ACM/IEEE 18th International Conference on
Model Driven Engineering Languages and Systems

September 28, 2015 • Ottawa (Canada)

Joint proceedings of

ACES-MB 2015 – Model-based Architecting of Cyber-physical and Embedded Systems

and

WUCOR 2015 – UML Consistency Rules

Iulia Dragomir, Susanne Graf, Gabor Karsai, Florian Noyrit, Iulian Ober,
Damiano Torre, Yvan Labiche, Marcela Genero, Maged Elaasar (Eds.)

Published on November 2015

Editors' addresses:

Iulia Dragomir
Aalto University, Department of Computer Science, PO Box 15400, FI-00076 Aalto, Finland

Susanne Graf
VERIMAG, 2 Avenue de Vignate, F-38610 Gieres, France

Gabor Karsai
Institute for Software Integrated Systems, Vanderbilt University, Box 351829, Nashville, TN 37235-1829, USA

Florian Noyrit
CEA LIST/LISE, Point Courrier 174, 91191 Gif-sur-Yvette, France

Iulian Ober
IRIT-University of Toulouse, 118 Route de Narbonne, 31062 Toulouse, France

Damiano Torre
Carleton University, Software Quality Engineering Laboratory, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada, and
University of Castilla-La Mancha, ALARCOS Research Group, Calle Altagracia, 50, 13071 Ciudad Real, Spain

Yvan Labiche
Carleton University, Software Quality Engineering Laboratory, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada

Marcela Genero
University of Castilla-La Mancha, ALARCOS Research Group, Calle Altagracia, 50, 13071 Ciudad Real, Spain

Maged Elaasar
Carleton University, Software Quality Engineering Laboratory, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada

Table of Contents

Introduction to ACES^{MB} 2015
– Model-based Architecting of Cyber-physical and Embedded Systems –

Iulia Dragomir*, Susanne Graf†, Gabor Karsai‡, Florian Noyrit§, Iulian Ober¶
* Aalto University, Finland. iulia.dragomir@aalto.fi
† CNRS-VERIMAG, France. susanne.graf@imag.fr
‡ ISIS, Vanderbilt University, USA. gabor.karsai@vanderbilt.edu
§ CEA LIST, France. florian.noyrit@cea.fr
¶ IRIT-University of Toulouse, France. iulian.ober@irit.fr

***Abstract*—The 8th ACES^{MB} workshop took place on September 28, 2015 at the 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS). The workshop brought together researchers and practitioners who work in the area of cyber-physical systems and apply model-based architecting techniques and tools. The workshop presented novel approaches, both theoretical and practical, as well as early results of their applications.**

Introduction

The design of embedded and cyber-physical systems with real-time and other critical constraints raises distinctive problems throughout the development process, in particular from system specifications to obtaining correct implementations. On the high-level side, system design is much more an art than a systematic activity, while on the low-level side design teams have to make specific architectural choices and handle non-functional constraints like real-time deadlines, energy consumption, etc., as early as possible in order to streamline the system development process. Model-based engineering techniques have now been established as the norm in industry since they are a major factor for further gains in productivity, quality and time-to-market such complex systems. They provide means to capture dedicated architectural and non-functional information in precise (and even formal) domain-specific models. They support compositional design of systems, in which functional aspects (platform independent) are separated from architectural and non-functional aspects (platform specific) until the integration step, etc. Many of the mentioned topics are emerging research areas, where efforts and results are still expected.

The 8th workshop on *Model-based architecting of cyber-physical and embedded systems*[1] brought together researchers and practitioners interested in model-based engineering to share and explore innovative ideas and experiences that contribute to better architecting embedded and cyber-physical systems, with a focus on approaches yielding efficient and provably correct designs.

[1] http://www.irit.fr/ACES-MB

Workshop contributions

The interest of the research and practice community for the ACES^{MB} workshop is attested by the steady number of attendees – around 20 people – at each edition. This year's program focused on novel approaches for the correct design of cyber-physical systems, both at the theoretical and the practical level. The selected talks ranged through experiences and challenges in industrial model-based engineering, incorporating formal methods in model-based systems engineering, and tools supporting design and analysis techniques. These contributions consist of an invited talk and 3 paper presentations detailed hereafter.

Invited talk. The keynote was given by Dr. Tao Yue from the Simula Research Laboratory, Norway, who discussed her experiences and insights gained from investigating the application of model-based engineering in different industrial domains (i.e., Communication, Oil&Gas, Maritime, Automated Material Handling and Geo Sports), for addressing different industrial challenges (i.e., Requirements, Architecture and Design, Testing, Product Line), and using diverse model-based engineering technologies.

Paper talks. 3 full papers had been accepted for presentation at the workshop, out of 4 full papers and 4 short papers, each being peer-reviewed by three independent reviewers. A synopsis of each presentation is given below.

I. Ruchkin et al. discussed dependency loops in the analysis of cyber-physical system designs. The authors defined the concept of dependency loops in the analysis process, as well as a spectrum of various relevant properties of the workflow, ranging from strong convergence to strong divergence. The authors also made proposals for resolving the analysis loops: iteration, constraint solving, and genetic algorithms.

J.O. Blech et al. introduced the concept of behavioral types for space-aware systems as a means to facilitate the development, commissioning, maintenance, and refactoring of cyber-physical systems with spatial characteristics. The approach is intended to be used in industrial automation, for which a formalization and analysis approach were presented.

V. Aravantinos et al. presented AUTOFOCUS3: a toolsuite for the design and analysis of cyber-physical systems. The

toolchain is comprehensive, as it covers a broad area of the engineering workflow (from requirements to design to synthesis to analysis, including safety analysis) and is well supported by tools. The authors paid particular attention to the details of the tool integration techniques used.

ACKNOWLEDGMENTS

We would like to thank the authors for submitting their papers at ACES^{MB} and the participants for their lively discussions, thus making the workshop successful. We especially thank our keynote speaker, Dr. Tao Yue, for her insightful presentation. We are grateful to the Program Committee and Steering Committee members for their support during the workshop organization.

Program Committee

- De-Jiu Chen, KTH Royal Institute of Technology, Sweden
- Arnaud Cuccuru, CEA LIST, France
- Patricia Derler, National Instruments, USA
- Iulia Dragomir, Aalto University, Finland
- Mamoun Filali, CNRS - IRIT, France
- Sébastien Gérard, CEA LIST, France
- Susanne Graf, CNRS - VERIMAG, France
- Gabor Karsai, ISIS, Vanderbilt University, USA
- Alexander Knapp, University of Augsburg, Germany
- Florian Noyrit, CEA LIST, France
- Pierluigi Nuzzo, UC Berkeley, USA
- Ileana Ober, IRIT - University of Toulouse, France
- Iulian Ober, IRIT - University of Toulouse, France
- Necmiye Ozay, University of Michigan, Usa
- Andreas Prinz, University of Agder, Norway
- Alejandra Ruiz Lopez, Tecnalia, Spain
- Bernhard Rumpe, RWTH Aachen, Germany
- Bran Selic, Malina Software, Canada
- Tullio Vardanega, University of Padua, Italy
- Eugenio Villar, University of Cantabria, Spain
- Thomas Weigert, UniqueSoft, USA
- Tim Weilkiens, OOSE Innovative Informatik GmbH, Germany
- Virginie Wiels, LAAS, Toulouse, France
- Qi Zhu, UC Riverside, USA

Steering Committee

- Stefan Van Baelen, iMindsvzw, Belgium
- Mamoun Filali, CNRS-IRIT, France
- Sébastien Gérard, CEA LIST, France
- Ileana Ober, IRIT - University of Toulouse, France
- Bran Selic, Malina Software, Canada
- Thomas Weigert, UniqueSoft, USA

Analytic Dependency Loops in Architectural Models of Cyber-Physical Systems

Ivan Ruchkin, Bradley Schmerl, David Garlan
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{iruchkin, schmerl, garlan}@cs.cmu.edu

***Abstract*—Rigorous engineering of safety-critical Cyber-Physical Systems (CPS) requires integration of heterogeneous modeling methods from different disciplines. It is often necessary to view this integration from the perspective of analyses – algorithms that read and change models. Although such analytic integration supports formal contract-based verification of model evolution, it suffers from the limitation of analytic dependency loops. Dependency loops between analyses cannot be resolved based on existing contract-based verification. This paper makes a step towards using rich architectural descriptions to resolve circular analytic dependencies. We characterize the dependency loop problem and discuss three algorithmic approaches to resolving such loops: analysis iteration, constraint solving, and genetic search. These approaches take advantage of information in multi-view architectures to resolve analytic dependency loops.**

***Keywords*—Analytical models, Component architectures, Embedded software, Systems engineering and theory**

I. Introduction

Cyber-physical systems (CPS), such as self-driving cars and autonomous drones, often operate in critical contexts and therefore require rigorous up-front engineering methods. The model-driven engineering (MDE) community has been developing formal approaches to designing and verifying systems to provide guarantees on performance, safety, and other critical qualities [1] [2]. For example, recent research on collision avoidance proposes various analysis and verification techniques to guarantee an absence of collisions [3] [4] [5].

