürnberger. "«I Know Where You Parked Last Summer». Automated Reverse Engineering and Privacy Analysis of Modern Cars". In: *2020 IEEE Symposium on Security and Privacy*. SP 2020 (San Francisco, California, USA, May 18–21, 2020). Proceedings. IEEE, 2020, pp. 1401–1415. isbn: 978-1-7281-3497-0. doi: 10.1109/SP40000.2020.00081 (cit. on p. 3).

[FPR02] Riccardo Focardi, Carla Piazza, and Sabina Rossi. "Proofs Methods for Bisimulation Based Information Flow Security". In: *Verification, Model Checking, and Abstract Interpretation*. Third International Workshop. VMCAI 2002 (Venice, Italy, Jan. 21–22, 2002). Revised Papers. Ed. by Agostino Cortesi. Lecture Notes in Computer Science 2294. Springer, 2002, pp. 16–31. isbn: 3-540-43631-6. doi: 10.1007/3-540-47813-2_2 (cit. on p. 109).

[FRS05] Riccardo Focardi, Sabina Rossi, and Andrei Sabelfeld. "Bridging Language-Based and Process Calculi Security". In: *Foundations of Software Science and Computational Structures*. 8th International Conference. FOSSACS 2005 (Edinburgh, UK, Apr. 4–8, 2005). Proceedings. Ed. by Vladimiro Sassone. Lecture Notes in Computer Science 3441. Springer, 2005, pp. 299–315. isbn: 3-540-25388-2. doi: 10.1007/978-3-540-31982-5_19 (cit. on pp. 2, 109).

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. "Algorithms for Model Checking HyperLTL and HyperCTL*". In: *Computer Aided Verification*. Proceedings. Part I. 27th International Conference. CAV 2015 (San Francisco, California, USA, July 18–24, 2015). Ed. by Daniel Kröning and Corina S. Păsăreanu. Lecture Notes in Computer Science 9206. Springer, 2015, pp. 30–48. isbn: 978-3-319-21689-8. doi: 10.1007/978-3-319-21690-4_3 (cit. on pp. 86, 109).

[FV09] Marcos Didonet Del Fabro and Patrick Valduriez. "Towards the efficient development of model transformations using model weaving and matching transformations". In: *Software and Systems Modeling* 8.3 (July 2009), pp. 305–324. issn: 1619-1366. doi: 10.1007/s10270-008-0094-z (cit. on p. 135).

[Gab+19] Sebastian Gabmeyer, Petra Kaufmann, Martina Seidl, Martin Gogolla, and Gerti Kappel. "A feature-based classification of formal verification techniques for software models". In: *Software and Systems Modeling* 18.1 (Feb. 2019): *Theme Sections on "Model-Driven Engineering for Component-Based Software Engineering" and "STAF 2015"*, pp. 473–498. issn: 1619-1366. doi: 10.1007/s10270-017-0591-z (cit. on p. 2).

[Gar14] David Garlan. "Software architecture. A travelogue". In: *Future of Software Engineering. FOSE 2014*. 36th International Conference on Software Engineering. ICSE '14 (Hyderabad, India, May 31–June 7, 2014). Proceedings. Ed. by James D. Herbsleb and Matthew B. Dwyer. ACM, 2014, pp. 29–39.

isbn: 978-1-4503-2865-4. doi: 10.1145/2593882.2593886 (cit. on pp. 5, 18).

[Gau+09] Jürgen Gausemeier, Wilhelm Schäfer, Joel Greenyer, Sascha Kahl, Sebastian Pook, and Jan Rieke. “Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems”. In: *Proceedings of the 17th International Conference on Engineering Design*. Vol. 6.2: *Design Methods and Tools*. ICED’09 (Palo Alto, California, USA, Aug. 24–27, 2009). Ed. by Margareta Norell Bergendahl, Martin Grimheden, Larry Leifer, Philipp Skogstad, and Udo Lindemann. DS 58-6. Design Society, 2009, pp. 1–12. isbn: 978-1-904670-10-0 (cit. on pp. 18, 62).

[Gau+14] Jürgen Gausemeier, Sebastian Korf, Mario Porrmann, Katharina Stahl, Oliver Sudmann, and Mareen Vaßholz. “Development of Self-optimizing Systems”. In: *Design Methodology for Intelligent Technical Systems. Develop Intelligent Technical Systems of the Future*. Ed. by Jürgen Gausemeier, Franz-Josef Rammig, and Wilhelm Schäfer. Lecture Notes in Mechanical Engineering. Springer, 2014. Chap. 3, pp. 65–115. isbn: 978-3-642-45434-9. doi: 10.1007/978-3-642-45435-6_3 (cit. on p. 14).

[GB03] Holger Giese and Sven Burmester. *Real-Time Statechart Semantics*. Tech. rep. tr-ri-03-239. University of Paderborn. June 2003. url: https://www.hni.uni-paderborn.de/pub/6360 (cit. on p. 22).

[GB20] Johannes Geismann and Eric Bodden. “A systematic literature review of model-driven security engineering for cyber-physical systems”. In: *Journal of Systems and Software* 169, Art. No. 110697 (Nov. 2020). issn: 0164-1212. doi: 10.1016/j.jss.2020.110697. In press (cit. on p. 4).

[GFDK09] Jürgen Gausemeier, Ursula Frank, Jörg Donoth, and Sascha Kahl. “Specification technique for the description of self-optimizing mechatronic systems”. In: *Research in Engineering Design* 20.4 (Nov. 2009), pp. 201–223. issn: 0934-9839. doi: 10.1007/s00163-008-0058-x (cit. on p. 14).

[GH17] Simon Greiner and Mihai Herda. *CoCoME with Security*. Tech. rep. Karlsruhe Reports in Informatics 2017,2. issn: 2190-4782. Karlsruhe Institute of Technology. 2017. doi: 10.5445/IR/1000065106 (cit. on pp. 86, 87, 100, 101, 146).

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-oriented Software*. With a forew. by Grady Booch. Addison-Wesley Longman, 1994. isbn: 0-201-63361-2 (cit. on pp. 127, 132, 133).

[Gir+17] Jairo Giraldo, Esha Sarkar, Alvaro A. Cárdenas, Michail Maniatakos, and Murat Kantarcioglu. “Security and Privacy in Cyber-Physical Systems. A Survey of Surveys”. In: *IEEE Design & Test* 34.4 (July–Aug. 2017): *Special Issue on Cyber-Physical Systems Security and Privacy*. Ed. by Michail Maniatakos, Alvaro A. Cárdenas, and Ramesh Karri, pp. 7–17. issn: 2168-2356. doi: 10.1109/MDAT.2017.2709310 (cit. on p. 1).

[GJ08] Lars Grunske and David Joyce. "Quantitative risk-based security prediction for component-based systems with explicitly modeled attack profiles". In: *Journal of Systems and Software* 81.8 (Aug. 2008), pp. 1327–1345. issn: 0164-1212. doi: 10.1016/j.jss.2007.11.716 (cit. on p. 51).

[GK16] Dieter Gollmann and Marina Krotofil. "Cyber-Physical Systems Security". In: *The New Codebreakers. Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*. Ed. by Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater. Lecture Notes in Computer Science 9100: Festschrift. Springer, 2016, pp. 195–204. isbn: 978-3-662-49300-7. doi: 10.1007/978-3-662-49301-4_14 (cit. on p. 1).

[GKHE18] Adnane Ghannem, Marouane Kessentini, Mohammad Salah Hamdi, and Ghizlane El-Boussaidi. "Model refactoring by example. A multi-objective search based software engineering approach". In: *Journal of Software. Evolution and Process* 30.4, Art. No. e1916 (Apr. 2018), pp. 1–20. issn: 2047-7473. doi: 10.1002/smr.1916 (cit. on pp. 142, 148).

[GM05] Roberto Giacobazzi and Isabella Mastroeni. "Timed Abstract Non-interference". In: *Formal Modeling and Analysis of Timed Systems*. Third International Conference. FORMATS 2005 (Uppsala, Sweden, Sept. 26–28, 2005). Proceedings. Ed. by Paul Pettersson and Wang Yi. Lecture Notes in Computer Science 3829. Springer, 2005, pp. 289–303. isbn: 3-540-30946-2. doi: 10.1007/11603009_22 (cit. on p. 110).

[GM82] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy* (Oakland, California, USA, Apr. 26–28, 1982). IEEE Computer Society, 1982, pp. 11–20. isbn: 0-8186-0410-7. doi: 10.1109/SP.1982.10014 (cit. on pp. 2, 29, 91).

[GM84] Joseph A. Goguen and José Meseguer. "Unwinding and Inference Control". In: *1984 IEEE Symposium on Security and Privacy* (Oakland, California, USA, Apr. 29–May 2, 1984). IEEE Computer Society, 1984, pp. 75–87. isbn: 0-8186-0532-4. doi: 10.1109/SP.1984.10019 (cit. on pp. 86, 108).

[GMB17] Simon Greiner, Martin Mohr, and Bernhard Beckert. "Modular Verification of Information Flow Security in Component-Based Systems". In: *Software Engineering and Formal Methods. 15th International Conference*. SEFM 2017 (Trento, Italy, Sept. 4–8, 2017). Proceedings. Ed. by Alessandro Cimatti and Marjan Sirjani. Lecture Notes in Computer Science 10469. Springer, 2017, pp. 300–315. isbn: 978-3-319-66196-4. doi: 10.1007/978-3-319-66197-1_19 (cit. on pp. 56, 82, 105).

[Gor+04] Roberto Gorrieri, Ruggero Lanotte, Andrea Maggiolo-Schettini, Fabio Martinelli, Simone Tini, and Enrico Tronci. "Automated analysis of timed security. A case study on web privacy". In: *International Journal of Information Security* 2.3–4 (2004): *Special issue on security in global computing*, pp. 168–186. issn: 1615-5262. doi: 10.1007/s10207-004-0037-9 (cit. on pp. 82, 111).

[Gra+18] Imen Graja, Slim Kallel, Nawal Guermouche, Saoussen Cheikhrouhou, and Ahmed Hadj Kacem. "A comprehensive survey on modeling of cyber-physical systems". In: *Concurrency and Computation. Practice and Experience*, Art. No. e4850 (Oct. 2018), pp. 1–18. issn: 1532-0634. doi: 10.1002/cpe.4850 (cit. on p. 1).

[Gre18] Simon Greiner. "A Framework for Non-Interference in Component-Based Systems". PhD thesis. Karlsruhe Institute of Technology, Jan. 2018. doi: 10.5445/IR/1000082042 (cit. on p. 82).

[GS13] Holger Giese and Wilhelm Schäfer. "Model-Driven Development of Safe Self-optimizing Mechatronic Systems with MechatronicUML". In: *Assurances for Self-Adaptive Systems. Principles, Models, and Techniques*. Ed. by Javier Cámara, Rogério de Lemos, Carlo Ghezzi, and Antónia Lopes. Lecture Notes in Computer Science 7740: State-of-the-Art Survey. Springer, 2013, pp. 152–186. isbn: 978-3-642-36248-4. doi: 10.1007/978-3-642-36249-1_6 (cit. on p. 18).

[GTB18] Raffaela Groner, Matthias Tichy, and Steffen Becker. "Towards Performance Engineering of Model Transformation". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE 2018 (Berlin, Germany, Apr. 9–13, 2018). Ed. by Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar. ACM, 2018, pp. 33–36. isbn: 978-1-4503-5629-9. doi: 10.1145/3185768.3186305 (cit. on p. 118).

[GTBF03] Holger Giese, Matthias Tichy, Sven Burmester, and Stephan Flake. "Towards the compositional verification of real-time UML designs". In: *Proceedings of the Joint 9th European Software Engineering Conference (ESEC) & 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11)*. ESEC/FSE 2003 (Helsinki, Finland, Sept. 1–5, 2003). Ed. by Paola Inverardi. ACM, 2003, pp. 38–47. isbn: 978-1-58113-743-9. doi: 10.1145/940071.940078 (cit. on pp. 18–20).

[Gue+13] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos. "Engineering model transformations with *trans*ML". In: *Software and Systems Modeling* 12.3 (July 2013), pp. 555–577. issn: 1619-1366. doi: 10.1007/s10270-011-0211-2 (cit. on pp. 117, 141).

[Hac+17] Jamal El Hachem, Tarek Al Khalil, Vanea Chiprianov, Ali Babar, and Philippe Aniorté. "A Model Driven Method to Design and Analyze Secure Architectures of Systems-of-Systems". In: *2017 22nd International Conference on Engineering of Complex Computer Systems*. ICECCS 2017 (Fukuoka, Japan, Nov. 6–8, 2017). Proceedings. IEEE Computer Society, 2017, pp. 166–169. isbn: 978-1-5386-2431-9. doi: 10.1109/ICECCS.2017.31 (cit. on p. 52).

[HBDS15] Christian Heinzemann, Christian Brenner, Stefan Dziwok, and Wilhelm Schäfer. “Automata-based refinement checking for real-time systems”. In: *Computer Science. Research and Development* 30.3–4 (Aug. 2015): *SE 2013*, pp. 255–283. issn: 2524-8510. doi: 10.1007/s00450-014-0257-9 (cit. on pp. 8, 20, 27, 86, 89, 90, 92–94, 97, 105, 106, 111, 146).

[HBV19] Christian Heinzemann, Steffen Becker, and Andreas Volk. “Transactional execution of hierarchical reconfigurations in cyber-physical systems”. In: *Software and Systems Modeling* 18.1 (Feb. 2019): *Theme Sections on "Model-Driven Engineering for Component-Based Software Engineering" and "STAF 2015"*, pp. 157–189. issn: 1619-1366. doi: 10.1007/s10270-017-0583-z (cit. on pp. 79, 147).

[Heb+18] Regina Hebig, Christoph Seidl, Thorsten Berger, John Kook Pedersen, and Andrzej Wąsowski. “Model transformation languages under a magnifying glass. A controlled experiment with Xtend, ATL, and QVT”. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE ’18 (Lake Buena Vista, Florida, USA, Nov. 4–9, 2018). Ed. by Gary T. Leavens, Alessandro Garcia, and Corina S. Păsăreanu. ACM, 2018, pp. 445–455. isbn: 978-1-4503-5573-5. doi: 10.1145/3236024.3236046 (cit. on pp. 6, 13, 113, 132, 134, 146).

[Hei15] Christian Heinzemann. “Verification and simulation of self-adaptive mechatronic systems”. PhD thesis. University of Paderborn, Sept. 2015. urn: urn:nbn:de:hbz:466:2-16778 (cit. on pp. 18, 20, 27, 79, 86, 89, 92, 106).

[Hei18] Jens Heinen. “Model Checking Timed Hyperproperties”. MA thesis. Saarbrücken: Saarland University, June 2018. url: https://www.react.uni-saarland.de/publications/Heinen18.html (cit. on pp. 86, 111).

[Hel+16] Rogardt Heldal, Patrizio Pelliccione, Ulf Eliasson, Jonn Lantz, Jesper Derehag, and Jon Whittle. “Descriptive vs prescriptive models in industry”. In: *19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS’16 (Saint-Malo, France, Oct. 2–7, 2016). Proceedings. Ed. by Benoit Baudry and Benoît Combemale. ACM, 2016, pp. 216–226. isbn: 978-1-4503-4321-3. doi: 10.1145/2976767.2976808 (cit. on p. 9).

[Her+08] Sebastian Herold et al. “CoCoME. The Common Component Modeling Example”. In: *The Common Component Modeling Example. Comparing Software Component Models*. Modelling Contest (Dagstuhl Castle, Germany, Aug. 1–3, 2007). Ed. by Andreas Rausch, Ralf H. Reussner, Raffaela Mirandola, and Frantisek Plasil. Lecture Notes in Computer Science 5153: Tutorial. Springer,

2008. Chap. 3, pp. 16–53. isbn: 978-3-540-85288-9. doi: 10.1007/978-3-540-85289-6_3 (cit. on pp. 100, 102, 104).

[HHJS11] Denis Hatebur, Maritta Heisel, Jan Jürjens, and Holger Schmidt. "Systematic Development of UMLsec Design Models Based on Security Requirements". In: *Fundamental Approaches to Software Engineering*. 14th International Conference. FASE 2011 (Saarbrücken, Germany, Mar. 30–Apr. 1, 2011). Proceedings. Ed. by Dimitra Giannakopoulou and Fernando Orejas. Lecture Notes in Computer Science 6603: Advanced Research in Computing and Software Science. Springer, 2011, pp. 232–246. isbn: 978-3-642-19810-6. doi: 10.1007/978-3-642-19811-3_17 (cit. on p. 51).

[Hir04] Martin Hirsch. "Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit Uppaal". German. Diploma thesis. University of Paderborn, June 2004 (cit. on pp. 23, 25).

[HKB17] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. "Approaches to Co-Evolution of Metamodels and Models. A Survey". In: *IEEE Transactions on Software Engineering* 43.5 (May 2017), pp. 396–414. issn: 0098-5589. doi: 10.1109/TSE.2016.2610424 (cit. on pp. 140, 143).

[HLLL17] Abdulmalik Humayed, Jingqiang Lin, Fengjun Li, and Bo Luo. "Cyber-Physical Systems Security. A Survey". In: *IEEE Internet of Things Journal* 4.6 (Dec. 2017): *Special Issue on Security and Privacy in Cyber-Physical Systems*, pp. 1802–1831. issn: 2327-4662. doi: 10.1109/JIOT.2017.2703172 (cit. on p. 1).

[HLMN08] Charles B. Haley, Robin C. Laney, Jonathan D. Moffett, and Bashar Nuseibeh. "Security Requirements Engineering. A Framework for Representation and Analysis". In: *IEEE Transactions on Software Engineering* 34.1 (Jan.–Feb. 2008), pp. 133–153. issn: 0098-5589. doi: 10.1109/TSE.2007.70754 (cit. on p. 4).

[HMSS07] Dieter Hutter, Heiko Mantel, Ina Schaefer, and Axel Schairer. "Security of multi-agent systems. A case study on comparison shopping". In: *Journal of Applied Logic* 5.2 (June 2007): *Logic-Based Agent Verification*. Ed. by Michael Fisher, Munindar Singh, Diana Spears, and Mike Wooldridge, pp. 303–332. issn: 1570-8683. doi: 10.1016/j.jal.2005.12.015 (cit. on p. 30).

[Hol+16a] Jörg Holtmann, Ruslan Bernijazov, Matthias Meyer, David Schmelter, and Christian Tschirner. "Integrated and iterative systems engineering and software requirements engineering for technical systems". In: *Journal of Software. Evolution and Process* 28.9 (Sept. 2016): *Best Papers of International Conference on Software and Systems Process 2015*. Ed. by Dietmar Pfahl, Marco Kuhrmann, Reda Bendraou, and Richard Turner, pp. 722–743. issn: 2047-7473. doi: 10.1002/smr.1780 (cit. on pp. 14, 18).

[Hol+16b] Jörg Holtmann, Markus Fockel, Thorsten Koch, David Schmelter, Christian Brenner, Ruslan Bernijazov, and Marcel Sander. *The MechatronicUML Requirements Engineering Method. Process and Language*. Tech. rep. tr-ri-16-351. Software Engineering Department, Fraunhofer IEM and Software Engineering Group, Heinz Nixdorf Institute. Dec. 2016. doi: 10.13140/RG.2.2.33223.29606 (cit. on pp. 18, 147).

[Hol19] Jörg Holtmann. "Improvement of Software Requirements Quality based on Systems Engineering". PhD thesis. Paderborn University, June 2019. doi: 10.17619/UNIPB/1-730 (cit. on pp. 18, 62, 147).

[Hou+10] Siv Hilde Houmb, Shareeful Islam, Eric Knauss, Jan Jürjens, and Kurt Schneider. "Eliciting security requirements and tracing them to design. An integration of Common Criteria, heuristics, and UMLsec". In: *Requirements Engineering* 15.1 (Mar. 2010): *Special Issue on RE'09. Security Requirements Engineering*. Ed. by Eric Dubois and Haralambos Mouratidis, pp. 63–93. issn: 0947-36020. doi: 10.1007/s00766-009-0093-9 (cit. on p. 51).

[How78] William E. Howden. "Theoretical and Empirical Studies of Program Testing". In: *IEEE Transactions on Software Engineering* SE-4.4 (July 1978), pp. 293–298. issn: 0098-5589. doi: 10.1109/TSE.1978.231514 (cit. on p. 90).

[HP14] Sebastian J. I. Herzig and Christiaan J. J. Paredis. "Bayesian Reasoning Over Models". In: *Model-Driven Engineering, Verification and Validation*. MoDeVVa 2014 (Valencia, Spain, Sept. 30, 2014). Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation. Ed. by Frédéric Boulanger, Michalis Famelis, and Daniel Ratiu. CEUR Workshop Proceedings 1235. issn: 1613-0073. 2014, pp. 69–78. urn: urn:nbn:de:0074-1235-4 (cit. on p. 140).

[HP85] David Harel and Amir Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems* (La Colle-sur-Loup, France, Oct. 8–19, 1984). Conference proceedings. Ed. by Krzysztof R. Apt. NATO ASI Series F: Computer and System Sciences 13. Springer, 1985, pp. 477–498. isbn: 978-3-642-82455-5. doi: 10.1007/978-3-642-82453-1_17 (cit. on p. 2).

[HPLP09] Yong Huang, Lingdi Ping, Shanping Li, and Xuezeng Pan. "Analysis for Real-time Intransitive Information Flow Security Properties". In: *Journal of Software* 4.1 (Feb. 2009), pp. 90–97. issn: 1796-217X. doi: 10.4304/jsw.4.1.90-97 (cit. on p. 110).

[HR04] David Harel and Bernhard Rumpe. "Meaningful Modeling. What's the Semantics of "Semantics"?" In: *Computer. Innovative Technology for Computer Professionals* 37.10 (Oct. 2004), pp. 64–72. issn: 0018-9162. doi: 10.1109/MC.2004.172 (cit. on p. 12).

[HS05] Daniel Hedin and David Sands. “Timing Aware Information Flow Security for a JavaCard-like Bytecode”. In: *Electronic Notes in Theoretical Computer Science* 141.1 (Dec. 2005): *Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation. Bytecode 2005*. Ed. by Fausto Spoto, pp. 163–182. issn: 1571-0661. doi: 10.1016/j.entcs.2005.02.031 (cit. on p. 110).

[HS12] Daniel Hedin and Andrei Sabelfeld. “A Perspective on Information-Flow Control”. In: *Software Safety and Security. Tools for Analysis and Verification*. Summer School (Marktoberdorf, Aug. 2–14, 2011). Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. NATO Science for Peace and Security Series D: Information and Communication Security 33. IOS Press, 2012, pp. 319–347. isbn: 978-1-61499-027-7. doi: 10.3233/978-1-61499-028-4-319 (cit. on pp. 2, 28).

[HSJ09] Thomas Heyman, Riccardo Scandariato, and Wouter Joosen. “Risk-Driven Architectural Decomposition”. In: *International Conference on Availability, Reliability and Security*. ARES 2009 (Fukuoka, Japan, Mar. 16–19, 2009). Proceedings. IEEE Computer Society, 2009, pp. 363–368. isbn: 978-1-4244-3572-2. doi: 10.1109/ARES.2009.32 (cit. on p. 81).

[HSS14] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. “Modeling and enforcing secure object flows in process-driven SOAs. An integrated model-driven approach”. In: *Software and Systems Modeling* 13.2 (May 2014), pp. 513–548. issn: 1619-1366. doi: 10.1007/s10270-012-0263-y (cit. on p. 50).

[HSST13] Christian Heinzemann, Oliver Sudmann, Wilhelm Schäfer, and Matthias Tichy. “A discipline-spanning development process for self-adaptive mechatronic systems”. In: *2013 International Conference on Software and Systems Process*. ICSSP (San Francisco, California, USA, May 18–19, 2013). Proceedings. Ed. by Jürgen Münch, Jo Ann Lan, and He Zhang. ACM, 2013, pp. 36–45. isbn: 978-1-4503-2062-7. doi: 10.1145/2486046.2486055 (cit. on pp. 18, 19).

[HT97] Geňa Hahn and Claude Tardif. “Graph homomorphisms. Structure and symmetry”. In: *Graph Symmetry. Algebraic Methods and Applications* (Montreal, Canada, July 1–12, 1996). Proceedings of the NATO Advanced Study Institute and Séminaire de mathématiques supérieures. Ed. by Geňa Hahn and Gert Sabidussi. NATO ASI Series C: Mathematical and Physical Sciences 497. Kluwer, 1997, pp. 107–166. isbn: 978-90-481-4885-1. doi: 10.1007/978-94-015-8937-6_4 (cit. on p. 40).

[HWS06] Marieke Huisman, Pratik Worah, and Kim Sunesen. “A Temporal Logic Characterisation of Observational Determinism”. In: *19th IEEE Computer Security Foundations Workshop*. CSFW 2006 (Venice, Italy, July 5–7, 2006). Proceedings. IEEE Computer Society, 2006, pp. 3–15. isbn: 0-7695-2615-2. doi: 10.1109/CSFW.2006.6 (cit. on pp. 86, 109).

[HZJ19] Hsi-Ming Ho, Ruoyu Zhou, and Timothy M. Jones. “On Verifying Timed Hyperproperties”. In: *26th International Symposium on Temporal Representation and Reasoning*. TIME 2019 (Málaga, Spain, Oct. 16–19, 2019). Ed. by Johann Gamper, Sophie Pinchinat, and Guido Sciavicco. Leibniz International Proceedings in Informatics 147. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, Art. No. 20, pp. 1–18. isbn: 978-3-95977-127-6. doi: 10.4230/LIPIcs.TIME.2019.20 (cit. on pp. 86, 111).

[IV11] Petko Ivanov and Konrad Voigt. “Schema, Ontology and Metamodel Matching. Different, But Indeed the Same?” In: *Model and Data Engineering*. First International Conference. MEDI 2011 (Óbidos, Portugal, Sept. 28–30, 2011). Proceedings. Ed. by Ladjel Bellatreche and Filipe Mota Pinto. Lecture Notes in Computer Science 6918. Springer, 2011, pp. 18–30. isbn: 978-3-642-24442-1. doi: 10.1007/978-3-642-24443-8_5 (cit. on p. 139).

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. “ATL. A model transformation tool”. In: *Science of Computer Programming* 72.1–2 (June 2008): *Second issue of experimental software and toolkits. EST*. Ed. by Mark G. J. van den Brand, pp. 31–39. issn: 0167-6423. doi: 10.1016/j.scico.2007.08.002 (cit. on pp. 114, 134).

[Jac89] Jeremy Jacob. “On the Derivation of Secure Components”. In: *1989 IEEE Computer Society Symposium on Security and Privacy* (Oakland, California, USA, May 1–3, 1989). Proceedings. IEEE Computer Society, 1989, pp. 242–247. isbn: 0-8186-1939-2. doi: 10.1109/SECPRI.1989.36298 (cit. on pp. 106, 150).

[Jam+18] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. “Microservices. The Journey So Far and Challenges Ahead”. Guest Editors’ Introduction. In: *IEEE Software* 35.3 (May–June 2018): *Microservices*. Ed. by Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov, pp. 24–35. issn: 0740-7459. doi: 10.1109/MS.2018.2141039 (cit. on p. 55).

[JATM13] Mathieu Jaume, Radoniaina Andriatsimandefitra, Valérie Viet Triem Tong, and Ludovic Mé. “Secure States *versus* Secure Executions. From Access Control to Flow Control”. In: *Information Systems Security*. 9th International Conference. ICISS 2013 (Kolkata, India, Dec. 16–20, 2013). Proceedings. Ed. by Aditya Bagchi and Indrakshi Ray. Lecture Notes in Computer Science 8303. Springer, 2013, pp. 148–162. isbn: 978-3-642-45203-1. doi: 10.1007/978-3-642-45204-8_11 (cit. on p. 111).

[Jau12] Mathieu Jaume. “Semantic Comparison of Security Policies. From Access Control Policies to Flow Properties”. In: *IEEE CS Security and Privacy Workshops*. SPW 2012 (San Francisco, California, USA, May 24–25, 2012). Proceedings. IEEE Computer Society, 2012, pp. 60–67. isbn: 978-1-4673-2157-0. doi: 10.1109/SPW.2012.33 (cit. on pp. 2, 29).