One important aspect of CPS design is using heterogeneous types of analysis to evaluate and evolve designs. For example, reliability analysis can evolve a design so that its elements have sufficient redundancy [6]; scheduling analysis can allocate computational elements to processors [7]. In reality, there are many analyses that are applied to CPS designs and one particularly challenging aspect is how to integrate, or order and properly apply, these analyses. Such analytic integration can be done by verifying logical conditions in order to control changes made to models [8]. In particular, prior work showed that verification based on *analysis contracts* can prevent errors caused by stale or missing information by determining which analyses must be redone, and in which order. The analytic perspective is convenient when model evolution patterns are simpler than patterns for model structure and behavior.

One of the problems that arises during analytic integration are *analysis dependency loops* – circular dependencies among several analyses that make it impossible to order these analyses in a sound way. Such loops may happen when analyses from different domains are developed independently but operate on the same design aspects, such as sensor infrastructure. For example, reliability analysis may change the number of sensors based on failure probabilities, and trust analysis may adjust the number of sensors to mitigate against malicious attacks on sensors. How can we conduct these analyses and be sure that we have sufficient sensors? Such dependency loops cannot be overcome with previous work using analysis contract specifications [9] and render the methodology inapplicable.

One way to resolve analysis dependency loops is to bring in a more detailed model of the system and analyses, making the dependency description better understood. In previous work we have advocated for the use of component-based (i.e., *architectural*) models to separate engineering into independent components and to assemble the components together [4] [10]. In this paper we take the first step to combining architectural and analytic approaches to CPS design. We take advantage of rich architectural models – multi-view descriptions and component types – to provide several approaches to resolving analytic dependency loops. Specifically, this paper makes the following contributions:

- A characterization of the problem of analytic dependency loops.
- Three algorithmic approaches to resolve dependency loops automatically, and a qualitative analysis of their applicability, strengths, and weaknesses.

Specifically, we examine analysis iteration, constraint solving, and genetic search as potential approaches to resolve dependencies, showing the cases in which each approach may best apply. Iteration and search rely on multi-view mappings to produce valid architectural models, and constraint solving uses a library of architectural types to set up constraint problems on architectural models.

The paper is organized as follows. The next section reviews the related work on CPS modeling and analysis. We then describe and exemplify the problem of circular analysis dependencies in Sec. III. In Sec. IV we propose three approaches to resolve such dependencies and discuss the approaches' qualities. We wrap up the paper by describing future research directions in Sec. V.

II. Related work

Recently several research efforts in architectural modeling have achieved substantial progress in MDE. Results include

composability and provability using component interfaces and contracts [11] [12] [13], rich simulation using multiple computational models [14], platform-based verification and reuse [15] [16], graph-based mapping and consistency between cyber and physical models [17], and semantic validation using logical metamodeling [18] [19]. These methods do not enable reasoning about how designs are modified throughout the engineering lifecycle. At best, there are tools like DESERT [20] for exploring the design space, but these do not support consistent evolution of a set of models. As a result, the integration of heterogeneous CPS models has to be maintained manually, which is tedious and error-prone.

The problem of dependency loops has been considered in many contexts. For example, dataflow systems that consist of concurrent actors may deadlock due to dependency loops among actors [21]. The authors develop a specification approach called *causality interfaces* for actors that helps resolve the loops. However, whether the causality interface approach can be applied to model-based analyses remains an open research question. Another approach is using game-theoretic models to synthesize proof of contract causality [9]. This method relies on detailed game models that may be difficult to obtain for heterogeneous domains where analyses often originate.

Existing research on analyses [22] [8] and change-driven transformations [23] applies formal reasoning at the level of analysis algorithms, which is distinct from architectural models. One aspect of this reasoning is identifying dependencies between analyses and ordering their execution to respect these dependencies. So far this body of work has only considered tree-shaped analysis graphs that do not have cycles [8]. The developed tools, e.g., ACTIVE [24], would not be applicable for circularly dependent analyses, which are more likely to be discovered as more domains are incorporated into the framework of analysis contracts. Our work identifies the potential ways to deal with such circularities.

Several dependency management methods address interactions in different parts of system design. For example, Qamar has developed a cross-domain dependency management approach that keeps track of dependent model variables in their designs across disciplinary and instrumental boundaries [25]. Another example is a multi-view architecture description language with dependency links to ensure consistency among views [26]. Such approaches focus on discovery and representation of dependencies and do not deal with algorithmic cycles directly. Our work therefore can be seen as a next step in automation of change and dependency management.

III. Analytic Dependency Loops

In this section we describe the problem of analytic dependency loops in detail. First we present a car model to ground the discussion and describe two example analyses from the reliability and security domains. Then we formalize several foundational concepts that help us characterize analytic dependency loops.

A. System Example

To illustrate circular dependencies between analyses, let us consider the internal digital system of a self-driving car.

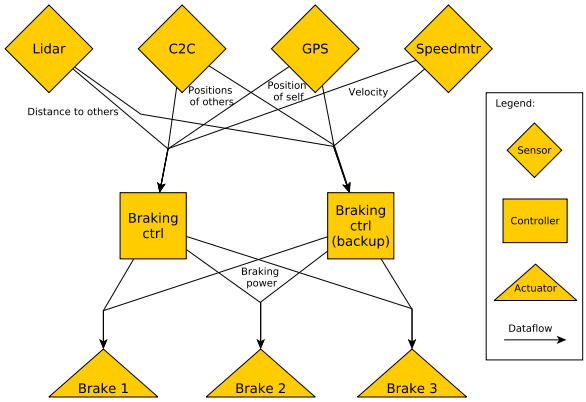

Fig. 1. Architecture of the braking subsystem of a self-driving vehicle.

Inspired by recent advances in the automotive industry [27], the car is designed to perform fully autonomous driving that includes acceleration, lane control and change, platooning, and braking to avoid collision. To inform its decisions, the car collects information about the environment through its sensors: sonar, lidar [28], speedometer, and wireless car-to-car (C2C) communication. Controllers make actuation decisions using algorithms executed in threads running on electronic control units (ECUs), and send these decisions to physical actuators such as steering, acceleration, and braking.

In this example, we focus on the braking subsystem, because it performs the safety-critical function of avoiding collision with various static and dynamic obstacles on the road. In Fig. 1 we show an example architecture of the braking subsystem. Sensors collect data about the position of the car, its speed, and the locations of obstacles. The controllers periodically make a decision about the timing and strength of braking, sending their commands to several braking actuators in the front and back of the car. Since braking control and actuation are critical functions, there is a reserve controller and redundant brakes for the case of nominal components malfunctioning. Throughout this section we will build a formal multi-view model of this architecture in order to precisely express the conditions leading to dependency loops.

One of the major quality attributes of the braking subsystem is safety, which itself depends on system *security* and *reliability*. Security needs to be considered because it may violate safety if a malicious attacker compromises sensors ($\mathbb{S}$). For instance, the braking system could be compromised internally through the CAN network of the car [29] or externally [30] by executing deception attacks on sensors [31]. Different types and placements of sensors (Place) [1] have varying capacity to be compromised by attacks, which determines their level of *trustworthiness* (Trust)[2] [32]. Sensors that output genuine data, or have a mechanism to determine genuine data, are considered trustworthy for modeling purposes. We extend this notion to controllers as well. According to [31], there exists a data decoding algorithm that is guaranteed to deliver genuine data when at least half of the sensors are trustworthy. We

[1] In the running example we consider car sensors placed internally, such as speedometer, and externally, such as car-to-car communication.

[2] For simplicity, we assume that Trust is binary – whether a sensor's output can be trusted.

model security concerns in the trustworthiness view V_{trust} (see the left half of Fig. 2) that contains components $\mathbb{C}_{trust}$ that may be compromised by a malicious attacker – sensors and controllers – and a bus connector $\mathbb{CN}_{trust}$ with data read and write operations.

Reliability of the system should be considered because safety is affected when components randomly fail (e.g., due to manufacturing defects). The reliability view V_{fmea} (see the right half of Fig. 2) contains physical components $\mathbb{C}_{fmea}$ that may fail – sensor devices, threads, electronic control units (ECUs) – and physical network connectors and buses $\mathbb{CN}_{fmea}$. View V_{fmea} focuses on such concerns as component probabilities of failure $\mathsf{P_{fail}}$, failure propagation among individual components, and failure effects. For instance, if the speedometer fails in Fig. 1, the controller will not have an accurate measurement of speed. However, this can be overcome by inferring the approximate speed from values delivered by a position sensor such as GPS. If, on the other hand, both lidar and C2C fail, there is no way for the controller to obtain the locations of obstacles, which is likely to result in a critical failure. Thus, different configurations of system failure (also known as *failure modes* [6]) may have different likelihoods of effects on the system.