[JJ11] Jostein Jensen and Martin Gilje Jaatun. “Not Ready for Prime Time. A Survey on Security in Model Driven Development”. In: *International Journal of Secure Software Engineering* 2.4 (Oct.–Dec. 2011), pp. 49–61. issn: 1947-3036. doi: 10.4018/jsse.2011100104 (cit. on p. 4).

[JK11] Jason Jaskolka and Ridha Khédri. “Exploring Covert Channels”. In: *Proceedings of the 44th Annual Hawaii International Conference on System Sciences*. HICSS-44 (Kōloa, Hawaii, USA, Jan. 4–7, 2011). Ed. by Ralph H. Sprague. IEEE Computer Society, 2011, pp. 1–10. isbn: 978-0-7695-4282-9. doi: 10.1109/HICSS.2011.201 (cit. on p. 28).

[JKZ12] Jason Jaskolka, Ridha Khédri, and Qinglei Zhang. “On the Necessary Conditions for Covert Channel Existence. A State-of-the-Art Survey”. In: *Procedia Computer Science* 10 (2012): *ANT 2012 and MobiWIS 2012*. Ed. by Elhadi M. Shakshuki and Muhammad Younas, pp. 458–465. issn: 1877-0509. doi: 10.1016/j.procs.2012.06.059 (cit. on p. 28).

[JLS00] Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. “Scaling up Uppaal. Automatic Verification of Real-Time Systems using Compositionality and Abstraction”. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. 6th International Symposium. FTRTFT 2000 (Pune, India, Sept. 20–22, 2000). Proceedings. Ed. by Mathai Joseph. Lecture Notes in Computer Science 1926. Springer, 2000, pp. 19–30. isbn: 3-540-41055-4. doi: 10.1007/3-540-45352-0_4 (cit. on pp. 24, 102, 105).

[Joh+19] Stefan John, Alexandru Burdusel, Robert Bill, Daniel Strüber, Gabriele Taentzer, Steffen Zschaler, and Manuel Wimmer. “Searching for Optimal Models. Comparing Two Encoding Approaches”. In: *Journal of Object Technology* 18.3, Art. No. 6 (July 2019): *The 12th International Conference on Model Transformations*. Ed. by Anthony Anjorin and Regina Hebig, pp. 1–22. issn: 1660-1769. doi: 10.5381/jot.2019.18.3.a6 (cit. on p. 142).

[Joh97] Ralph E. Johnson. “Frameworks = (Components + Patterns)”. In: *Communications of the ACM* 40.10 (Oct. 1997): *Object-Oriented Application Frameworks*. Ed. by Diane Crawford, pp. 39–42. issn: 0001-0782. doi: 10.1145/262793.262799 (cit. on pp. 114, 127).

[Jür+19] Jan Jürjens et al. “Maintaining Security in Software Evolution”. In: *Managed Software Evolution*. Ed. by Ralf H. Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin. Springer, 2019. Chap. 9, pp. 207–253. isbn: 978-3-030-13498-3. doi: 10.1007/978-3-030-13499-0_9 (cit. on p. 148).

[Jür05] Jan Jürjens. *Secure systems development with UML*. Springer, 2005. isbn: 978-3-540-00701-2. doi: 10.1007/b137706 (cit. on pp. 34, 51).

[JV17] Jason Jaskolka and John Villasenor. “An Approach for Identifying and Analyzing Implicit Interactions in Distributed Systems”. In: *IEEE Transactions on Reliability* 66.2 (June 2017), pp. 529–546. issn: 0018-9529. doi: 10.1109/TR.2017.2665164 (cit. on p. 81).

[Kah+19] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Jürgen Dingel, and Dániel Varró. “Survey and classification of model transformation tools”. In: *Software and Systems Modeling* 18.4 (Aug. 2019), pp. 2361–2397. issn: 1619-1366. doi: 10.1007/s10270-018-0665-6 (cit. on pp. 6, 13).

[KAH17] Timo Kehrer, Abdullah M. Alshanqiti, and Reiko Heckel. “Automatic Inference of Rule-Based Specifications of Complex In-place Model Transformations”. In: *Theory and Practice of Model Transformation*. 10th International Conference. ICMT 2017 (Marburg, Germany, July 17–18, 2017). Proceedings. Ed. by Esther Guerra and Mark G. J. van den Brand. Lecture Notes in Computer Science 10374. Springer, 2017, pp. 92–107. isbn: 978-3-319-61472-4. doi: 10.1007/978-3-319-61473-1_7 (cit. on p. 142).

[Kap+12] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. “Model Transformation By-Example. A Survey of the First Wave”. In: *Conceptual Modelling and Its Theoretical Foundations. Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday*. Ed. by Antje Düsterhöft, Meike Klettke, and Klaus-Dieter Schewe. Lecture Notes in Computer Science 7260: Festschrift. Springer, 2012, pp. 197–215. isbn: 978-3-642-28278-2. doi: 10.1007/978-3-642-28279-9_15 (cit. on p. 142).

[KB06] Boris Köpf and David A. Basin. “Timing-Sensitive Information Flow Analysis for Synchronous Systems”. In: *Computer Security. ESORICS 2006*. 11th European Symposium on Research in Computer Security (Hamburg, Germany, Sept. 18–20, 2006). Proceedings. Ed. by Dieter Gollmann, Jan Meier, and Andrei Sabelfeld. Lecture Notes in Computer Science 4189. Springer, 2006, pp. 243–262. isbn: 3-540-44601-X. doi: 10.1007/11863908_16 (cit. on p. 110).

[KBD16] Lydia Kaiser, Christian Bremer, and Roman Dumitrescu. “Exhaustiveness of Systems Structures in Model-based Systems Engineering for Mechatronic Systems”. In: *Procedia Technology* 26 (2016): *3rd International Conference on System-Integrated Intelligence. New Challenges for Product and Production Engineering*. Ed. by Ansgar Trächtler, Berend Denkena, and Klaus-Dieter Thoben, pp. 428–435. issn: 2212-0173. doi: 10.1016/j.protcy.2016.08.055 (cit. on p. 15).

[KBM16] Tomaž Kosar, Sudev Bohrab, and Marjan Mernik. “Domain-Specific Languages. A Systematic Mapping Study”. In: *Information and Software Technology* 71 (Mar. 2016), pp. 77–91. issn: 0950-5849. doi: 10.1016/j.infsof.2015.11.001 (cit. on pp. 1, 10).

[KBN14] Hasan Kahtan, Nordin Abu Bakar, and Rosmawati Nordin. “Dependability Attributes for increased Security in Component-based Software Development”. In: *Journal of Computer Science* 10.7 (2014), pp. 1298–1306. issn: 1549-3636. doi: 10.3844/jcssp.2014.1298.1306 (cit. on p. 80).

[KDHM13] Lydia Kaiser, Roman Dumitrescu, Jörg Holtmann, and Matthias Meyer. “Automatic Verification of Modeling Rules in Systems Engineering for Mechatronic Systems”. In: *ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. Vol. 2.B: *33rd Computers and Information in Engineering Conference* (Portland, Oregon, USA, Aug. 4–7, 2013). Conference Proceedings. ASME, 2013, Art. No. DETC2013-12330. isbn: 978-0-7918-5586-7. doi: 10.1115/DETC2013-12330 (cit. on p. 116).

[Kes+14] Marouane Kessentini, Ali Ouni, Philip Langer, Manuel Wimmer, and Slim Bechikh. “Search-based metamodel matching with structural and syntactic measures”. In: *Journal of Systems and Software* 97 (Nov. 2014), pp. 1–14. issn: 0164-1212. doi: 10.1016/j.jss.2014.06.040 (cit. on p. 144).

[KG17] Madhavi Karanam and Lavanya Gottemukkala. “Model Transformation Languages. State-of-the-art”. In: *International Journal on Computer Science and Engineering* 9.6 (June 2017), pp. 404–408. issn: 2229-5631. url: http://www.enggjournals.com/ijcse/doc/IJCSE17-09-06-014.pdf (cit. on pp. 6, 13, 113, 134).

[KGBH10] Lucia Kapová, Thomas Goldschmidt, Steffen Becker, and Jörg Henß. “Evaluating Maintainability with Code Metrics for Model-to-Model Transformations”. In: *Research into Practice. Reality and Gaps*. 6th International Conference on the Quality of Software Architectures. QoSA 2010 (Prague, Czech Republic, June 23–25, 2010). Proceedings. Ed. by George T. Heineman, Jan Kofron, and Frantisek Plasil. Lecture Notes in Computer Science 6093. Springer, 2010, pp. 151–166. isbn: 978-3-642-13820-1. doi: 10.1007/978-3-642-13821-8_12 (cit. on p. 118).

[KHHJ08] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. “Implicit Flows. Can’t Live with ’Em, Can’t Live without ’Em”. In: *Information Systems Security*. 4th International Conference. ICISS 2008 (Hyderabad, India, Dec. 16–20, 2008). Proceedings. Ed. by R. Sekar and Arun K. Pujari. Lecture Notes in Computer Science 5352. Springer, 2008, pp. 56–70. isbn: 978-3-540-89861-0. doi: 10.1007/978-3-540-89862-7_4 (cit. on p. 28).

[KHN11] Kresimir Kasal, Johannes Heurix, and Thomas Neubauer. “Model-Driven Development Meets Security. An Evaluation of Current Approaches”. In: *Proceedings of the 44th Annual Hawaii International Conference on System Sciences*. HICSS-44 (Kōloa, Hawaii, USA, Jan. 4–7, 2011). Ed. by Ralph H. Sprague. IEEE Computer Society, 2011, pp. 1–9. isbn: 978-0-7695-4282-9. doi: 10.1109/HICSS.2011.310 (cit. on p. 4).

[KI16] Naurin Farooq Khan and Naveed Ikram. “Security Requirements Engineering. A Systematic Mapping”. 2010–2015. In: *2016 International Conference on Software Security and Assurance*. ICSSA 2016 (St. Pölten, Austria, Aug. 24–25, 2016). Proceedings. IEEE Computer Society, 2016, pp. 31–36. isbn: 978-1-5090-4388-0. doi: 10.1109/ICSSA.2016.13 (cit. on p. 4).

[KKE18] Djamel Eddine Khelladi, Roland Kretschmer, and Alexander Egyed. “Change Propagation-based and Composition-based Co-evolution of Transformations with Evolving Metamodels”. In: *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS’18 (Copenhagen, Denmark, Oct. 14–19, 2018). Proceedings. Ed. by Andrzej Wąsowski, Richard F. Paige, and Øystein Haugen. ACM, 2018, pp. 404–414. isbn: 978-1-4503-4949-9. doi: 10.1145/3239372.3239380 (cit. on p. 143).

[Kle08] Anneke Kleppe. *Software Language Engineering. Creating Domain-Specific Languages using Metamodels*. With a forew. by Jean-Marie Favre. Addison-Wesley, 2008. isbn: 978-0-321-55345-4 (cit. on p. 10).

[KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. “Model Checking Timed UML State Machines and Collaborations”. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. 7th International Symposium. FTRTFT 2002 (Oldenburg, Germany, Sept. 9–12, 2002). Proceedings. Ed. by Werner Damm and Ernst-Rüdiger Olderog. Lecture Notes in Computer Science 2469. Springer, 2002, pp. 395–416. isbn: 3-540-44165-4. doi: 10.1007/3-540-45739-9_23 (cit. on p. 74).

[Köh+19] Maximilian A. Köhl, Kevin Baum, Markus Langer, Daniel Oster, Timo Speith, and Dimitri Bohlender. “Explainability as a Non-Functional Requirement”. In: *27th IEEE International Requirements Engineering Conference*. RE 2019 (Jeju, South Korea, Sept. 23–27, 2019). Proceedings. Ed. by Daniela E. Damian, Anna Perini, and Seok-Won Lee. IEEE, 2019, pp. 363–368. isbn: 978-1-7281-3912-8. doi: 10.1109/RE.2019.00046 (cit. on p. 140).

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. “The Epsilon Transformation Language”. In: *Theory and Practice of Model Transformations*. First International Conference. ICMT 2008 (Zürich, Switzerland, July 1–2, 2008). Proceedings. Ed. by Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio. Lecture Notes in Computer Science 5063. Springer, 2008, pp. 46–60. isbn: 978-3-540-69926-2. doi: 10.1007/978-3-540-69927-9_4 (cit. on p. 134).

[Kra07] Jeff Kramer. “Is abstraction the key to computing?” In: *Communications of the ACM* 50.4 (Apr. 2007), pp. 36–42. issn: 0001-0782. doi: 10.1145/1232743.1232745 (cit. on p. 1).

[KS17] Rajesh Kumar and Mariëlle Stoelinga. “Quantitative Security and Safety Analysis with Attack-Fault Trees”. In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering*. HASE 2017 (Singapore, Jan. 12–14, 2017). Proceedings. IEEE Computer Society, 2017, pp. 25–32. isbn: 978-1-5090-4636-2. doi: 10.1109/HASE.2017.12 (cit. on p. 107).

[KSBB12] Marouane Kessentini, Houari A. Sahraoui, Mounir Boukadoum, and Omar Benomar. "Search-based model transformation by example". In: *Software and Systems Modeling* 11.2 (May 2012), pp. 209–226. issn: 1619-1366. doi: 10.1007/s10270-010-0175-7 (cit. on pp. 142, 148).

[KSBR15] Kuzman Katkalov, Kurt Stenzel, Marian Borek, and Wolfgang Reif. "Modeling information flow properties with UML". In: *2015 7th International Conference on New Technologies, Mobility and Security*. NTMS 2015 (Paris, France, July 27–29, 2015). Proceedings of NTMS'2015 Conference and Workshops. Ed. by Mohamad Badra, Azzedine Boukerche, and Pascal Urien. IEEE, 2015, pp. 1–5. isbn: 978-1-4799-8784-9. doi: 10.1109/NTMS.2015.7266507 (cit. on pp. 34, 51).

[KSNW19] Narges Khakpour, Charilaos Skandylas, Goran Saman Nariman, and Danny Weyns. "Towards secure architecture-based adaptations". In: *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2019 (Montreal, Canada, May 25–26, 2019). Proceedings. Ed. by Marin Litoiu, Siobhán Clarke, and Kenji Tei. ACM, 2019, pp. 114–125. isbn: 978-1-7281-3368-3. doi: 10.1109/SEAMS.2019.00023 (cit. on pp. 79, 147).

[KSW18] Wael Kessentini, Houari A. Sahraoui, and Manuel Wimmer. "Automated Co-evolution of Metamodels and Transformation Rules. A Search-Based Approach". In: *Search-Based Software Engineering*. 10th International Symposium. SSBSE 2018 (Montpellier, France, Sept. 8–9, 2018). Proceedings. Ed. by Thelma Elita Colanzi and Phil McMinn. Lecture Notes in Computer Science 11036. Springer, 2018, pp. 229–245. isbn: 978-3-319-99240-2. doi: 10.1007/978-3-319-99241-9_12 (cit. on pp. 143, 144).

[KSW19] Wael Kessentini, Houari A. Sahraoui, and Manuel Wimmer. "Automated metamodel/model co-evolution. A search-based approach". In: *Information and Software Technology* 106 (Feb. 2019), pp. 49–67. issn: 0950-5849. doi: 10.1016/j.infsof.2018.09.003 (cit. on pp. 143, 144).

[KTRK16] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. "Automatically Deriving the Specification of Model Editing Operations from Meta-Models". In: *Theory and Practice of Model Transformations*. 9th International Conference. ICMT 2016 (Vienna, Austria, July 4–5, 2016). Proceedings. Ed. by Pieter Van Gorp and Gregor Engels. Lecture Notes in Computer Science 9765. Springer, 2016, pp. 173–188. isbn: 978-3-319-42063-9. doi: 10.1007/978-3-319-42064-6_12 (cit. on p. 143).

[Küh06] Thomas Kühne. "Matters of (Meta-)Modeling". In: *Software and Systems Modeling* 5.4 (Dec. 2006), pp. 369–385. issn: 1619-1366. doi: 10.1007/s10270-006-0017-9 (cit. on p. 10).

[Kur10] Ivan Kurtev. “Application of reflection in a model transformation language”. In: *Software and Systems Modeling* 9.3 (June 2010), pp. 311–333. issn: 1619-1366. doi: 10.1007/s10270-009-0138-z (cit. on p. 133).

[Kus+13] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. “Reality Check for Model Transformation Reuse. The ATL Transformation Zoo Case Study”. In: *Analysis of Model Transformations*. AMT 2013 (Miami, Florida, USA, Sept. 29, 2013). Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013). Ed. by Benoit Baudry, Jürgen Dingel, Levi Lúcio, and Hans Vangheluwe. CEUR Workshop Proceedings 1077. issn: 1613-0073. 2013. urn: urn:nbn:de:0074-1077-8 (cit. on p. 139).

[Kus+15] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. “Reuse in model-to-model transformation languages. Are we there yet?” In: *Software and Systems Modeling* 14.2 (May 2015), pp. 537–572. issn: 1619-1366. doi: 10.1007/s10270-013-0343-7 (cit. on pp. 118, 141).

[KW12] Derrick G. Kourie and Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer, 2012. isbn: 978-3-642-27918-8. doi: 10.1007/978-3-642-27919-5 (cit. on p. 2).

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained. The Model Driven Architecture*. Practice and promise. With a forew. by Andrew Watson. Addison-Wesley Object Technology Series. 2003. isbn: 0-321-19442-X (cit. on pp. 6, 9, 12, 13).

[KWH11] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. “Timing- and Termination-Sensitive Secure Information Flow. Exploring a New Approach”. In: *2011 IEEE Symposium on Security and Privacy*. SP 2011 (Berkeley, California, USA, May 22–25, 2011). Proceedings. IEEE Computer Society, 2011, pp. 413–428. isbn: 978-1-4577-0147-4. doi: 10.1109/SP.2011.19 (cit. on pp. 85, 87, 110, 146).

[Lam73] Butler W. Lampson. “A Note on the Confinement Problem”. In: *Communications of the ACM* 16.10 (Oct. 1973), pp. 613–615. issn: 0001-0782. doi: 10.1145/362375.362389 (cit. on p. 28).

[Lam77] Leslie Lamport. “Proving the Correctness of Multiprocess Programs”. In: *IEEE Transactions on Software Engineering* SE-3.2 (Mar. 1977), pp. 125–143. issn: 0098-5589. doi: 10.1109/TSE.1977.229904 (cit. on pp. 19, 32).

[Lau14] Kung-Kiu Lau. “Software component models. Past, present and future”. In: *Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE’14 (Marcq-en-Barœul, France, June 30–July 4, 2014). Part of CompArch 2014. Ed. by Lionel Seinturier, Eduardo Santana de Almeida, and Jan Carlson. ACM, 2014, pp. 185–186. isbn: 978-1-4503-2577-6. doi: 10.1145/2602458.2611456 (cit. on p. 18).

[Lau17] Kung-Kiu Lau. "From Formal Methods to Software Components. Back to the Future?" In: *Formal Aspects of Component Software*. 13th International Conference. FACS 2016 (Besançon, France, Oct. 19–21, 2016). Revised Selected Papers. Ed. by Olga Kouchnarenko and Ramtin Khosravi. Lecture Notes in Computer Science 10231. 2017, pp. 10–14. isbn: 978-3-319-57665-7. doi: 10.1007/978-3-319-57666-4_2 (cit. on p. 56).

[Lee09] Edward A. Lee. "Computing needs time". In: *Communications of the ACM* 52.5 (May 2009): *Security in the Browser*, pp. 70–79. issn: 0001-0782. doi: 10.1145/1506409.1506426 (cit. on p. 22).

[Lee10] Edward A. Lee. "CPS foundations". In: *Proceedings of the 47th Design Automation Conference*. DAC 2010 (Anaheim, California, USA, July 13–18, 2010). Ed. by Sachin S. Sapatnekar. ACM, 2010, pp. 737–742. isbn: 978-1-4503-0002-5. doi: 10.1145/1837274.1837462 (cit. on p. 1).

[Lee18] Edward A. Lee. "What Is Real Time Computing? A Personal View". In: *IEEE Design & Test* 35.2 (Mar.–Apr. 2018): *Special Issue on Time-Critical Systems Design*, pp. 64–72. issn: 2168-2356. doi: 10.1109/MDAT.2017.2766560 (cit. on p. 5).

[Lem+13] Rogério de Lemos et al. "Software Engineering for Self-Adaptive Systems. A Second Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Vol. II. International Seminar (Dagstuhl Castle, Germany, Oct. 24–29, 2010). Revised Selected and Invited Papers. Ed. by Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw. Lecture Notes in Computer Science 7475: State-of-the-Art Survey. Springer, 2013, pp. 1–32. isbn: 978-3-642-35812-8. doi: 10.1007/978-3-642-35813-5_1 (cit. on p. 14).

[LFH14] Lamine Lafi, Jamel Feki, and Slimane Hammoudi. "Metamodel Matching Techniques. Review, Comparison and Evaluation". In: *International Journal of Information System Modeling and Design* 5.2 (Apr.–June 2014), pp. 70–94. issn: 1947-8186. doi: 10.4018/ijismd.2014040104 (cit. on pp. 114, 143, 144).

[LHBJ06] Denivaldo Lopes, Slimane Hammoudi, Jean Bézivin, and Frédéric Jouault. "Mapping Specification in MDA. From Theory to Practice". In: *Interoperability of Enterprise Software and Applications*. First International Conference. INTEROP-ESA 2005 (Geneva, Switzerland, Feb. 21–25, 2005). Ed. by Dimitri Konstantas, Jean-Paul Bourrières, Michel Léonard, and Nacer Boudjlida. Springer, 2006, pp. 253–264. isbn: 978-1-84628-152-5. doi: 10.1007/1-84628-152-0_23 (cit. on pp. 117, 141).

[Lie+18] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. "Model-based engineering in the embedded systems domain. An industrial survey on the state-of-practice". In: *Software and Systems Modeling* 17.1 (Feb. 2018), pp. 91–113. issn: 1619-1366. doi: 10.1007/s10270-016-0523-3 (cit. on p. 1).

[LK14] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. "Model-Transformation Design Patterns". In: *IEEE Transactions on Software Engineering* 40.12 (Dec. 2014), pp. 1224–1259. issn: 0098-5589. doi: 10.1109/TSE.2014.2354344 (cit. on pp. 115, 127, 131).

[LLN18] Kim Guldstrand Larsen, Florian Lorber, and Brian Nielsen. "20 Years of Uppaal Enabled Industrial Model-Based Validation and Beyond". In: *Leveraging Applications of Formal Methods, Verification and Validation*. Proceedings. Part IV: *Industrial Practice*. 8th International Symposium. ISoLA 2018 (Limassol, Cyprus, Nov. 5–9, 2018). Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science 11247. Springer, 2018, pp. 212–229. isbn: 978-3-030-03426-9. doi: 10.1007/978-3-030-03427-6_18 (cit. on pp. 6, 86, 90, 107).

[LMT02] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Simone Tini. "Privacy in Real-Time Systems". In: *Electronic Notes in Theoretical Computer Science* 52.3 (May 2002): *MTCS 2001. Models for Time-Critical Systems*. Satellite Event of CONCUR 2001. Ed. by Flavio Corradini and Walter Vogler, pp. 295–305. issn: 1571-0661. doi: 10.1016/S1571-0661(04)00229-4 (cit. on p. 111).

[LMT04] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Simone Tini. "Information flow in hybrid systems". In: *ACM Transactions on Embedded Computing Systems* 3.4 (Nov. 2004), pp. 760–799. issn: 1539-9087. doi: 10.1145/1027794.1027799 (cit. on p. 78).

[LMT10] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Angelo Troina. "Time and Probability-Based Information Flow Analysis". In: *IEEE Transactions on Software Engineering* 36.5 (Sept.–Oct. 2010), pp. 719–734. issn: 0098-5589. doi: 10.1109/TSE.2010.4 (cit. on pp. 86, 111).

[LMT17] Ximeng Li, Heiko Mantel, and Markus Tasch. "Taming Message-Passing Communication in Compositional Reasoning About Confidentiality". In: *Programming Languages and Systems*. 15th Asian Symposium. APLAS 2017 (Suzhou, China, Nov. 27–29, 2017). Proceedings. Ed. by Bor-Yuh Evan Chang. Lecture Notes in Computer Science 10695. Springer, 2017, pp. 45–66. isbn: 978-3-319-71236-9. doi: 10.1007/978-3-319-71237-6_3 (cit. on pp. 56, 82).

[LS11] Thomas Leveque and Séverine Sentilles. “Refining extra-functional property values in hierarchical component models”. In: *Proceedings of the 2011 Federated Events on Component-Based Software Engineering & Software Architecture. CompArch'11*. 14th International ACM Sigsoft Symposium on Component Based Software Engineering. CBSE 2011 (Boulder, Colorado, USA, June 20–24, 2011). Ed. by Ivica Crnković, Judith A. Stafford, Antonia Bertolino, and Kendra M. L. Cooper. ACM, 2011, pp. 83–92. ISBN: 978-1-4503-0723-9. DOI: 10.1145/2000229.2000242 (cit. on p. 55).

[Lúc+14] Levi Lúcio, Qin Zhang, Phu Hong Nguyen, Moussa Amrani, Jacques Klein, Hans Vangheluwe, and Yves Le Traon. “Advances in Model-Driven Security”. In: *Advances in Computers*. Vol. 93. Ed. by Atif Memon. Elsevier, 2014. Chap. 3, pp. 103–152. ISBN: 978-0-12-800162-2. DOI: 10.1016/B978-0-12-800162-2.00003-8 (cit. on p. 4).

[Lúc+16] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. “Model transformation intents and their properties”. In: *Software and Systems Modeling* 15.3 (July 2016), pp. 647–684. ISSN: 1619-1366. DOI: 10.1007/s10270-014-0429-x (cit. on pp. 13, 140).

[Lun+19] Yuriy Zacchia Lun, Alessandro D'Innocenzo, Francesco Smarra, Ivano Malavolta, and Maria Domenica Di Benedetto. “State of the art of cyber-physical systems security. An automatic control perspective”. In: *Journal of Systems and Software* 149 (Mar. 2019), pp. 174–216. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.12.006 (cit. on p. 1).

[LVDN17] Laurens Lemaire, Jan Vossaert, Bart De Decker, and Vincent Naessens. “Extending FAST-CPS for the Analysis of Data Flows in Cyber-Physical Systems”. In: *Computer Network Security*. 7th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. MMM-ACNS 2017 (Warsaw, Poland, Aug. 28–30, 2017). Proceedings. Ed. by Jacek Rak, John Bay, Igor V. Kotenko, Leonard J. Popyack, Victor A. Skormin, and Krzysztof Szczypiorski. Lecture Notes in Computer Science 10446. Springer, 2017, pp. 37–49. ISBN: 978-3-319-65126-2. DOI: 10.1007/978-3-319-65127-9_4 (cit. on p. 52).

[LVJN17] Laurens Lemaire, Jan Vossaert, Joachim Jansen, and Vincent Naessens. “A logic-based framework for the security analysis of Industrial Control Systems”. In: *Automatic Control and Computer Sciences* 51.2 (Mar. 2017), pp. 114–123. ISSN: 0146-4116. DOI: 10.3103/S0146411617020055 (cit. on p. 52).

[LW07] Kung-Kiu Lau and Zheng Wang. “Software Component Models”. In: *IEEE Transactions on Software Engineering* 33.10 (Oct. 2007), pp. 709–724. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70726 (cit. on pp. 18, 20).

[LW94] Barbara Liskov and Jeannette M. Wing. “A Behavioral Notion of Subtyping”. In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383 (cit. on pp. 12, 123).

[MA11] Mubarak Mohammad and Vangalur S. Alagar. “A formal approach for the specification and verification of trustworthy component-based systems”. In: *Journal of Systems and Software* 84.1 (Jan. 2011): *Information Networking and Software Services*. Ed. by Irfan Awan, Muhammad Younas, and Makoto Takizawa, pp. 77–104. ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.08.048 (cit. on pp. 56, 80).

[Man00a] Heiko Mantel. “Possibilistic Definitions of Security. An Assembly Kit”. In: *13th IEEE Computer Security Foundations Workshop*. CSFW-13 (Cambridge, UK, July 3–5, 2000). Proceedings. IEEE Computer Society, 2000, pp. 185–199. ISBN: 0-7695-0671-2. DOI: 10.1109/CSFW.2000.856936 (cit. on pp. 2, 29, 49, 91).