The views V_{trust} and V_{fmea} are related to each other through view-to-view mappings of components: $\mathsf{R}^V_V \subset \mathbb{C} \cup \mathbb{CN} \times \mathbb{C} \cup \mathbb{CN}$. Component $c_1 \in \mathbb{C}_{fmea}$ is considered mapped to $c_2 \in \mathbb{C}_{trust}$ when $(c_1, c_2) \in \mathsf{R}^V_V$, and analogously for connectors. Some components such as ECUs in V_{fmea} do not have a counterpart in V_{trust}, so the views are not full abstractions of each other as they are required to be in some approaches (e.g., structural consistency [33]). However, in our example, it is important that sensors and controllers are mapped to each other in both views: every sensor and controller considered for trust needs to be considered for failure, and vice versa. Hence we will use the following condition of consistency:

$$\begin{aligned} &\forall c_1 \in \mathbb{S}_{trust} \cup \mathbb{R}_{trust} \\ &\quad \exists c_2 \in \mathbb{S}_{fmea} \cup \mathbb{R}_{fmea} \cdot (c_1, c_2) \in \mathsf{R}^V_V \qquad (1) \\ &\quad\quad \wedge \\ &\forall c_2 \in \mathbb{S}_{fmea} \cup \mathbb{R}_{fmea} \\ &\quad \exists c_1 \in \mathbb{S}_{trust} \cup \mathbb{R}_{trust} \cdot (c_1, c_2) \in \mathsf{R}^V_V \qquad (2) \end{aligned}$$

where

$$\mathbb{S}_{trust} \cup \mathbb{R}_{trust} \subset \mathbb{C}_{trust} \wedge \mathbb{S}_{fmea} \cup \mathbb{R}_{fmea} \subset \mathbb{C}_{fmea}.$$

The views and relations constitute a full *architectural model* M of the system: $\mathsf{M} \equiv (\mathsf{V}_{trust}, \mathsf{V}_{fmea}, \mathsf{R}^V_V)$. Outside the formal boundaries of M we define component and connector types $\mathbb{T}$ to reuse common aspects of components. Types specify relevant properties such as Trust and $\mathsf{P_{fail}}$, and formally are domains of these functions. For example, a component type could describe a lidar device from a particular supplier and its characteristics. Formally, types are assigned to components and connectors with a typing function $\mathsf{T} : \mathbb{C} \cup \mathbb{CN} \to \{\mathbb{T}\}$ that maps an architectural element to a subset of types. This way we can specify and reuse types separately from systems.

B. Analyses and Contracts

Architectural views undergo continual change as engineers search for a design that satisfies the requirements. Often design exploration and refinement relies upon algorithms and tools, which read and change models. We call such tools *analyses* [22] [8]. Many analyses originate in different domains and make implicit interdependent assumptions about each other. For example, real-time scheduling assumes that there is sufficient electrical power for every processor at all times. At the same time, battery design process requires that computations do not consume more power than the battery can reasonably provide. Such analytic assumptions need to be explicitly considered and reconciled. Let us consider two analyses from the fields of sensor security and system reliability respectively:

- *Trustworthiness Analysis.* A_{trust} [34] modifies the system to ensure that in case of a malicious attack on sensors the system can still function within acceptable error margins. This is achieved by considering a particular attacker profile and determining the necessary number of sensors of each kind. A_{trust} operates over V_{trust}.
- *Failure Modes and Effects Analysis (FMEA).* A_{fmea} [35] determines failure modes and their probabilities. We consider a version of FMEA that redesigns the system so that it does not have critical failure modes (i.e., those where the system is unsafe) with likelihood more than some threshold α_{fail}. A_{fmea} operates over V_{fmea}.

In previous work [36], we considered A_{trust} and A_{fmea} to be integrated without dependency cycles. However, this is not a realistic solution: as more analyses are considered, cyclic dependencies are increasingly likely to occur, and cannot be avoided without significantly changing the analyses. Therefore, in this paper we consider A_{trust} and A_{fmea} to be separate but dependent on each other (as we elaborate later), thus introducing a cyclic dependency that needs to be addressed.

Formally, analysis A is a function that has system designs as its domain and codomain: $\mathsf{A} : \mathsf{M} \to \mathsf{M}$. Many analyses including A_{trust} and A_{fmea} operate only on their specific view V, in which case we can restrict an analysis to this view: $\mathsf{A} : \mathsf{V} \to \mathsf{V}$. In this case executing, or applying, an analysis A to a system model M requires two steps:

1) Obtaining $\mathsf{A}(\mathsf{M})$ and making it the new system under design.
2) Restoring consistency among views in M.

In our example each analysis modifies its own view, which means in step 2 the changes need to be propagated to the other view. This can be done using mapping R^V_V following existing approaches like change propagation [37] or model synchronization [38]. Although re-establishing view consistency is an important part of analysis the workflow, we do not concentrate on it in this paper.

An important assumption of our work is the idempotence property $\mathsf{A}(\mathsf{A}(\mathsf{V})) = \mathsf{A}(\mathsf{V})$. Both analyses that we consider in this paper are idempotent because they directly address a particular quality attribute, and do not modify the system if the attribute is already satisfied. We rely on idempotence in Sec. IV to resolve dependency cycles. Applicability of the discussed resolution techniques to non-idempotent analyses will be considered in the future work.

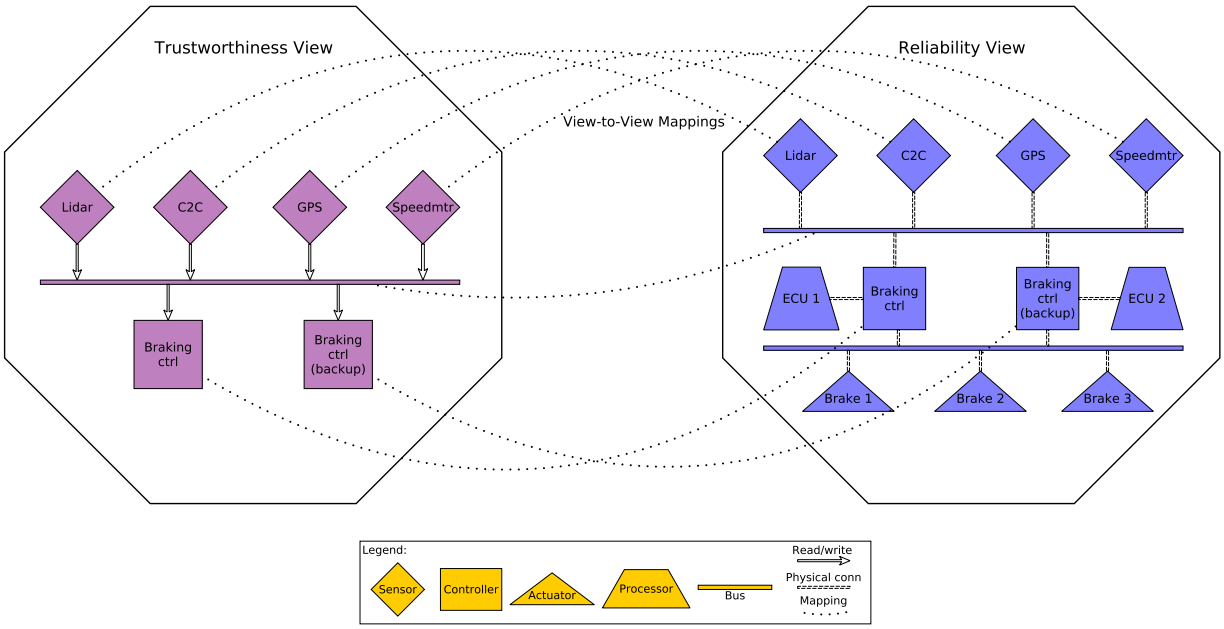

Fig. 2. A multi-view model M of the braking system: V_{trust}, V_{fmea}, and R^V_V.

Following [8], for each A we define analysis contracts as tuples of inputs I, outputs O, assumptions A, and guarantees G: $C_{\mathsf{A}} \equiv (I, O, A, G)$. For simplicity we will write $\mathsf{A}.I$ meaning $C_{\mathsf{A}}.I$. Therefore we have the following contracts[3]:

$$
\begin{aligned}
&\mathsf{C}_{trust}.I = \{\mathbb{S}, \mathsf{Place}, \dots\} \\
&\mathsf{C}_{trust}.O = \{\mathbb{S}, \mathsf{Trust}\} \\
&\mathsf{C}_{trust}.A = \dots \\
&\mathsf{C}_{trust}.G = \text{“system is trustworthy”}\ ^{4} \\
&\\
&\mathsf{C}_{fmea}.I = \{\mathbb{S}, \mathsf{P}_{\mathsf{fail}}, \alpha_{fail}\} \\
&\mathsf{C}_{fmea}.O = \{\mathbb{S}, \dots\} \\
&\mathsf{C}_{fmea}.A = \dots \\
&\mathsf{C}_{fmea}.G = \text{“system is reliable”}\ ^{5}
\end{aligned}
\tag{3}
$$

We limit our discussion in the rest of the paper to A_{trust} and A_{fmea}. However, in a typical engineering context there may be dozens of analyses from heterogeneous domains that have dependencies. For example, control analysis may determine whether a control algorithm satisfies control requirements such as rise time and percent overshoot [39]. Schedulability analyses such as binpacking and frequency scaling [22] determine the capacity behind the control algorithm to compute outputs in time, but at the same time depend upon the quantity of control computation and communication.

[3]Some inputs, outputs, and assumptions are omitted because they do not contribute to the discussion of dependency loops. Full contracts can be found in [36].