[Man00b] Heiko Mantel. “Unwinding Possibilistic Security Properties”. In: *Computer Security. ESORICS 2000*. 6th European Symposium on Research in Computer Security (Toulouse, France, Oct. 4–6, 2000). Proceedings. Ed. by Frédéric Cuppens, Yves Deswarte, Dieter Gollmann, and Michael Waidner. Lecture Notes in Computer Science 1895. Springer, 2000, pp. 238–254. ISBN: 3-540-41031-7. DOI: 10.1007/10722599_15 (cit. on pp. 86, 108).

[Man02] Heiko Mantel. “On the Composition of Secure Systems”. In: *2002 IEEE Symposium on Security and Privacy* (Berkeley, California, USA, May 12–15, 2002). Proceedings. IEEE Computer Society, 2002, pp. 88–101. ISBN: 0-7695-1543-6. DOI: 10.1109/SECPRI.2002.1004364 (cit. on pp. 5, 32, 49, 55, 56, 66, 71, 82, 146).

[Man03] Heiko Mantel. “A uniform framework for the formal specification and verification of information flow security”. PhD thesis. Saarbrücken: Saarland University, 2003. DOI: 10.22028/D291-25715 (cit. on pp. 2, 7, 29–31, 60, 62, 71, 72, 79, 145, 150).

[Man11] Heiko Mantel. *Information Flow and Noninterference*. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. 2nd ed. Springer, 2011, pp. 605–607. ISBN: 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5_874 (cit. on pp. 2, 4, 28, 145).

[MB19] Omar Masmali and Omar Badreddin. “Model Driven Security. A Systematic Mapping Study”. In: *Software Engineering* 7.2 (June 2019), pp. 30–38. ISSN: 2376-8029. DOI: 10.11648/j.se.20190702.12 (cit. on p. 4).

[MBSF10] Daniel Mellado, Carlos Blanco, Luís Enrique Sanchez, and Eduardo Fernández-Medina. “A systematic review of security requirements engineering”. In: *Computer Standards & Interfaces* 32.4 (June 2010), pp. 153–165. issn: 0920-5489. doi: 10.1016/j.csi.2010.01.006 (cit. on p. 4).

[McC87] Daryl McCullough. “Specifications for Multi-Level Security and a Hook-Up Property”. In: *1987 IEEE Symposium on Security and Privacy* (Oakland, California, USA, Apr. 27–29, 1987). IEEE Computer Society, 1987, pp. 161–166. isbn: 0-8186-0771-8. doi: 10.1109/SP.1987.10009 (cit. on pp. 60, 71).

[McC88] Daryl McCullough. “Noninterference and the composability of security properties”. In: *1988 IEEE Symposium on Security and Privacy* (Oakland, California, USA, Apr. 18–21, 1988). Proceedings. IEEE Computer Society, 1988, pp. 177–186. isbn: 0-8186-0850-1. doi: 10.1109/SECPRI.1988.8110 (cit. on p. 56).

[McL94] John McLean. “A general theory of composition for trace sets closed under selective interleaving functions”. In: *1994 IEEE Computer Society Symposium on Research in Security and Privacy* (Oakland, California, USA, May 16–18, 1994). Proceedings. IEEE Computer Society, 1994, pp. 79–93. isbn: 0-8186-5675-1. doi: 10.1109/RISP.1994.296590 (cit. on pp. 60, 91).

[McL96] John McLean. “A General Theory of Composition for a Class of “Possibilistic” Properties”. In: *IEEE Transactions on Software Engineering* 22.1 (Jan. 1996), pp. 53–67. issn: 0098-5589. doi: 10.1109/32.481534 (cit. on pp. 2, 29, 32, 49, 55, 56, 74).

[Mel+16] Niklas Mellegård, Adry Ferwerda, Kenneth Lind, Rogardt Heldal, and Michel R. V. Chaudron. “Impact of Introducing Domain-Specific Modelling in Software Maintenance. An Industrial Case Study”. In: *IEEE Transactions on Software Engineering* 42.3 (Mar. 2016), pp. 245–260. issn: 0098-5589. doi: 10.1109/TSE.2015.2479221 (cit. on p. 1).

[Mén+19] Daniel Méndez Fernández, Wolfgang Böhm, Andreas Vogelsang, Jakob Mund, Manfred Broy, Marco Kuhrmann, and Thorsten Weyer. “Artefacts in software engineering. A fundamental positioning”. In: *Software and Systems Modeling* 18.5 (Oct. 2019), pp. 2777–2786. issn: 1619-1366. doi: 10.1007/s10270-019-00714-3 (cit. on pp. 1, 9).

[Men13] Tom Mens. “Model Transformation. A Survey of the State of the Art”. In: *Model-Driven Engineering for Distributed Real-Time Systems. MARTE Modeling, Model Transformations and their Usages*. Ed. by Jean-Philippe Babau, Mireille Blay-Fornarino, Joël Champeau, Sylvain Robert, and Antonio Sabetta. Wiley, 2013. Chap. 1, pp. 1–19. isbn: 978-1-84821-115-5. doi: 10.1002/9781118558096.ch1 (cit. on pp. 6, 12).

[Met05] Andreas Metzger. "A Systematic Look at Model Transformations". In: *Model-Driven Software Development*. Ed. by Sami Beydeda, Matthias Book, and Volker Gruhn. Springer, 2005, pp. 19–33. isbn: 978-3-540-25613-7. doi: 10.1007/3-540-28554-7_2 (cit. on pp. 6, 12).

[Mey92] Bertrand Meyer. "Applying 'Design by Contract'". In: *Computer* 25.10 (Oct. 1992), pp. 40–51. issn: 0018-9162. doi: 10.1109/2.161279 (cit. on p. 19).

[MG06] Tom Mens and Pieter Van Gorp. "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (Mar. 2006): *Proceedings of the International Workshop on Graph and Model Transformation. GraMoT 2005*. Ed. by Gabor Karsai and Gabriele Taentzer, pp. 125–142. issn: 1571-0661. doi: 10.1016/j.entcs.2005.10.021 (cit. on pp. 6, 12, 13, 113).

[MG07] Haralambos Mouratidis and Paolo Giorgini. "Secure Tropos. A Security-Oriented Extension of the Tropos Methodology". In: *International Journal of Software Engineering and Knowledge Engineering* 17.2 (Apr. 2007), pp. 285–309. issn: 0218-1940. doi: 10.1142/S0218194007003240 (cit. on p. 51).

[MGM03] Haralambos Mouratidis, Paolo Giorgini, and Gordon A. Manson. "Integrating Security and Systems Engineering. Towards the Modelling of Secure Information Systems". In: *Advanced Information Systems Engineering*. 15th International Conference. CAiSE 2003 (Klagenfurt/Velden, Austria, June 16–18, 2003). Proceedings. Ed. by Johann Eder and Michele Missikoff. Lecture Notes in Computer Science 2681. Springer, 2003, pp. 63–78. isbn: 3-540-40442-2. doi: 10.1007/3-540-45017-3_7 (cit. on p. 51).

[Mic68] Donald Michie. ""Memo" Functions and Machine Learning". In: *Nature. International weekly journal of science* 218.5136 (Apr. 1968), pp. 19–22. issn: 0028-0836. doi: 10.1038/218019a0 (cit. on p. 129).

[Mil71] Robin Milner. "An Algebraic Definition of Simulation Between Programs". In: *Proceedings of the Second International Joint Conference on Artificial Intelligence*. IJCAI 1971 (London, UK, Sept. 1–3, 1971). Ed. by David C. Cooper. IJCAI, 1971, pp. 481–489. isbn: 0-934613-34-6. url: http://ijcai.org/Proceedings/71/Papers/044.pdf (cit. on pp. 25, 71).

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer, 1980. isbn: 3-540-10235-3. doi: 10.1007/3-540-10235-3 (cit. on p. 24).

[MJ10] Haralambos Mouratidis and Jan Jürjens. "From goal-driven security requirements engineering to secure design". In: *International Journal of Intelligent Systems* 25.8 (Aug. 2010): *Goal-driven Requirements Engineering*, pp. 813–840. issn: 1098-111X. doi: 10.1002/int.20432 (cit. on p. 51).

[MNAM17] Nabil M. Mohammed, Mahmood Niazi, Mohammad Alshayeb, and Sajjad Mahmood. “Exploring software security approaches in software development lifecycle. A systematic mapping study”. In: *Computer Standards & Interfaces* 50 (Feb. 2017), pp. 107–115. ISSN: 0920-5489. DOI: 10.1016/j.csi.2016.10.001 (cit. on p. 34).

[MQ17] Chunyan Mu and Shengchao Qin. “Time-sensitive Information Flow Control in Timed Event-B”. In: *The 11th International Symposium on Theoretical Aspects of Software Engineering*. TASE 2017 (Sophia Antipolis, France, Sept. 13–15, 2017). Proceedings. Ed. by Frédéric Mallet, Min Zhang, and Eric Madelaine. IEEE Computer Society, 2017, pp. 1–8. ISBN: 978-1-5386-1924-7. DOI: 10.1109/TASE.2017.8285631 (cit. on p. 110).

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. “Consistent Extra-Functional Properties Tagging for Component and Connector Models”. In: *Interplay of Model-Driven and Component-Based Software Engineering*. ModComp 2016 (Saint-Malo, France, Oct. 2, 2016). Proceedings of the 3rd International Workshop on Interplay of Model-Driven and Component-Based Software Engineering. Ed. by Federico Ciccozzi and Ivano Malavolta. CEUR Workshop Proceedings 1723. ISSN: 1613-0073. 2016, pp. 19–24. URN: urn:nbn:de:0074-1723-0 (cit. on p. 55).

[MS18] Azad M. Madni and Michael Sievers. “Model-based systems engineering. Motivation, current status, and research opportunities”. In: *Systems Engineering. The Journal of the International Council on Systems Engineering* 21.3 (May 2018): *20th Anniversary Special Issue*, pp. 172–190. ISSN: 1098-1241. DOI: 10.1002/sys.21438 (cit. on p. 4).

[MSB02] John McLean, Roger R. Schell, and Donald L. Brinkley. *Security Models*. In: *Encyclopedia of Software Engineering*. Ed. by John J. Marciniak. Wiley, 2002. ISBN: 978-0-471-37737-5. DOI: 10.1002/0471028959.sof297 (cit. on p. 28).

[MSS18] Chihab Eddine Mokaddem, Houari A. Sahraoui, and Eugene Syriani. “Recommending Model Refactoring Rules from Refactoring Examples”. In: *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS’18 (Copenhagen, Denmark, Oct. 14–19, 2018). Proceedings. Ed. by Andrzej Wąsowski, Richard F. Paige, and Øystein Haugen. ACM, 2018, pp. 257–266. ISBN: 978-1-4503-4949-9. DOI: 10.1145/3239372.3239406 (cit. on pp. 142, 148).

[MT00] Nenad Medvidovic and Richard N. Taylor. “A Classification and Comparison Framework for Software Architecture Description Languages”. In: *IEEE Transactions on Software Engineering* 26.1 (Jan. 2000), pp. 70–93. ISSN: 0098-5589. DOI: 10.1109/32.825767 (cit. on pp. 5, 52).

[MZ07] Ron van der Meyden and Chenyi Zhang. “Algorithmic Verification of Noninterference Properties”. In: *Electronic Notes in Theoretical Computer Science* 168 (Feb. 2007): *Proceedings of the Second International Workshop on Views on Designing Complex Architectures. VODCA 2006*. Ed. by Maurice ter Beek and Fabio Gadducci, pp. 61–75. issn: 1571-0661. doi: 10.1016/j.entcs.2006.11.002 (cit. on pp. 86, 109).

[MZ10] Ron van der Meyden and Chenyi Zhang. “A comparison of semantic models for noninterference”. In: *Theoretical Computer Science* 411.47 (Oct. 2010), pp. 4123–4147. issn: 0304-3975. doi: 10.1016/j.tcs.2010.08.013 (cit. on pp. 2, 109).

[NAY17] Phu Hong Nguyen, Shaukat Ali, and Tao Yue. “Model-based security engineering for cyber-physical systems. A systematic mapping study”. In: *Information and Software Technology* 83 (Mar. 2017), pp. 116–135. issn: 0950-5849. doi: 10.1016/j.infsof.2016.11.004 (cit. on p. 4).

[Nie+15] Claus Ballegaard Nielsen, Peter Gorm Larsen, John S. Fitzgerald, Jim Woodcock, and Jan Peleska. “Systems of Systems Engineering. Basic Concepts, Model-Based Techniques, and Research Directions”. In: *ACM Computing Surveys* 48.2, Art. No. 18 (Nov. 2015), pp. 1–41. issn: 0360-0300. doi: 10.1145/2794381 (cit. on p. 52).

[NKKT15] Phu Hong Nguyen, Max E. Kramer, Jacques Klein, and Yves Le Traon. “An extensive systematic review on the Model-Driven Development of secure systems”. In: *Information and Software Technology* 68 (Dec. 2015), pp. 62–81. issn: 0950-5849. doi: 10.1016/j.infsof.2015.08.006 (cit. on pp. 4, 34).

[NNV17] Flemming Nielson, Hanne Riis Nielson, and Panagiotis Vasilikos. “Information Flow for Timed Automata”. In: *Models, Algorithms, Logics and Tools. Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*. Ed. by Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare. Lecture Notes in Computer Science 10460: Festschrift. Springer, 2017, pp. 3–21. isbn: 978-3-319-63120-2. doi: 10.1007/978-3-319-63121-9_1 (cit. on p. 110).

[OD15] Samir Ouchani and Mourad Debbabi. “Specification, verification, and quantification of security in model-based systems”. In: *Computing* 97.7 (July 2015): *Special Issue on Contributions of computational intelligence in designing complex information systems*. Ed. by Ladjel Bellatreche, Abdelmalek Amine, and Otmane Aït Mohamed, pp. 691–711. issn: 0010-485X. doi: 10.1007/s00607-015-0445-x (cit. on p. 34).