[4]System trustworthiness may have several different operationalizations [32]. For example, we could assume that the system is trustworthy when at least half of its sensors are trustworthy [31]. A particular operationalization of trustworthiness is outside of this paper's scope.

[5]Analogously, the interpretation of reliability may differ from system to system. We reason about reliability as a whole without binding ourselves to a particular definition.

According to Eqs. 3 both A_{trust} and A_{fmea} modify the set of system's sensors $\mathbb{S}$, meaning that these two analyses have a circular dependency on each other. This makes it impossible to find a valid sequence of their execution based on just their contracts. Therefore we need to study the nature of analytic dependency loops closer.

C. Dependency Loops

To characterize the dependency loops precisely, let us introduce several formal definitions for analyses, dependencies, and dependency loops between analyses. First, two analyses are dependent if inputs of one have commonalities with outputs of the other.

Definition 1: Analysis A_i *is dependent* on analysis A_j, denoted $d(\mathsf{A}_i, \mathsf{A}_j)$, if $\mathsf{A}_i.I \cap \mathsf{A}_j.O \neq \emptyset$.

Second, a dependency loop is a chain of analysis dependencies where the last element depends on the first one. The smallest dependency loop is a pair of mutually dependent analyses.

Definition 2: Analyses $\mathsf{A}_1 \dots \mathsf{A}_n$ form a *dependency loop*, denoted $\mathsf{Loop}(\mathsf{A}_1 \dots \mathsf{A}_n)$, if:

$$d(\mathsf{A}_1, \mathsf{A}_2) \wedge \dots \wedge d(\mathsf{A}_{n-1}, \mathsf{A}_n) \wedge d(\mathsf{A}_n, \mathsf{A}_1)$$

Eqs. 3 indicate that $\mathsf{Loop}(\mathsf{A}_{fmea}, \mathsf{A}_{trust})$. A dependency loop makes it impossible to use the graph-based ordering algorithm [8] to find a sound sequence of analyses because analysis contracts do not have sufficient specification to resolve the dependency. Therefore we explore other ways to resolve loops. Our ultimate goal is to "skip" the loop and find a design that would theoretically satisfy the loop. Such a design, when fed into each of the analyses, would not change. This situation resembles the fixed point concept from numeric analysis [40], hence we adapt definitions from that field.

Definition 3: A system model or view M is a *fixpoint* of an analysis set $\mathcal{AN}$, denoted $\mathsf{M} \in \mathsf{FP}(\mathcal{AN})$, if $\forall \mathsf{A} \in \mathcal{AN} \cdot \mathsf{A}(\mathsf{M}) = \mathsf{M}$.

From Def. 3 it follows that a fixpoint M satisfies the guarantees of all analyses in $\mathcal{AN}$. This is a necessary, but not sufficient, condition: a model may satisfy all guarantees of an analyses, but not be a fixpoint because the analysis may still modify the model (e.g., to optimize it further). We define models that satisfy all guarantees of an analysis as its *candidate fixpoints*. Also, a fixpoint may not satisfy some assumptions because analyses may exclude their fixpoints from the applicability set since no further changes are possible or needed.

Now consider a set of analyses $\mathcal{AN}$ and a system model M. Below are several mutually exclusive *cases* for fixpoints. These cases support two goals. First, they will help us qualitatively evaluate techniques for dependency loop resolution, which we present in the next section. Second, knowing the case of a particular loop narrows down the available techniques, thus streamlining the resolution of this loop.

- C1 *Strong convergence*: a fixpoint exists and is reachable by any sequence of analyses. This may happen when there are two analyses and their changes to the system do not practically overlap.
- C2 *Weak convergence*: a fixpoint exists and is reachable by some sequence of analyses. This is more likely to be the case when there are several analyses and they interact differently depending on their order of execution.
- C3 *Weak divergence*: a fixpoint exists but is not reachable by any sequence of analyses. E.g., there is a stable alternation between two designs with two analyses.
- C4 *Divergence*: a fixpoint does not exist, but at least one candidate fixpoint exists.
- C5 *Strong divergence*: no candidate fixpoints exist: no model satisfies a conjunction of guarantees of all analyses.

Now that the problem of analytic dependency loops is formally defined, we proceed to the methods of its resolution.

IV. Resolution of Dependency Loops

The goal of dependency loop resolution is, given a system M and a set of circularly dependent analyses $\mathsf{Loop}(\mathcal{AN})$, to find such analysis A′ that would produce a fixpoint of $\mathcal{AN}$:

$$\mathsf{A}'(\mathsf{M}) \in \mathsf{FP}(\mathcal{AN}).$$

This problem has two sub-parts: *finding* a fixpoint and *verifying* that a given model is a fixpoint. For the former, we are not looking for a mathematically optimal solution or a specific fixpoint because system design is often done via *satisficing* [41] rather than optimizing. Many aspects of design are poorly quantifiable: supplier availability and negotiation, diverse qualities of the system, component compatibility, and so on. Therefore we prefer an acceptable suboptimal design to an exhaustive search of a design space that is often unbounded or too large. For the latter part however, we do require an accurate approach, otherwise analysis results may be unsound and potentially lead to design errors.

TABLE I. Convergence, example of C1 and C2.

Sensors	G_{trust}	G_{fmea}
A	✓	✗
B	✗	✗
AB	✓	✗
ABB	✗	✓
AABB	✓	✓

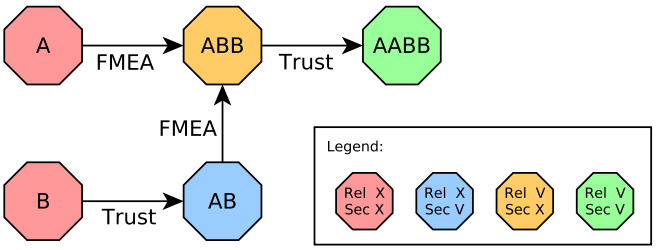

Fig. 3. Example workflow of analyses for convergence.

For further discussion consider the application of A_{fmea} and A_{trust} in several specific contexts. Assume that two types of sensors are given: A and B. A is trustworthy but unreliable, and B is reliable but untrustworthy – these characteristics are specified in the sensors' architectural types. The specific calculations of aggregates do not concern us at this moment, and we abstract reliability and trustworthiness as boolean properties.

First let us consider the convergence situation. Tab. I shows the evaluation of a sensor configuration for the convergence situation. Each line represents a configuration of the system in terms of sensors. AABB is the desired fixpoint configuration that is both trustworthy and reliable. As Fig. 3 indicates, the alternating analyses converge on the fixpoint.

Similarly, Tab. II and III represent divergence with and without a fixpoint respectively. Fig. 4 shows an alternation situation where ABB is not trustworthy and AAB is not reliable, and analyses keep alternating between designs without converging on an existing but unreachable fixpoint AABB.

TABLE II. Divergence, example of C3 and C4.

Sensors	G_{trust}	G_{fmea}
AB	✓	✗
ABB	✗	✓
AAB	✓	✗
AABB	✓	✓

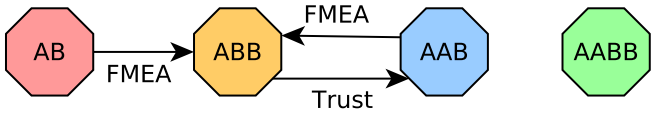

Fig. 4. Example workflow of analyses for divergence. See legend in Fig. 3.

To achieve practical dependency resolution we consider three methods: analysis iteration, constraint solving, and genetic search.

Analysis Iteration. This method iteratively searches for a fixpoint by applying analyses to the model in some sequence, similarly to a method of numeric computation of functional

TABLE III. STRONG DIVERGENCE, EXAMPLE OF C5.

Sensors	G_{trust}	G_{fmea}
A	✓	✗
B	✗	✗
AB	✓	✗
ABB	✗	✓
AAB	✓	✗
AABB	✓	✗
ABBB	✗	✓
AAAB	✓	✗

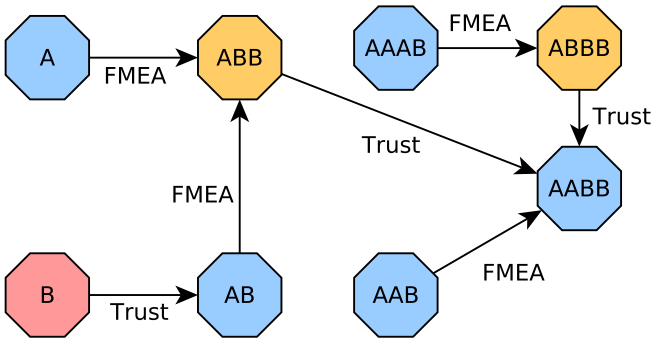

Fig. 5. Example workflow of analyses for strong divergence. See legend in Fig. 3.

fixpoints [40]. In our case, however, the order of analysis iteration is an open question. One option is to select random sequences of analyses, which would find a fixpoint in C1 and could find one in C2. A more sophisticated approach is to construct a contract-guided sequence: only analyses with satisfied assumptions are applied; from those, analyses with unsatisfied guarantees are given a priority. Selection may be random or lexicographical. Another to enhance analysis iteration is to define a partial order on each view, and apply analyses that move the views towards the goal. Analysis iteration can also be used as an accurate fixpoint verification method for C1, C2, C3 and candidate verification for C4 in accordance with Def. 3.

An advantage of analysis iteration is that it is simple and does not require extra specification. In particular, for Tab. I and Fig. 3 iteration would converge on the AABB model given a starting point of A or B. For larger models however iteration is computationally expensive[6], may not converge, and its success may depend heavily on the starting model: there are cases when iteration converges when started from one model but not from another. Therefore, we suggest two other approaches.

Constraint solving. This method searches for a fixpoint by constructing a constraint satisfaction problem and feeding it to a solver. We can use Satisfiability Modulo Theories (SMT) [42] as an example of a constraint solving approach. To set up a constraint problem, one needs to translate relevant architectural types from the model (denoted $SMT(\mathsf{M})$) and analysis guarantees into problem constraints using an existing theory (e.g., integers or reals). The set of sensors under search would become an underspecified part of the satisfaction problem, so that a solver can find its valuation that satisfies constraints. For instance, constraint solving would find AABB in Tab. II/Fig. 4, but analysis iteration would not find a path to it. Constraint solving would also demonstrate absence of a fixpoint in Tab. III/Fig. 5, although it would only explore within the given bounds.

Constraint solving can be successfully used to find a fixpoint in C1, C2, C3, find candidate fixpoints and demonstrate absence of fixpoints in C4, and demonstrate absence of candidate fixpoints in C5 – as long as a constraint problem can be constructed and a (candidate) fixpoint lies within the constraints. The possibility of constructing a constraint satisfaction problem depends on the particular solving framework. For instance, SMT does not yet have theories for calculating real numbers. Unfortunately, constraint solving cannot verify fixpoints because it does not directly execute analyses; nevertheless, it can verify candidate fixpoints. For instance, in the case of SMT if $SMT(\mathsf{M}) \wedge \neg G_1 \wedge \cdots \wedge \neg G_n$ is UNSAT then the M is a candidate fixpoint, at least within the checking bounds. We can overcome the checking bound limitation with the next cycle resolution approach.

Genetic search. This method executes for a system model M obtaining $\mathsf{A}_1(\mathsf{M}) \ldots \mathsf{A}_n(\mathsf{M})$ and deriving hybrids among the architectures, in a way similar to crossover in genetic algorithms [43]. For two analyses A_{fmea} and A_{trust} the set of candidates is $\mathsf{A}_{fmea}(\mathsf{M}) \oplus \mathsf{A}_{trust}(\mathsf{M}) \cup \forall i \subseteq \mathsf{A}_{fmea}(\mathsf{M}) \cap \mathsf{A}_{trust}(\mathsf{M})$, where $\oplus$ is an exclusive OR over sets. Genetic search may find fixpoints in C1, C2, C3, and C4. Genetic search may be particularly useful in cases where the fixpoint is outside the bounds of constraint solving but can be reached by a mutation. For instance, if AABB were outside of the constraint checking bounds in Tab. II/Fig. 4, genetic search would still have a chance to find this model.

A special case of genetic search for the case of two models crossover – *merging models* – can be useful in cases where constraint solving is not: crossover may find a candidate that does not satisfy some guarantees or constraints. Such candidates may provide insights to engineers that would lead to relaxing inappropriate constraints or finding important subspaces of the design space. Merging would rely on view-to-view mappings to achieve consistency in the produced architectural models. We expect merging to be practically limited to sets of components because merging connectors may lead to combinatorial explosion due to non-determinism of where connectors attach. Another drawback of genetic search is that it cannot perform fixpoint verification, and therefore it should be paired with another method like iteration. Thus, non-determinism of genetic search is both its strength and weakness.

All three methods and their expected applicability are summarized in Tab. IV. The constraint solving column assumes that a constraint problem can be formulated in one of the existing theories. This table indicates that no single method can capture all possible dependency cases, and their combination is necessary to provide a robust solution to this engineering problem. We have shown that even for two analyses, A_{trust} and A_{fmea}, the circular case may be different, which would lead to different approaches being fruitful.

This section explored solutions for the relatively simple example of two mutually dependent analyses, A_{trust} and

[6]For practical application of analysis iteration it is crucial that the consistency propagation algorithm is efficient since it is run after every iteration.

TABLE IV. SUMMARY OF APPLICABILITY FOR LOOP RESOLUTION METHODS.

Case	Analysis Iteration	Constraint Solving	Genetic Search
Find C1	✓	✓	✓
Verify C1	✓	✗	✗
Find C2	✗	✓	✓
Verify C2	✓	✗	✗
Find C3	✗	✓	✓
Verify C3	✓	✗	✗
Find C4	✗	✓	✓
Verify C4	✓	✓	✗
Detect C5	✗	✓	✗

A_{fmea}. In a more complex case many analyses depend on each other and make interrelated and often vague assumptions. One may use other ways to deal with this complexity. For example, one may think of re-writing several analyses as one monolithic multi-analysis with algorithms encapsulated. This approach has a benefit of being simple and more controllable (e.g., additional optimization can be applied during consolidation), however it is more fragile because the constituent analyses cannot be reused individually or combined in a different fashion. Instead, our contract-based approach emphasizes more general modular composition, formal verification, and scalability for larger numbers of analyses.

V. FUTURE WORK AND CONCLUSION

This paper explored the challenging problem of resolving analysis dependencies. As we showed, these dependencies often cannot be resolved using contract specification and needs extra description, such as architectural types and mappings. We sketched and exemplified three approaches to cycle resolution: iterative execution, constraint solving, and genetic algorithms. We expect these descriptions and approaches be applicable to other domains and analyses (e.g., cost-benefit analysis of architecture) in our future work.

An important future work direction is implementing and integrating dependency resolution algorithms into our architectural and analytic framework [24]. The tools would need access to architectural styles and analysis descriptions to perform the intended functions. A major step is a design of a general API so that dependency resolution can be extended with new techniques. An implementation of dependency resolution would also be helpful to demonstrate practical feasibility of our cycle resolution techniques. This implementation can be further enhanced in several ways. One is concurrent execution of different techniques and aggregation of their results. Another way to enhance loop resolution is to combine it with optimization and search for optimal fixpoints.

We envision our work to be more general and systematic than ad hoc analysis integration, so empirical validation is essential. We have previously formalized a number of scientific and engineering domains: real-time CPU scheduling, electrical and thermal analysis of batteries [8], reliability, sensor security, and secure control [36]. To demonstrate the generality and effectiveness of the described cycle resolution techniques we will revisit these domains to discover circular dependencies, which we previously designed away. Beyond that, we plan to look for realistic CPS projects to investigate the effect on analytic cycles on larger system designs.

An even deeper level of integration between the analytic and architectural approaches would involve using system invariants during analysis execution. Currently, satisfaction of system invariants is a concern orthogonal to analytic execution. One way to use invariants is to discharge analytic assumptions with them, instead of verifying the assumptions directly. Similarly, one can use analytic guarantees to discharge system invariants after running an analysis. We hope that this would lead to a significant reduction of verification time and effort. We expect that bringing analyses and architecture closer together would lead to a cohesive and versatile toolbox of domain integration tools that can be applied in various engineering contexts, such as aerospace, automotive, energy, and medical CPS.

ACKNOWLEDGMENTS

The authors would like to thank Ashwini Rao, Dionisio De Niz, and Sagar Chaki for their work on the initial vision of the system example and analyses in [36], and Nicholas Rouquette for a motivating discussion of circular dependencies in model-based engineering.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This work was also supported in part by the National Science Foundation under Grant CNS-0834701, and the National Security Agency.

REFERENCES

[1] E. A. Lee, "Cyber Physical Systems: Design Challenges," in *Proceedings of the 11th Symposium on Object Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 363–369.

[2] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *2010 47th ACM/IEEE Design Automation Conference (DAC)*, 2010, pp. 731–736.

[3] S. Mitsch, K. Ghorbal, and A. Platzer, "On Provably Safe Obstacle Avoidance for Autonomous Robotic Ground Vehicles," in *Proc. of Robotics: Science and Systems*, 2013.

[4] I. Ruchkin, B. Schmerl, and D. Garlan, "Architectural Abstractions for Hybrid Programs," in *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*, ser. CBSE '15. New York, NY, USA: ACM, 2015, pp. 65–74.

[5] D. Phan, J. Yang, D. Ratasich, R. Grosu, S. A. Smolka, and S. D. Stoller, "Collision Avoidance for Mobile Robots with Limited Sensing in Unknown Environments," in *Proc. of the 15th International Conference on Runtime Verification*, 2015.

[6] D. H. Stamatis and H. Schneider., *Failure Mode and Effect Analysis: FMEA from Theory to Execution*, 2nd ed. Milwaukee, Wisc: Amer Society for Quality, Jun. 2003.

[7] M. Klein, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Springer, 1993.

[8] I. Ruchkin, D. De Niz, S. Chaki, and D. Garlan, "Contract-based Integration of Cyber-physical Analyses," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14. New York, NY, USA: ACM, 2014, pp. 23:1–23:10.