[OL15] Samir Ouchani and Gabriele Lenzini. “Generating attacks in SysML activity diagrams by detecting attack surfaces”. In: *Journal of Ambient Intelligence and Humanized Computing* 6.3 (June 2015): *Special Issue on ANT2014*. Ed. by

Ansar-Ul-Haque Yasar, Zahoor Khan, and Elhadi M. Shakshuki, pp. 361–373. issn: 1868-5137. doi: 10.1007/s12652-015-0269-8 (cit. on p. 52).

[OMD13] Samir Ouchani, Otmane Aït Mohamed, and Mourad Debbabi. “A Security Risk Assessment Framework for SysML Activity Diagrams”. In: *2013 Seventh International Conference on Software Security and Reliability*. SERE 2013 (Gaithersburg, Maryland, USA, June 18–20, 2013). Proceedings. IEEE, 2013, pp. 227–236. isbn: 978-1-4799-0406-8. doi: 10.1109/SERE.2013.11 (cit. on p. 51).

[ORY01] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. “Local Reasoning about Programs that Alter Data Structures”. In: *Computer Science Logic*. 15th International Workshop. CSL 2001 (Paris, France, Sept. 10–13, 2001). Proceedings. Ed. by Laurent Fribourg. Lecture Notes in Computer Science 2142. Springer, 2001, pp. 1–19. isbn: 3-540-42554-3. doi: 10.1007/3-540-44802-0_1 (cit. on p. 55).

[OTH13] Robert F. Oates, Fran Thom, and Graham Herries. “Security-Aware, Model-Based Systems Engineering with SysML”. In: *1st International Symposium for ICS & SCADA Cyber Security Research 2013*. ICS-CSR 2013 (Leicester, UK, Sept. 16–17, 2013). Proceedings. Ed. by Helge Janicke and Kevin I. Jones. Electronic Workshops in Computing. BCS, 2013, pp. 78–87. isbn: 978-1-780172-32-3. doi: 10.14236/ewic/ICSCSR2013.9 (cit. on p. 52).

[PB13] Ludovic Piètre-Cambacédès and Marc Bouissou. “Cross-fertilization between safety and security engineering”. In: *Reliability Engineering & System Safety* 110 (Feb. 2013), pp. 110–126. issn: 0951-8320. doi: 10.1016/j.ress.2012.09.011 (cit. on p. 148).

[Pel+19] Sven Peldszus, Katja Tuma, Daniel Strüber, Jan Jürjens, and Riccardo Scandariato. “Secure Data-Flow Compliance Checks between Models and Code Based on Automated Mappings”. In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems*. MODELS 2019 (Munich, Germany, Sept. 15–20, 2019). Proceedings. Ed. by Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño. IEEE, 2019, pp. 23–33. isbn: 978-1-7281-2536-7. doi: 10.1109/MODELS.2019.00-18 (cit. on p. 150).

[Pel19] Doron A. Peled. “Formal Methods”. In: *Handbook of Software Engineering*. Ed. by Sungdeok Cha, Richard N. Taylor, and Kyo Chul Kang. Springer, 2019, pp. 193–222. isbn: 978-3-030-00261-9. doi: 10.1007/978-3-030-00262-6_5 (cit. on p. 2).

[PK13] Pavithra Prabhakar and Boris Köpf. “Verifying information flow properties of hybrid systems”. In: *Proceedings of the 2nd ACM International Conference on High Confidence Networked Systems*. HiCoNS 2013 (Philadelphia, Pennsylvania, USA, Apr. 9–11, 2013). Ed. by Linda Bushnell, Larry Rohrbough, Saurabh Amin, and Xenofon D. Koutsoukos. ACM, 2013, pp. 77–84. isbn: 978-1-4503-1961-4. doi: 10.1145/2461446.2461458 (cit. on p. 78).

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science* (Providence, Rhode Island, USA, Oct. 31–Nov. 1, 1977). issn: 0272-5428. IEEE Computer Society, 1977, pp. 46–57. doi: 10.1109/SFCS.1977.32 (cit. on p. 19).

[Rab16] Markus N. Rabe. “A temporal logic approach to information-flow control”. PhD thesis. Saarbrücken: Saarland University, Feb. 2016. doi: 10.22028/D291-26650 (cit. on pp. 86, 109).

[RFB12] Ana Luísa Ramos, José Vasconcelos Ferreira, and Jaume Barceló. “Model-Based Systems Engineering. An Emerging Approach for Modern Systems”. In: *IEEE Transactions on Systems, Man, and Cybernetics. Part C*. Applications and Reviews 42.1 (Jan. 2012): *Special Issue on Semantics*, pp. 101–111. issn: 1094-6977. doi: 10.1109/TSMCC.2011.2106495 (cit. on p. 4).

[RH09] Per Runeson and Martin Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empirical Software Engineering. An International Journal* 14.2 (Apr. 2009), pp. 131–164. issn: 1382-3256. doi: 10.1007/s10664-008-9102-8 (cit. on pp. 99, 104, 135, 138).

[RH13] A. William Roscoe and Jian Huang. “Checking noninterference in Timed CSP”. In: *Formal Aspects of Computing. Applicable Formal Methods* 25.1 (Jan. 2013), pp. 3–35. issn: 0934-5043. doi: 10.1007/s00165-012-0251-6 (cit. on p. 110).

[Rie15] Jan Rieke. “Model consistency management for systems engineering”. PhD thesis. University of Paderborn, Jan. 2015. Verlagsschriftenreihe des Heinz Nixdorf Instituts 335. issn: 2195-5239. urn: urn:nbn:de:hbz:466:2-15597 (cit. on pp. 17, 18, 62).

[RIKD18] Adrian Rutle, Ludovico Iovino, Harald König, and Zinovy Diskin. “Automatic Transformation Co-evolution Using Traceability Models and Graph Transformation”. In: *Modelling Foundations and Applications*. 14th European Conference. ECMFA 2018 (Toulouse, France, June 26–28, 2018). Proceedings. Ed. by Alfonso Pierantonio and Salvador Trujillo. Lecture Notes in Computer Science 10890. Springer, 2018, pp. 80–96. isbn: 978-3-319-92996-5. doi: 10.1007/978-3-319-92997-2_6 (cit. on p. 143).

[RJB17] Willard Rafnsson, Limin Jia, and Lujo Bauer. “Timing-Sensitive Noninterference through Composition”. In: *Principles of Security and Trust*. 6th International Conference. POST 2017 (Uppsala, Sweden, Apr. 24–25, 2017). Proceedings. Ed. by Matteo Maffei and Mark Ryan. Lecture Notes in Computer Science 10204: Advanced Research in Computing and Software Science. Springer, 2017, pp. 3–25. isbn: 978-3-662-54454-9. doi: 10.1007/978-3-662-54455-6_1 (cit. on pp. 56, 82, 110).

[RMKP12] Louis M. Rose, Nicholas Drivalos Matragkas, Dimitrios S. Kolovos, and Richard F. Paige. "A feature model for model-to-text transformation languages". In: *2012 4th International Workshop on Modeling in Software Engineering*. MiSE 2012 (Zürich, Switzerland, June 2–3, 2012). Proceedings. Ed. by Joanne M. Atlee, Robert Baillargeon, Robert B. France, Geri Georg, Ana Moreira, Bernhard Rumpe, and Steffen Zschaler. IEEE Computer Society, 2012, pp. 57–63. isbn: 978-1-4673-1757-3. doi: 10.1109/MISE.2012.6226015 (cit. on p. 12).

[RMR15] Jose Fran. Ruiz, Antonio Maña, and Carsten Rudolph. "An Integrated Security and Systems Engineering Process and Modelling Framework". In: *The Computer Journal. Section D*. Security in Computer Systems and Networks 58.10 (Oct. 2015): *Special Focus on Secure Information Systems Engineering*, pp. 2328–2350. issn: 0010-4620. doi: 10.1093/comjnl/bxu152 (cit. on p. 51).

[RNHR13] Andreas Rentschler, Qais Noorshams, Lucia Happe, and Ralf H. Reussner. "Interactive Visual Analytics for Efficient Maintenance of Model Transformations". In: *Theory and Practice of Model Transformations*. 6th International Conference. ICMT 2013 (Budapest, Hungary, June 18–19, 2013). Proceedings. Ed. by Keith Duddy and Gerti Kappel. Lecture Notes in Computer Science 7909. Springer, 2013, pp. 141–157. isbn: 978-3-642-38882-8. doi: 10.1007/978-3-642-38883-5_14 (cit. on p. 118).

[Ros95] A. William Roscoe. "CSP and determinism in security modelling". In: *1995 IEEE Symposium on Security and Privacy* (Oakland, California, USA, May 8–10, 1995). Proceedings. IEEE Computer Society, 1995, pp. 114–127. isbn: 0-8186-7015-0. doi: 10.1109/SECPRI.1995.398927 (cit. on pp. 106, 150).

[RS01] Peter Y. A. Ryan and Steve A. Schneider. "Process Algebra and Non-Interference". In: *Journal of Computer Security* 9.1–2 (2001). Ed. by Paul F. Syverson, pp. 75–103. issn: 0926-227X. doi: 10.3233/JCS-2001-91-204 (cit. on p. 2).

[RSVW10] Bernhard Rumpe, Martin Schindler, Steven Völkel, and Ingo Weisemöller. "Generative software development". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. Vol. 2. ICSE 2010 (Cape Town, South Africa, May 1–8, 2010). Proceedings. Ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel. ACM, 2010, pp. 473–474. isbn: 978-1-60558-719-6. doi: 10.1145/1810295.1810436 (cit. on p. 10).

[SA09] Joon Son and Jim Alves-Foss. "A formal framework for real-time information flow analysis". In: *Computers & Security* 28.6 (Sept. 2009), pp. 421–432. issn: 0167-4048. doi: 10.1016/j.cose.2009.01.005 (cit. on p. 110).

[Sab01] Andrei Sabelfeld. “The Impact of Synchronisation on Secure Information Flow in Concurrent Programs”. In: *Perspectives of System Informatics*. 4th International Andrei Ershov Memorial Conference. PSI 2001 (Akademgorodok, Novosibirsk, Russia, July 2–6, 2001). Revised Papers. Ed. by Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin. Lecture Notes in Computer Science 2244. Springer, 2001, pp. 225–239. isbn: 3-540-43075-X. doi: 10.1007/3-540-45575-2_22 (cit. on p. 32).

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *2010 IEEE Symposium on Security and Privacy* (Berkeley/Oakland, California, USA, May 16–19, 2010). Proceedings. IEEE Computer Society, 2010, pp. 317–331. isbn: 978-0-7695-4035-1. doi: 10.1109/SP.2010.26 (cit. on p. 149).

[SABB14] Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. “Model-Driven Information Flow Security for Component-Based Systems”. In: *From Programs to Systems. The Systems Perspective in Computing*. ETAPS Workshop. FPS 2014 (Grenoble, France, Apr. 6, 2014). Proceedings. Ed. by Saddek Bensalem, Yassine Lakhnech, and Axel Legay. Lecture Notes in Computer Science 8415: Festschrift. Springer, 2014, pp. 1–20. isbn: 978-3-642-54847-5. doi: 10.1007/978-3-642-54848-2_1 (cit. on pp. 56, 82).

[SARL13] Lilia Sfaxi, Takoua Abdellatif, Riadh Robbana, and Yassine Lakhnech. “Information flow control of component-based distributed systems”. In: *Concurrency and Computation. Practice and Experience* 25.2 (Feb. 2013): *New Technologies of Distributed Systems*, pp. 161–179. issn: 1532-0634. doi: 10.1002/cpe.2807 (cit. on pp. 56, 81).

[Sch06] Douglas C. Schmidt. “Model-Driven Engineering”. Guest Editor’s Introduction. In: *Computer. Innovative Technology for Computer Professionals* 39.2 (Feb. 2006), pp. 25–31. issn: 0018-9162. doi: 10.1109/MC.2006.58 (cit. on pp. 1, 9, 10).

[SEH19] David Schubert, Hendrik Eikerling, and Jörg Holtmann. “Application-aware Intrusion Detection. A Systematic Literature Review and Implications for Automotive Systems”. In: *17th escar Europe. Embedded Security in Cars* (Stuttgart, Germany, Nov. 19–20, 2019). Ruhr-Universität Bochum, 2019, pp. 29–43. doi: 10.13154/294-6654 (cit. on p. 149).

[SGM18] Christoph Schmittner, Gerhard Griessnig, and Zhendong Ma. “Status of the Development of ISO/SAE 21434”. In: *Systems, Software and Services Process Improvement*. 25th European Conference. EuroSPI 2018 (Bilbao, Spain, Sept. 5–7, 2018). Proceedings. Ed. by Xabier Larrucea, Izaskun Santamaria, Rory V. O’Connor, and Richard Messnarz. Communications in Computer and Information Science 896. Springer, 2018, pp. 504–513. isbn: 978-3-319-97924-3. doi: 10.1007/978-3-319-97925-0_43 (cit. on p. 3).

[She96] Sarah A. Sheard. "Twelve Systems Engineering Roles". In: *INCOSE International Symposium* 6.1 (July 1996), pp. 478–485. issn: 2334-5837. doi: 10.1002/j.2334-5837.1996.tb02042.x (cit. on p. 14).

[SHF18] Riccardo Scandariato, Jennifer Horkoff, and Robert Feldt. "Generative secure design, defined". In: *2018 ACM/IEEE 40th International Conference on Software Engineering. New Ideas and Emerging Results*. ICSE-NIER 2018 (Gothenburg, Sweden, May 30–June 1, 2018). Proceedings. Ed. by Andrea Zisman and Sven Apel. ACM, 2018, pp. 1–4. isbn: 978-1-4503-5662-6. doi: 10.1145/3183399.3183400 (cit. on p. 148).

[Shi19] Shin-Shing Shin. "Empirical study on the effectiveness and efficiency of model-driven architecture techniques". In: *Software and Systems Modeling* 18.5 (Oct. 2019), pp. 3083–3096. issn: 1619-1366. doi: 10.1007/s10270-018-00711-y (cit. on p. 118).

[Sil15] Alberto Rodrigues da Silva. "Model-driven engineering. A survey supported by the unified conceptual model". In: *Computer Languages, Systems & Structures* 43 (Oct. 2015), pp. 139–155. issn: 1477-8424. doi: 10.1016/j.cl.2015.06.001 (cit. on pp. 1, 9).