[9] M. Bartoletti, T. Cimoli, P. Di Giamberardino, and R. Zunino, "Vicious circles in contracts and in logic," *Science of Computer Programming*, vol. 109, pp. 61–95, Oct. 2015.

[10] A. Rajhans, A. Bhave, I. Ruchkin, B. Krogh, D. Garlan, A. Platzer, and B. Schmerl, "Supporting Heterogeneity in Cyber-Physical Systems Architectures," *IEEE Transactions on Automatic Control*, vol. 59, no. 12, pp. 3178–3193, Dec. 2014.

[11] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*," *European Journal of Control*, vol. 18, no. 3, pp. 217–238, 2012.

[12] P. Derler, E. A. Lee, S. Tripakis, and M. Torngren, "Cyber-physical System Design Contracts," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ser. ICCPS '13. New York, NY, USA: ACM, 2013, pp. 109–118.

[13] P. Nuzzo, H. Xu, N. Ozay, J. Finn, A. Sangiovanni-Vincentelli, R. Murray, A. Donze, and S. Seshia, "A Contract-Based Methodology for Aircraft Electric Power System Design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

[14] S. Tripakis, C. Stergiou, C. Shaver, and E. A. Lee, "A Modular Formal Semantics for Ptolemy," *Mathematical Structures in Computer Science. Accepted for publication*, 2012.

[15] V. Subramonian and C. Gill, "Towards Integrated Model-Driven Verification and Empirical Validation of Reusable Software Frameworks for Automotive Systems," in *Model-Driven Development of Reliable Automotive Services*. Springer Berlin Heidelberg, 2008, pp. 118–132.

[16] A. Davare, D. Densmore, L. Guo, R. Passerone, A. L. Sangiovanni-Vincentelli, A. Simalatsar, and Q. Zhu, "metroII: A Design Environment for Cyber-physical Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, pp. 49:1–49:31, 2013.

[17] A. Bhave, "Multi-View Consistency in Architectures for Cyber-Physical Systems," Ph.D. dissertation, Carnegie Mellon University, Dec. 2011.

[18] G. Simko, D. Lindecker, T. Levendovszky, S. Neema, and J. Sztipanovits, "Specification of Cyber-Physical Components with Formal Semantics Integration and Composition," in *Model-Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, Jan. 2013, pp. 471–487.

[19] Sandeep Neema, Ted Bapty, and Janos Sztipanovits, "Multi-Model Language Suite for Cyber-Physical Systems," Institute for Software Integrated Systems, Vanderbilt University, Tech. Rep., 2013.

[20] J. Sztipanovits, G. Karsai, S. Neema, and T. Bapty, "The Model-Integrated Computing Tool Suite," in *Model-Based Engineering of Embedded Real-Time Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Jan. 2010, no. 6100, pp. 369–376.

[21] Y. Zhou and E. A. Lee, "A Causality Interface for Deadlock Analysis in Dataflow," in *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, ser. EMSOFT '06. New York, NY, USA: ACM, 2006, pp. 44–52.

[22] M.-Y. Nam, D. de Niz, L. Wrage, and L. Sha, "Resource allocation contracts for open analytic runtime models," in *Proc. of the 9th International Conference on Embedded Software*, ser. EMSOFT '11. New York, NY, USA: ACM, 2011, pp. 13–22.

[23] G. Bergmann, I. Roth, G. Varro, and D. Varro, "Change-driven model transformations," *Software & Systems Modeling*, vol. 11, no. 3, pp. 431–461, Mar. 2011.

[24] I. Ruchkin, D. De Niz, S. Chaki, and D. Garlan, "ACTIVE: A Tool for Integrating Analysis Contracts," in *5th Analytic Virtual Integration of Cyber-Physical Systems Workshop*, Rome, Italy, Dec. 2014.

[25] A. Qamar, "Model and Dependency Management in Mechatronic Design," Ph.D. dissertation, KTH Sweden, Stockholm, Sweden, 2013.

[26] A. Radjenovic and R. Paige, "The Role of Dependency Links in Ensuring Architectural View Consistency," in *Seventh Working IEEE/IFIP Conference on Software Architecture, 2008. WICSA 2008*, Feb. 2008, pp. 199 –208.

[27] Paul Gao, Russel Hensley, and Andreas Zielke, "A road map to the future for the auto industry," *McKinsey Quarterly*, Oct. 2014.

[28] A. Iliaifar, "LIDAR, lasers, and logic: Anatomy of an autonomous vehicle," 2013. [Online]. Available: http://www.digitaltrends.com/cars

[29] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental Security Analysis of a Modern Automobile," in *2010 IEEE Symposium on Security and Privacy (SP)*, May 2010, pp. 447–462.

[30] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *Proc. of the 20th USENIX Conference on Security*, Berkeley, CA, USA, 2011, pp. 6–22.

[31] H. Fawzi, P. Tabuada, and S. Diggavi, "Secure Estimation and Control for Cyber-Physical Systems Under Adversarial Attacks," *IEEE Transactions on Automatic Control*, vol. 59, no. 6, pp. 1454–1467, Jun. 2014.

[32] F. G. Marmol and G. M. Perez, "Towards pre-standardization of trust and reputation models for distributed and heterogeneous systems," *Computer Standards & Interfaces*, vol. 32, no. 4, pp. 185–196, Jun. 2010.

[33] A. Bhave, B. Krogh, D. Garlan, and B. Schmerl, "View Consistency in Architectures for Cyber-Physical Systems," in *2011 IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS)*, Apr. 2011, pp. 151 –160.

[34] L.-A. Tang, X. Yu, S. Kim, Q. Gu, J. Han, A. Leung, and T. La Porta, "Trustworthiness analysis of sensor data in cyber-physical systems," *Journal of Computer and System Sciences*, vol. 79, no. 3, pp. 383–401, May 2013.

[35] M. Hecht, A. Lam, and C. Vogl, "A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex," in *16th International Conference on Engineering of Complex Computer Systems*, 2011, pp. 361–366.

[36] I. Ruchkin, A. Rao, D. De Niz, S. Chaki, and D. Garlan, "Eliminating Inter-Domain Vulnerabilities in Cyber-Physical Systems: An Analysis Contracts Approach," in *Proc. of the First ACM Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC)*, Denver, Colorado, 2015.

[37] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio, "A model-driven approach to automate the propagation of changes among Architecture Description Languages," *Software & Systems Modeling*, vol. 11, no. 1, pp. 29–53, Jul. 2012.

[38] Z. Diskin, "Algebraic Models for Bidirectional Model Synchronization," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Vlter, Eds. Springer Berlin Heidelberg, 2008, no. 5301, pp. 21–36.

[39] D. W. S. Clair, *Controller Tuning and Control Loop Performance*, 2nd ed. Newark: Straight-Line Control Co., Jan. 1990.

[40] D. Borwein and J. Borwein, "Fixed point iterations for real functions," *Journal of Mathematical Analysis and Applications*, vol. 157, no. 1, pp. 112–126, May 1991.

[41] H. Simon, "Rational choice and the structure of the environment," *Psychological Review*, vol. 63, no. 2, pp. 129–138, 1956.

[42] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an Abstract DavisPutnamLogemannLoveland Procedure to DPLL(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, Nov. 2006.

[43] J. Holland, "Genetic Algorithms and the Optimal Allocation of Trials," *SIAM Journal on Computing*, vol. 2, no. 2, pp. 88–105, Jun. 1973.

Behavioral Types for Space-aware Systems

Jan Olaf Blech
RMIT University
Melbourne, Australia
{janolaf.blech@rmit.edu.au}

Peter Herrmann
Norwegian University of Science and Technology
Trondheim, Norway
{herrmann@item.ntnu.no}

***Abstract*—Behavioral types for space-aware systems are proposed as a means to facilitate the development, commissioning, maintenance, and refactoring of systems with cyber-physical characteristics. In this paper, we particularly introduce the formal definition of behavioral types that are associated with system components in order to specify their expected behavior. As application domain, we concentrate on systems from industrial automation that encompass recurring behavior.**

I. Introduction

In the industrial automation domain, many systems consist of physically distributed components that cooperate with each other by carrying out recurring behavioral patterns. A typical example is a state-of-the-art assembly line consisting of a series of robots that build complex articles like cars. To work correctly, the behaviors of the components need to fulfill various software and physical behavioral aspects that can be quite diverse and may comprise, for instance, communication protocols, heat emission or spatial occupation (e.g., a robot adding a part to a car must perform trajectories such that the car's carriage is not damaged).

To handle the complexity and diversity of specifying component and system behaviors, we introduce *space-aware behavioral types* that allow us to capture both software and physical aspects. As with types in traditional programming languages, e.g., primitive datatypes and their composition, the behavioral types can be used to eliminate error sources already at the development time of software systems. This is analog to classical static type checks performed by a compiler. Furthermore, we can use the behavioral types to eliminate runtime errors. This resembles dynamic type checks that, in many programming languages, are performed when accessing pointers that reference data of types which cannot be statically determined. Behavioral types also provide additional information about components which can be used for tool-based operations that allow the discovery of components and the dynamic reconfiguration of systems.

The behavioral types introduced in this paper are applicable on different scales such as to express the interaction of the various parts of a single robot or to specify collaboration aspects between different sites (cf. [8]). The limitation to recurring behavior makes it possible to verify behavior by checking only a finite number of situations which eases the use of highly automatic verification tools. Our approach makes it possible to check the following features of a type system:

- *Type compatibility checking* — as known from types of imperative programming languages, e.g., checking whether we can add an integer to a float — with space-aware behavioral types associated to components.
- *Subtyping* allows the replacement of a component with a certain behavioral type t by another component that has a subtype t' of type t. We base subtyping on spatial geometric refinement that can be checked automatically.
- *Type composition* is necessary to infer types of components that are composed of existing components with known types.

In addition, we want to ensure

- *Type conformance*, i.e., the question whether a component really behaves according to its specification: the geometric spatial behavioral type.

A. Motivating Examples

Loading robot: Figure 1 shows two pictures of a robot interacting with a moving device. The robot and the device have spatial behaviors, i.e., their positions in space change during time. At various points on the time scale, that we call *timepoints*, they physically occupy certain spaces that can be characterized by coordinates in a geometric coordinate system. On the one hand, we like to ensure using space-aware behavioral types that the robot does not collide with the moving device. On the other one, we also want to guarantee that the robot grip is coming very close to the device in order to avoid that articles are damaged while being loaded onto the device.

The robot consists of three segments and a tool that are attached to each other via joints. Each of the four robot components has an individual spatial behavior relative to the parts it is attached to. As depicted on the left side of Fig. 1, this spatial behavior can be expressed with a space-aware behavioral type that encodes the movement of a robot part over time. Typically, the behavioral description of each type is relative to a distinct point in the coordinate system. For example, multiple instances of the tool may have the same type, but may be deployed independently in different locations (e.g., segments 1 and 3). Likewise, we can use a behavioral type expressing the behavior of the moving device.

The right side of Fig. 1 shows the composition of the types from the robot's components into a single type representing the behavior of the overall robot. The composed type for the robot takes the relative spatial movements of the segment and

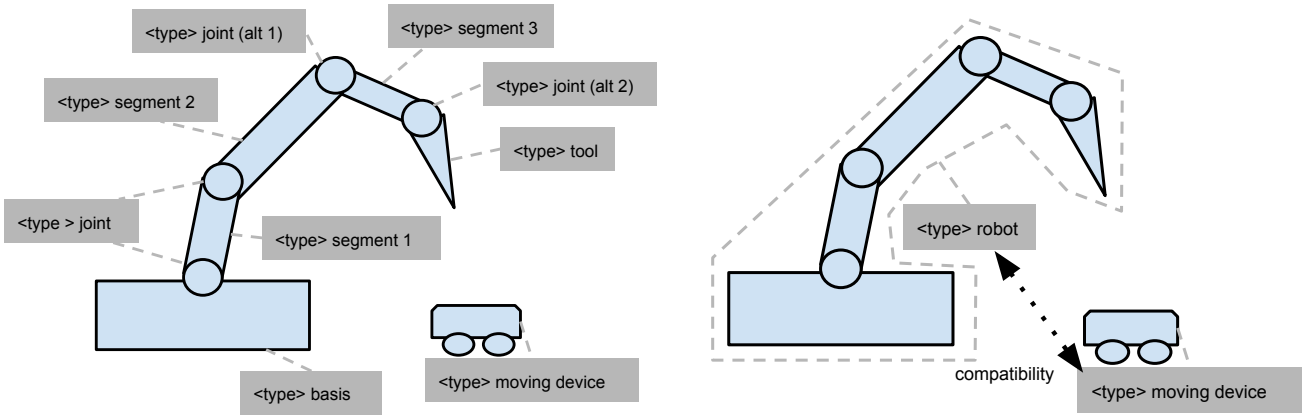

Fig. 1. Behavioral types for a robot

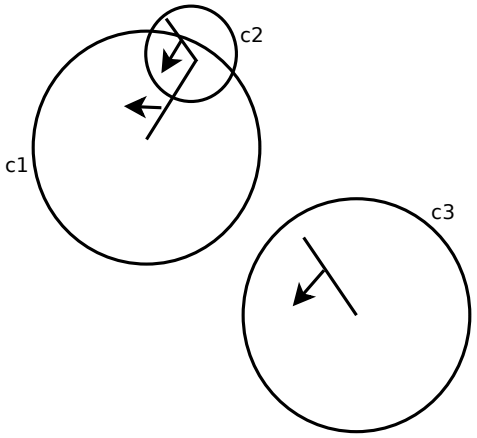

Fig. 2. Spatial behavior of rotating robot arms

tools to each other into account. To verify that the robot does not collide with the moving devices but that its grip comes sufficiently close, we can apply the composed type of the robot instead of the four simple ones referring to its parts. The spatial verifications are carried out by type checking of the composed robot type and the one of the moving device. Using the subtyping feature, we can even replace robot segments by other ones without needing to repeat the type checking proofs of safety properties as long as the replaced segments are in certain relations with the original ones.

Rotating robot arm: Another example of a robot composition is depicted in Figure 2. Here, three components are shown: *c1* is a robot arm. It performs a circular movement around a center point and features a reference point at the outer end that turns counterclockwise. This behavior is captured using a space-aware behavioral type. Another component *c2* also carries out a counterclockwise circular movement albeit with a smaller radius. This is also encoded in a space-aware behavioral type. *c2* gets attached to *c1* via the reference point. By type composition, we can create a behavioral type modeling the joint behavior of *c1* and *c2*.

c3 is also a robot arm, possibly of the same kind as *c1*, that performs the same rotational movement around a different center point. In consequence, the behavioral type of *c3* may be the same as the one of *c1* which, however, refers to another center point.

A typical type checking problem is the decision whether the system composed of *c1* and *c2* can collide with *c3*. For type checking, we compute the least common multiple of the cycle times for each of the three components and compare for each time point whether a collision may occur. The use of time points instead of time intervals requires that the spatial behavior at each time point is a safe approximation of the behavior during the adjacent time intervals. We will discuss this later in detail.

B. Related Work

The idea to use well defined specifications that define the interfaces of software component systems, has been made popular by the design-by-contract approach for software components [31]. More recent work comprises specification and contract languages for component-based systems that have been studied in the context of web services. Process algebra-like languages and associated techniques are studied in [11], [16]. Another algebraic approach to service composition is presented in [18]. In [27], so-called External State Machines (ESMs) are used to specify the interface behavior of functional software building blocks. The ESMs do not only facilitate the integration of the building blocks into their environment but make also compositional model checking of the building blocks possible.

Behavioral types have been studied as interface automata [1] for software components and in the Ptolemy II project [30] for the software part of real-time systems. Further, their use as means for behavioral checks at runtime for component-based systems was investigated in [2].

We proposed a behavioral type system in [9]. In [6], ensuring behavioral type correctness at runtime using techniques from runtime verification was discussed in the context of

Java/OSGi-based applications. Moreover, we studied compatibility checking in [7]. This paper also features a solution for behavioral type coercion for a highly restricted class of behavioral types. Furthermore, we have applied a behavioral types concept to the software part of automation control systems [37] which can be seen as a precursor of the work presented here. Providing a format for spatial behavioral types and means to reason about it is a new contribution of this paper.

Specification of spatial properties has been studied using process algebra-like formalisms [13], [14]. A type system based on this formalism was introduced in [12] for concurrency and resource control. The author presents typing rules and automatic type checking which is not a focus here. Moreover, a verification tool has been developed to check properties based on this formalism in [15]. In contrast, we are interested in developing a solution that fits for industrial robots and related machinery. Therefore, we restrict ourselves to the checking of recurrent behavior in geometric space and concentrate us on tailoring a formalism and compliant checking techniques for this particular domain. Contracts between components with a cyber-physical flavour have been studied in the SPEEDS project [3], [4], [20]. Here, the contracts also take behavior in the form of a transition system into account. In [32] contracts for avionic components are studied.

Reasoning about spatial and geometric constraints is described in, e.g., [5], [25]. A particularly important application domain is robot path planning which has been studied for decades (e.g., [26], [29]). Spatial types are also used for databases, e.g., to manage geometric objects [21] or in Geographic Information Systems [36]. A challenge of these approaches is to guarantee that a reasonable subset of the spatial logic is decidable and, of course, that realistic system models can be checked in an acceptable period of time (e.g., see decidability results in [17]). Logic approaches for hybrid systems (e.g., [19], [34], [35]) provide comprehensive languages and tools for describing cyber-physical systems. In contrast to these works, our focus is stronger aligned with the industrial automation domain and the use as a behavioral type system. The time and geometry focus on the reasoning side of our framework can be complemented by a topological view. This has advantages in areas such as security analysis [33].

As we will show below, the approach presented here fits well to the existing verification technique BeSpaceD [10] that already proved that it can be used to check spatial properties of various systems (see [22], [23], [24]).

C. Overview

Section II introduces our space-aware behavioral types. The underlying semantics and related behavioral types features are discussed in Sect. III. An evaluation is featured in Sect. IV followed by a conclusion in Sect. V.