[SK03] Shane Sendall and Wojtek Kozaczynski. "Model Transformation. The Heart and Soul of Model-Driven Software Development". In: *IEEE Software* 20.5 (Sept.–Oct. 2003). Ed. by Stephen J. Mellor, Anthony N. Clark, and Takao Futagami, pp. 42–45. issn: 0740-7459. doi: 10.1109/MS.2003.1231150 (cit. on pp. 6, 12, 13).

[SK12] Prabhakaran Salini and Selvadurai Kanmani. "Survey and analysis on Security Requirements Engineering". In: *Computers & Electrical Engineering* 38.6 (Nov. 2012), pp. 1785–1797. issn: 0045-7906. doi: 10.1016/j.compeleceng.2012.08.008 (cit. on p. 4).

[SLCS12] Mehrdad Saadatmand, Thomas Leveque, Antonio Cicchetti, and Mikael Sjödin. "Managing Timing Implications of Security Aspects in Model-Driven Development of Real-Time Embedded Systems". In: *International Journal on Advances in Security* 5.3&4 (2012), pp. 68–80. issn: 1942-2636. url: http://www.iariajournals.org/security/tocv5n34.html (cit. on pp. 56, 80, 149).

[SM03] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (Jan. 2003). Ed. by Li Gong, Joshua D. Guttman, Peter Y. A. Ryan, and Steve A. Schneider, pp. 5–19. issn: 0733-8716. doi: 10.1109/JSAC.2002.806121 (cit. on pp. 2, 28, 109).

[Smi07] Geoffrey Smith. “Principles of Secure Information Flow Analysis”. In: *Malware Detection*. Ed. by Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang. Advances in Information Security 27. Springer, 2007. Chap. 13, pp. 291–307. isbn: 978-0-387-32720-4. doi: 10.1007/978-0-387-44599-1_13 (cit. on pp. 2, 28).

[Smi09] Geoffrey Smith. “On the Foundations of Quantitative Information Flow”. In: *Foundations of Software Science and Computational Structures*. 12th International Conference. FOSSACS 2009 (York, UK, Mar. 22–25, 2009). Proceedings. Ed. by Luca de Alfaro. Lecture Notes in Computer Science 5504. Springer, 2009, pp. 288–302. isbn: 978-3-642-00595-4. doi: 10.1007/978-3-642-00596-1_21 (cit. on p. 107).

[Smi82] Brian Cantwell Smith. “Procedural reflection in programming languages”. PhD thesis. Cambridge: Massachusetts Institute of Technology, Feb. 1982. url: http://hdl.handle.net/1721.1/15961 (cit. on p. 133).

[SRVK11] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. “Metamodelling. State of the Art and Research Challenges”. In: *Model-Based Engineering of Embedded Real-Time Systems*. International Dagstuhl Workshop. MBEERTS 2007 (Dagstuhl Castle, Germany, Nov. 4–9, 2007). Revised Selected Papers. Ed. by Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz. Lecture Notes in Computer Science 6100: State-of-the-Art Survey. Springer, 2011. Chap. 3, pp. 57–76. isbn: 978-3-642-16276-3. doi: 10.1007/978-3-642-16277-0_3 (cit. on p. 10).

[SS09] Andrei Sabelfeld and David Sands. “Declassification. Dimensions and principles”. In: *Journal of Computer Security* 17.5 (2009): *18th IEEE Computer Security Foundations Symposium. CSF 18*. Ed. by Joshua D. Guttman, pp. 517–548. issn: 0926-227X. doi: 10.3233/JCS-2009-0352 (cit. on pp. 49, 79, 107, 147).

[SSCC09] Séverine Sentilles, Petr Stepan, Jan Carlson, and Ivica Crnković. “Integration of Extra-Functional Properties in Component Models”. In: *Component-Based Software Engineering*. 12th International Symposium. CBSE 2009 (East Stroudsburg, Pennsylvania, USA, June 24–26, 2009). Proceedings. Ed. by Grace A. Lewis, Iman Poernomo, and Christine Hofmeister. Lecture Notes in Computer Science 5582. Springer, 2009, pp. 173–190. isbn: 978-3-642-02413-9. doi: 10.1007/978-3-642-02414-6_11 (cit. on p. 55).

[SSS09] Fredrik Seehusen, Bjørnar Solhaug, and Ketil Stølen. “Adherence preserving refinement of trace-set properties in STAIRS. Exemplified for information flow properties and policies”. In: *Software and Systems Modeling* 8.1 (Feb. 2009), pp. 45–65. issn: 1619-1366. doi: 10.1007/s10270-008-0102-3 (cit. on p. 50).

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. German. Springer, 1973. isbn: 3-211-81106-0 (cit. on pp. 9, 10).

[Ste+12] Curtis Steward, Luay A. Wahsheh, Aftab Ahmad, Jonathan M. Graham, Cheryl V. Hinds, Aurelia T. Williams, and Sandra J. DeLoatch. “Software Security. The Dangerous Afterthought”. In: *Proceedings of the Ninth International Conference on Information Technology. New Generations*. ITNG 2012 (Las Vegas, Nevada, USA, Apr. 16–18, 2012). Ed. by Shahram Latifi. IEEE Computer Society, 2012, pp. 815–818. isbn: 978-0-7695-4654-4. doi: 10.1109/ITNG.2012.60 (cit. on pp. 4, 33).

[Sti03] Colin Stirling. “Bisimulation and Language Equivalence”. In: *Logic for Concurrency and Synchronisation*. Ed. by Ruy J. G. B. de Queiroz. Trends in Logic: Studia Logica Library 18. Kluwer, 2003. Chap. 7, pp. 269–284. isbn: 1-4020-1270-5. doi: 10.1007/0-306-48088-3_7 (cit. on p. 71).

[Sti98] Colin Stirling. “The Joys of Bisimulation”. In: *Mathematical Foundations of Computer Science 1998*. 23rd International Symposium. MFCS’98 (Brno, Czech Republic, Aug. 24–28, 1998). Proceedings. Ed. by Luboš Brim, Jozef Gruska, and Jiří Zlatuška. Lecture Notes in Computer Science 1450. Springer, 1998, pp. 142–151. isbn: 3-540-64827-5. doi: 10.1007/BFb0055763 (cit. on pp. 25, 73).

[Str+16] Daniel Strüber, Timo Kehrer, Thorsten Arendt, Christopher Pietsch, and Dennis Reuling. “Scalability of Model Transformations. Position Paper and Benchmark Set”. In: *Scalable Model Driven Engineering*. BigMDE 2016 (Vienna, Austria, July 8, 2016). Proceedings of the 4rd Workshop on Scalable Model Driven Engineering. Ed. by Dimitrios S. Kolovos, Davide Di Ruscio, Nicholas Drivalos Matragkas, Jesús Sánchez Cuadrado, István Ráth, and Massimo Tisi. CEUR Workshop Proceedings 1652. issn: 1613-0073. 2016, pp. 21–30. urn: urn:nbn:de:0074-1652-4 (cit. on pp. 113, 118).

[Str+18] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. “Variability-based model transformation. Formal foundation and application”. In: *Formal Aspects of Computing. Applicable Formal Methods* 30.1 (Jan. 2018): *Extended versions of papers presented at FASE’16*, pp. 133–162. issn: 0934-5043. doi: 10.1007/s00165-017-0441-3 (cit. on p. 118).

[Sty04] Martin R. Stytz. “Considering Defense in Depth for Software Applications”. In: *IEEE Security & Privacy. Building Confidence in a Networked World* 2.1 (Jan.–Feb. 2004), pp. 72–75. issn: 1540-7993. doi: 10.1109/MSECP.2004.1264860 (cit. on p. 149).

[Sun+14] Cong Sun, Ning Xi, Jinku Li, Qingsong Yao, and Jianfeng Ma. “Verifying Secure Interface Composition for Component-Based System Designs”. In: *21st Asia-Pacific Software Engineering Conference*. Vol. 1: *Research Papers*. APSEC 2014 (Jeju, South Korea, Dec. 1–4, 2014). Proceedings. Ed. by Sungdeok Cha, Yann-Gaël Guéhéneuc, and Gihwon Kwon. IEEE Computer

Society, 2014, pp. 359–366. isbn: 978-1-4799-7425-2. doi: 10.1109/APSEC.2014.60 (cit. on p. 82).

[SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development. Technology, Engineering, Management*. In collab. with Jorn Bettin, Arno Haase, and Simon Helsen. Trans. by Bettina von Stockfleth. With a forew. by Krzysztof Czarnecki. Wiley, 2006. isbn: 978-0-470-02570-3 (cit. on pp. 9, 10, 12).

[SW07] Wilhelm Schäfer and Heike Wehrheim. "The Challenges of Building Advanced Mechatronic Systems". In: *Future of Software Engineering*. FoSE 2007 (Minneapolis, Minnesota, USA, May 23–25, 2007). Ed. by Lionel C. Briand and Alexander L. Wolf. IEEE Computer Society, 2007, pp. 72–84. isbn: 0-7695-2829-5. doi: 10.1109/FOSE.2007.28 (cit. on p. 1).

[Szy02] Clemens A. Szyperski. *Component software. Beyond object-oriented programming*. In collab. with Dominik Gruntz and Stephan Murer. 2nd ed. Addison-Wesley Component Software Series. Addison-Wesley Longman, 2002. isbn: 0-201-74572-0 (cit. on pp. 5, 18).

[TA05] Tachio Terauchi and Alexander Aiken. "Secure Information Flow as a Safety Problem". In: *Static Analysis*. 12th International Symposium. SAS 2005 (London, UK, Sept. 7–9, 2005). Proceedings. Ed. by Chris Hankin and Igor Siveroni. Lecture Notes in Computer Science 3672. Springer, 2005, pp. 352–367. isbn: 3-540-28584-9. doi: 10.1007/11547662_24 (cit. on p. 109).

[TÇS18] Katja Tuma, Gül Çalikli, and Riccardo Scandariato. "Threat analysis of software systems. A systematic literature review". In: *Journal of Systems and Software* 144 (Oct. 2018), pp. 275–294. issn: 0164-1212. doi: 10.1016/j.jss.2018.06.073 (cit. on pp. 46, 48).

[Tes04] Luca Tesei. "Specification and Verification using Timed Automata". PhD thesis. University of Pisa, May 2004 (cit. on p. 111).

[The+18] Christopher Theisen, Nuthan Munaiah, Mahran Al-Zyoud, Jeffrey C. Carver, Andrew Meneely, and Laurie Williams. "Attack surface definitions. A systematic literature review". In: *Information and Software Technology* 104 (Dec. 2018), pp. 94–103. issn: 0950-5849. doi: 10.1016/j.infsof.2018.07.008 (cit. on p. 1).

[Tis+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. "On the Use of Higher-Order Model Transformations". In: *Model Driven Architecture. Foundations and Applications*. 5th European Conference. ECMDA-FA 2009 (Enschede, Netherlands, June 23–26, 2009). Proceedings. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Lecture Notes in Computer Science 5562. Springer, 2009, pp. 18–33. isbn: 978-3-642-02673-7. doi: 10.1007/978-3-642-02674-4_3 (cit. on p. 140).

[TJM08] Inger Anne Tøndel, Martin Gilje Jaatun, and Per Håkon Meland. “Security Requirements for the Rest of Us. A Survey”. In: *IEEE Software* 25.1 (Jan.–Feb. 2008): *Security for the Rest of Us*. Ed. by Konstantin Beznosov and Brian Chess, pp. 20–27. issn: 0740-7459. doi: 10.1109/MS.2008.19 (cit. on p. 4).

[TSB19] Katja Tuma, Riccardo Scandariato, and Musard Balliu. “Flaws in Flows. Unveiling Design Flaws via Information Flow Analysis”. In: *IEEE International Conference on Software Architecture*. ICSA 2019 (Hamburg, Germany, Mar. 25–29, 2019). Proceedings. IEEE, 2019, pp. 191–200. isbn: 978-1-7281-0528-4. doi: 10.1109/ICSA.2019.00028 (cit. on p. 149).

[TSHL16] Klaus Thoma, Benjamin Scharte, Daniel Hiller, and Tobias Leismann. “Resilience Engineering as Part of Security Research. Definitions, Concepts and Science Approaches”. In: *European Journal for Security Research* 1.1 (Apr. 2016), pp. 3–19. issn: 2365-0931. doi: 10.1007/s41125-016-0002-4 (cit. on pp. 147, 148).

[Tür17] Sven Türpe. “The Trouble with Security Requirements”. In: *2017 IEEE 25th International Requirements Engineering Conference*. RE 2017 (Lisbon, Portugal, Sept. 4–8, 2017). Proceedings. IEEE Computer Society, 2017, pp. 122–133. isbn: 978-1-5386-3191-1. doi: 10.1109/RE.2017.13 (cit. on pp. 4, 48).

[UFF12] Anton V. Uzunov, Eduardo B. Fernández, and Katrina E. Falkner. “Engineering Security into Distributed Systems. A Survey of Methodologies”. In: *Journal of Universal Computer Science* 18.20 (2012), pp. 2920–3006. issn: 0948-695x. doi: 10.3217/jucs-018-20-2920 (cit. on p. 34).

[UFF13] Anton V. Uzunov, Katrina E. Falkner, and Eduardo B. Fernández. “Decomposing Distributed Software Architectures for the Determination and Incorporation of Security and Other Non-functional Requirements”. In: *2013 22nd Australasian Conference on Software Engineering*. ASWEC 2013 (Melbourne, Australia, June 4–7, 2013). Proceedings. IEEE Computer Society, 2013, pp. 30–39. isbn: 978-0-7695-4995-8. doi: 10.1109/ASWEC.2013.14 (cit. on p. 81).

[UFF15] Anton V. Uzunov, Katrina E. Falkner, and Eduardo B. Fernández. “A comprehensive pattern-oriented approach to engineering security methodologies”. In: *Information and Software Technology* 57 (Jan. 2015), pp. 217–247. issn: 0950-5849. doi: 10.1016/j.infsof.2014.09.001 (cit. on pp. 44–46).

[UFF18] Anton V. Uzunov, Eduardo B. Fernández, and Katrina E. Falkner. “Assessing and improving the quality of security methodologies for distributed systems”. In: *Journal of Software. Evolution and Process* 30.11, Art. No. e1980 (Nov. 2018), pp. 1–56. issn: 2047-7473. doi: 10.1002/smr.1980 (cit. on pp. 34–36, 43, 44, 46, 47, 53, 145).

[Val+16] Tassio Vale, Ivica Crnković, Eduardo Santana de Almeida, Paulo Anselmo da Mota Silveira Neto, Yguaratã Cerqueira Cavalcanti, and Silvio Romero de Lemos Meira. "Twenty-eight years of component-based software engineering". In: *Journal of Systems and Software* 111 (Jan. 2016), pp. 128–148. issn: 0164-1212. doi: 10.1016/j.jss.2015.09.019 (cit. on pp. 5, 18).

[Var06] Dániel Varró. "Model Transformation by Example". In: *Model Driven Engineering Languages and Systems*. 9th International Conference. MoDELS 2006 (Genova, Italy, Oct. 1–6, 2006). Proceedings. Ed. by Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio. Lecture Notes in Computer Science 4199. Springer, 2006, pp. 410–424. isbn: 3-540-45772-0. doi: 10.1007/11880240_29 (cit. on p. 142).

[VBT15] Antonio Vetro, Wolfgang Böhm, and Marco Torchiano. "On the Benefits and Barriers When Adopting Software Modelling and Model Driven Techniques. An External, Differentiated Replication". In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM 2015 (Beijing, China, Oct. 22–23, 2015). Proceedings. IEEE, 2015, pp. 168–171. isbn: 978-1-4673-7899-4. doi: 10.1109/ESEM.2015.7321210 (cit. on p. 1).

[VGNH14] Maria Vasilevskaya, Linda Ariani Gunawan, Simin Nadjm-Tehrani, and Peter Herrmann. "Integrating security mechanisms into embedded systems by domain-specific modelling". In: *Security and Communication Networks* 7.12 (Dec. 2014), pp. 2815–2832. issn: 1939-0114. doi: 10.1002/sec.819 (cit. on p. 51).

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2–3 (1996), pp. 167–188. issn: 0926-227X. doi: 10.3233/JCS-1996-42-304 (cit. on p. 109).

[VN15] Maria Vasilevskaya and Simin Nadjm-Tehrani. "Quantifying Risks to Data Assets Using Formal Metrics in Embedded System Design". In: *Computer Safety, Reliability, and Security*. 34th International Conference. SAFECOMP 2015 (Delft, Netherlands, Sept. 23–25, 2015). Proceedings. Ed. by Floor Koornneef and Coen van Gulijk. Lecture Notes in Computer Science 9337. Springer, 2015, pp. 347–361. isbn: 978-3-319-24254-5. doi: 10.1007/978-3-319-24255-2_25 (cit. on p. 51).

[VN16] Maria Vasilevskaya and Simin Nadjm-Tehrani. "Model-Based Security Risk Analysis for Networked Embedded Systems". In: *Critical Information Infrastructures Security*. 9th International Conference. CRITIS 2014 (Limassol, Cyprus, Oct. 13–15, 2014). Revised Selected Papers. Ed. by Christos G. Panayiotou, Georgios Ellinas, Elias Kyriakides, and Marios M. Polycarpou. Lecture Notes in Computer Science 8985. Springer, 2016, pp. 381–386. isbn: 978-3-319-31663-5. doi: 10.1007/978-3-319-31664-2_39 (cit. on p. 51).

[VNN18] Panagiotis Vasilikos, Flemming Nielson, and Hanne Riis Nielson. "Secure Information Release in Timed Automata". In: *Principles of Security and Trust*. 7th International Conference. POST 2018 (Thessaloniki, Greece, Apr. 16–17, 2018). Proceedings. Ed. by Lujo Bauer and Ralf Küsters. Lecture Notes in Computer Science 10804: Advanced Research in Computing and Software Science. Springer, 2018, pp. 28–52. isbn: 978-3-319-89721-9. doi: 10.1007/978-3-319-89722-6_2 (cit. on pp. 86, 110).

[Wan04] Farn Wang. "Formal verification of timed systems. A survey and perspective". In: *Proceedings of the IEEE* 92.8 (Aug. 2004), pp. 1283–1305. issn: 0018-9219. doi: 10.1109/JPROC.2004.831197 (cit. on pp. 6, 85).

[WBM14] Michael Waidner, Michael Backes, and Jörn Müller-Quade, eds. *Development of Secure Software with Security by Design*. In collab. with Eric Bodden et al. Trends and Strategy Report. SIT Technical Reports SIT-TR-2014-03. Fraunhofer SIT. July 2014. isbn: 978-3-8396-0768-8 (cit. on p. 4).

[WDR13] Md Tawhid Bin Waez, Jürgen Dingel, and Karen Rudie. "A survey of timed automata for the development of real-time systems". In: *Computer Science Review* 9 (Aug. 2013), pp. 1–26. issn: 1574-0137. doi: 10.1016/j.cosrev.2013.05.001 (cit. on p. 5).

[Wey19] Danny Weyns. "Software Engineering of Self-adaptive Systems". In: *Handbook of Software Engineering*. Ed. by Sungdeok Cha, Richard N. Taylor, and Kyo Chul Kang. Springer, 2019, pp. 399–443. isbn: 978-3-030-00261-9. doi: 10.1007/978-3-030-00262-6_11 (cit. on p. 14).

[Wim+10] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. "Surviving the Heterogeneity Jungle with Composite Mapping Operators". In: *Theory and Practice of Model Transformations*. Third International Conference. ICMT 2010 (Málaga, Spain, June 28–July 2, 2010). Proceedings. Ed. by Laurence Tratt and Martin Gogolla. Lecture Notes in Computer Science 6142. Springer, 2010, pp. 260–275. isbn: 978-3-642-13687-0. doi: 10.1007/978-3-642-13688-7_18 (cit. on pp. 117, 118, 121, 135, 139, 141).

[Wim+12] Manuel Wimmer et al. "Surveying Rule Inheritance in Model-to-Model Transformation Languages". In: *Journal of Object Technology* 11.2, Art. No. 3 (Aug. 2012), pp. 1–46. issn: 1660-1769. doi: 10.5381/jot.2012.11.2.a3 (cit. on p. 118).

[WL13] Manuel Wimmer and Philip Langer. "A Benchmark for Model Matching Systems. The Heterogeneous Metamodel Case". In: *Softwaretechnik-Trends* 33.2 (May 2013), pp. 101–104. issn: 0720-8928. url: https://dl.gi.de/handle/20.500.12116/8773. doi: 10.1007/s40568-013-0062-9 (cit. on p. 139).

[WL97] Carsten Weise and Dirk Lenzkes. “Efficient Scaling-Invariant Checking of Timed Bisimulation”. In: *14th Annual Symposium on Theoretical Aspects of Computer Science*. STACS 97 (Lübeck, Germany, Feb. 27–Mar. 1, 1997). Proceedings. Ed. by Rüdiger Reischuk and Michel Morvan. Lecture Notes in Computer Science 1200. Springer, 1997, pp. 177–188. isbn: 3-540-62616-6. doi: 10.1007/BFb0023458 (cit. on pp. 105, 109).

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. “Formal methods. Practice and experience”. In: *ACM Computing Surveys* 41.4, Art. No. 19 (Oct. 2009): *Special Issue on Software Verification*, pp. 1–36. issn: 0360-0300. doi: 10.1145/1592434.1592436 (cit. on p. 2).

[WM16] Oliver Woizekowski and Ron van der Meyden. “On Reductions from Multi-Domain Noninterference to the Two-Level Case”. In: *Computer Security. ESORICS 2016*. Proceedings. Part I. 21st European Symposium on Research in Computer Security (Heraklion, Greece, Sept. 26–30, 2016). Ed. by Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows. Lecture Notes in Computer Science 9878. Springer, 2016, pp. 520–537. isbn: 978-3-319-45743-7. doi: 10.1007/978-3-319-45744-4_26 (cit. on p. 79).

[Wol+20] Sabine Wolny, Alexandra Mazak, Christine Carpella, Verena Geist, and Manuel Wimmer. “Thirteen years of SysML. A systematic mapping study”. In: *Software and Systems Modeling* 19.1 (Jan. 2020), pp. 111–169. issn: 1619-1374. doi: 10.1007/s10270-019-00735-y (cit. on p. 4).

[WR10] Katinka Wolter and Philipp Reinecke. “Performance and Security Tradeoff”. In: *Formal Methods for Quantitative Aspects of Programming Languages*. 10th International School on Formal Methods for the Design of Computer, Communication and Software Systems. SFM 2010 (Bertinoro, Italy, June 21–26, 2010). Advanced Lectures. Ed. by Alessandro Aldini, Marco Bernardo, Alessandra Di Pierro, and Herbert Wiklicky. Lecture Notes in Computer Science 6154: Tutorial. Springer, 2010, pp. 135–167. isbn: 978-3-642-13677-1. doi: 10.1007/978-3-642-13678-8_4 (cit. on p. 149).

[WSD10] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. “Module superimposition. A composition technique for rule-based model transformation languages”. In: *Software and Systems Modeling* 9.3 (June 2010), pp. 285–309. issn: 1619-1366. doi: 10.1007/s10270-009-0134-3 (cit. on pp. 133, 134).

[WY14] Jingming Wang and Huiqun Yu. “Analysis of the Composition of Non-Deducibility in Cyber-Physical Systems”. In: *Applied Mathematics & Information Sciences* 8.6 (2014), pp. 3137–3143. issn: 1935-0090. doi: 10.12785/amis/080655 (cit. on p. 82).

[XL19] Wenjun Xiong and Robert Lagerström. “Threat modeling. A systematic literature review”. In: *Computers & Security* 84 (July 2019), pp. 53–69. issn: 0167-4048. doi: 10.1016/j.cose.2019.03.010 (cit. on pp. 46, 48).

[Yi91] Wang Yi. "A Calculus of Real Time Systems". PhD thesis. Göteborg: Chalmers University of Technology, 1991. Doktorsavhandlingar vid Chalmers Tekniska Högskola: new ser. 806. ISBN: 91-7032-589-8 (cit. on pp. 25, 75, 77).

[Yur+17] Kateryna Yurchenko, Moritz Behr, Heiko Klare, Max E. Kramer, and Ralf H. Reussner. "Architecture-driven Reduction of Specification Overhead for Verifying Confidentiality in Component-based Software Systems". In: *MODELS 2017 Satellite Events*. MODELS-SE 2017 (Austin, Texas, USA, Sept. 17, 2017). Proceedings of MODELS 2017 Satellite Event. Ed. by Loli Burgueño et al. CEUR Workshop Proceedings 2019. ISSN: 1613-0073. 2017, pp. 321–323. URN: urn:nbn:de:0074-2019-2 (cit. on p. 82).

[ZA08] Jie Zhou and Jim Alves-Foss. "Security policy refinement and enforcement for the design of multi-level secure systems". In: *Journal of Computer Security* 16.2 (2008): *Privacy, Security and Trust (PST) Technologies. Evolution and Challenges*. Ed. by George O. M. Yee, Ali A. Ghorbani, and Patrick C. K. Hung, pp. 107–131. ISSN: 0926-227X. DOI: 10.3233/JCS-2008-16202 (cit. on p. 81).

[ZFR13] Liguo Zhang, Yaser P. Fallah, and Jihene Rezgui. "Cyber-Physical Systems. Computation, Communication, and Control". In: *International Journal of Distributed Sensor Networks* 9.2, Art. No. 475818 (Feb. 2013), pp. 1–2. ISSN: 1550-1477. DOI: 10.1155/2013/475818 (cit. on p. 1).

[ZJKK17] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. "Perceptions on the State of the Art in Verification and Validation in Cyber-Physical Systems". In: *IEEE Systems Journal. A Publication of the IEEE Systems Council* 11.4 (Dec. 2017), pp. 2614–2627. ISSN: 1937-9234. DOI: 10.1109/JSYST.2015.2496293 (cit. on p. 2).

[ZL95] Aris Zakinthinos and E. Stewart Lee. "The Composability of Non-Interference". In: *Journal of Computer Security* 3.4 (1995), pp. 269–282. ISSN: 0926-227X. DOI: 10.3233/JCS-1994/1995-3404 (cit. on pp. 56, 71, 83).

[ZL96] Aris Zakinthinos and E. Stewart Lee. "How and why feedback composition fails". In: *9th IEEE Computer Security Foundations Workshop* (Kenmare, Ireland, June 10–12, 1996). Proceedings. IEEE Computer Society, 1996, pp. 95–101. ISBN: 0-8186-7522-5. DOI: 10.1109/CSFW.1996.503694 (cit. on pp. 55, 71, 78, 83).

[Zsc10] Steffen Zschaler. "Formal specification of non-functional properties of component-based software systems. A semantic framework and some applications thereof". In: *Software and Systems Modeling* 9.2 (Apr. 2010), pp. 161–201. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0115-6 (cit. on p. 55).

Standards

[ISO20] International Organization for Standardization and Society of Automotive Engineers. *Road vehicles. Cybersecurity engineering*. ISO/SAE DIS 21434. 2020. URL: https://www.iso.org/standard/70918.html. In preparation (cit. on p. 3).

[OMG13] Object Management Group. *Business Process Model and Notation*. BPMN. formal/2013-12-09. Version 2.0.2. Dec. 2013. URL: http://www.omg.org/spec/BPMN/2.0.2 (cit. on pp. 14, 17, 88).

[OMG14] Object Management Group. *Object Constraint Language*. formal/2014-02-03. Version 2.4. Feb. 2014. URL: http://www.omg.org/spec/OCL/2.4 (cit. on pp. 12, 122, 128).

[OMG16] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. formal/2016-06-03. Version 1.3. June 2016. URL: http://www.omg.org/spec/QVT/1.3 (cit. on pp. 13, 117, 134, 136).

[OMG17] Object Management Group. *OMG Unified Modeling Language*. OMG UML. formal/17-12-05. Version 2.5.1. Dec. 2017. URL: http://www.omg.org/spec/UML/2.5.1 (cit. on pp. 10, 23, 34, 116).

[OMG19] Object Management Group. *OMG Systems Modeling Language*. OMG SysML. formal/19-11-01. Version 1.6. Nov. 2019. URL: http://www.omg.org/spec/SysML/1.6 (cit. on p. 4).