II. Space-aware Behavioral Types

In general, we describe spatiotemporal behavior for the industrial automation domain by defining which properties hold at which timepoint. Due to the recurrent nature of the behavior, we have to observe only a finite number of timepoints. In Sect. II-A we describe the basic formalism of our behavioral type definitions and introduce certain templates facilitating the use of our method. Thereafter, we discuss the constructors and composition operators in Sect. II-B. In the remainder of this section we justify our modeling choices.

A. Behavioral Descriptions

We use simple logic-based descriptions to define abstract datatypes. These *behavioral descriptions* can be composed of the following operators and predicates:

- *Logical operators:* $\wedge$, $\vee$, and $\neg$ as well as abbreviations such as $\longrightarrow$ and $\bigwedge_{i \in I}$.
- *Predicates that characterize timepoints.* This includes expressions such as timepoints modulo a cycle time — after which the behavior is repeated — and time intervals.
- *Predicates characterizing events.* In addition to the space-aware aspects one can also use events to specify software interaction protocols [9].
- *Predicates indicating nodes and edges in a graph structure.*
- *Predicates indicating occupation of geometric space.*
- *Parameters defining the ownership of space occupation.* Here, spatial occupation behavior is associated with a certain component that *owns* the occupied space.

Our way to associate space occupation with ownership allows us to specify various spatial properties of a component in separation. As already mentioned, examples for such properties that may all refer to the same physical component C, may be C's physical occupation of space, the distribution of heat emitted by C, and the range over which C may broadcast wireless communication messages. These properties can be modeled by separate predicates that all use C as their owner. In consequence, the individual properties can be separately verified by type checking which is carried out based on two different approximation approaches:

- *Overapproximation* means to consider a geometric space that is at least as large as the one that is factually covered by an owner. This fits to properties like the physical occupation of space or the distribution of heat.
- *Underapproximation* refers to a geometric space that is at most as large as the one factually covered. We can use it, for example, to check broadcasting ranges.

The two approaches are closer described in Sect. III.

Templates: Behavioral descriptions encoding a component of the industrial automation / robot domain can follow the templates shown in Fig. 3. The specification features a conjunction over implications. Each implication refers to certain conditions that hold at a certain timepoint and in the presence of events. The conditions can be, for instance, aspects referring to the spatial occupation of a geometric object. Each aspect itself is constructed as predicates of the behavioral description language introduced above. It primarily features constraints on space such as conjunctions of predicates that

$t = 1 \wedge (\neg)$ **event** $E_0 \wedge ... \wedge (\neg)$ event $E_n \longrightarrow$
Space Occupation Aspect $1 \wedge ... \wedge$ Space Occupation Aspect m
$\wedge$
...
$\wedge$
$t = 1 \wedge (\neg)$ event $E_0 \wedge ... \wedge (\neg)$ **event** $E_n \longrightarrow$
Space Occupation Aspect $h \wedge ... \wedge$ Space Occupation Aspect j
$\wedge$
...
$\wedge$
$t = cycletime \wedge (\neg)$ **event** $E_0 \wedge ... \wedge (\neg)$ event $E_n \longrightarrow$
Space Occupation Aspect $1 \wedge ... \wedge$ Space Occupation Aspect m
$\wedge$
...
$\wedge$
$t = cycletime \wedge (\neg)$ event $E_0 \wedge ... \wedge (\neg)$ **event** $E_n \longrightarrow$
Space Occupation Aspect $h \wedge ... \wedge$ Space Occupation Aspect j

Fig. 3. Template for a behavioral description

indicate the occupation of space for a geometric object. A space occupation aspect is either classified as an over- or an underapproximation.

The template specifies spatial behavior up to the timepoint referring to finishing a recurrent behavior cycle. After the cycle time, the behavioral description is repeated. This, however, does not necessarily always result in the same behavior, since events may be different. Having a cycle time is a typical feature in industrial automation and a key characteristic of Programmable Logic Controllers (PLC) used to control automation facilities (e.g., the IEC 61131-3 and IEC 61499 standards) and for controlling industrial robots.

Behavioral descriptions may be specified by developers manually. However, typical descriptions can comprise several thousand cases. Thus, a preferable way is to specify a system in a simulation or development tool and generate the behavioral description automatically. We have successfully done this using the model-based engineering tool Reactive Blocks [28] as described in [23], [24].

B. Type Constructors and Composition

Type constructors use behavioral descriptions and additional information to create a space-aware behavioral type. We present two kinds of space-aware behavioral types. *Primitive space-aware behavioral types* are often used to capture the behavior of a single atomic component, whereas *composed space-aware behavioral types* are typically applied to capture the behavior of composed systems. However, composed types may also be applied to characterize different aspects of a single atomic component and a primitive type may be used to capture the behavior of a composed system, when no detailed behavior of subcomponents is available or it is not necessary to describe that separately.

Basic space-aware behavioral types: We define three different kinds for the primitive behavioral types:

1) A behavioral description bd may be accompanied by the cycle time ct, after which the behavior is repeated to form a geometric spatial behavioral type using the tuple
$$(bd, ct)$$
2) An extended definition features a geometric offset go which is a point in the geometric space. Likewise, space-aware behavioral types allow to shift the starting time of a cycle by a time offset to in order to allow the reuse of the behavioral description for a component that may be started with a delay. The spatial and starting time impacts of the behavioral description can be described by the following tuples:
$$(bd, go, ct),\ (bd, to, ct) \text{ and } (bd, go, to, ct)$$
3) A component that features a behavior in time and space may be attached to a joint device of another component where this joint device has its own spatiotemporal behavior. This *relative movement* of a component to another is captured in the following type definition: A type may feature a set RP of *reference points* through which other components may be attached to it and a behavioral description is provided with each reference point. For instance, the segments and the tool of the robot introduced in Sect. I-A are attached to each other via reference points. Each reference point exhibits its own spatiotemporal behavior that depends on both, the physical placement of the reference point as well as the behavior of the overall component. In the type constructor, we model the relation between reference points and their behavioral descriptions by the function $\mapsto$ mapping all elements of set RP to the set BD of all possible behavioral descriptions. Thus, if $bd_i \in BD$ is the behavioral description of a reference point $rp_i \in RP$, the formula $\mapsto(rp_i) = bd_i$ holds which

we express as $rp_i \mapsto bd_i$ for convenience. The type constructor is defined as follows:

$$(bd, RP, \mapsto, ct)$$

The behavioral description used in the reference point must only describe the movement of a single point in relation to time and events.

Composed space-aware behavioral types: The behavior of multiple components can be combined, e.g., to form new components or to define alternative types. A way to combine behavior types syntactically is *type composition*. Its semantics is highlighted in the following:

1) The union type $+$ provides an alternative between two different space-aware behavioral types gbt and gbt' each defined as one of the three types introduced above:

$$gbt + gbt'$$

As an example, the intended semantics — a behavioral alternative — of a union of two space-aware behavioral types is given below (lcm denotes the *least common multiple*):

$$(bd, ct) + (bd', ct') \triangleq ((bd \vee bd'), lcm(ct, ct'))$$

2) Compositions as expressed by the operator $\times$ correspond to record types in programming languages:

$$gbt \times gbt'$$

Semantically, that corresponds to the following operation on the behavioral description level:

$$(bd, ct) \times (bd', ct') \triangleq ((bd \wedge bd'), lcm(ct, ct'))$$

Furthermore, as in records, we support an implementation that maps names to behavioral descriptions. This allows us to have record-like field descriptors.

3) Composing structures of components attached to reference points, like in the robot example depicted in Fig. 1, usually leads to lengthy nested behavioral descriptions. To simplify these definitions, we offer non-nested type constructors for such structures. The non-nested variant does not have to be attached to a base component such it does not need to feature a cycle time. The simplified constructor can be used if a structure consisting of composed components is modeled by the basic space-aware behavioral type gbt of kind 3, i.e., $gbt \triangleq (bd, RP, \mapsto, ct)$. We also introduce the set GBT that features the geometric spatial behavior in the remainder of the nested structure, as well as the function $\rightarrow$ mapping the reference points $rp_i \in RP$ in the composed structure to their respective behaviors $gbt_i \in GBT$, i.e., $rp_i \rightarrow gbt_i$. The resulting behavioral type is syntactically defined in the following way:

$$(gbt, RP, \rightarrow)$$

To illustrate this, we regard our motivating example from Sect. I-A and Fig. 1. The composed type for the robot is made up of the behavioral type tt of the tool and the types at_1, at_2, at_3 of the three robot arm segments. The four types can be nested in the following way:

$$\begin{aligned}&(at_1, \{rp_{at_1}\},\\ &\ rp_{at_1} \mapsto (at_2, \{rp_{at_2}\},\\ &\qquad rp_{at_2} \mapsto (at_3, \{rp_{at_3}\}, rp_{at_3} \mapsto (tt))\\ &\qquad)\\ &)\end{aligned}$$

Using our introduced definition, the behavioral type of each segment type at_i with a behavioral description ab_i has the form:

$$(ab_i, \{rp_{at_i}\}, rp_{at_i} \rightarrow rpb_{rp_{at_i}})$$

where rpb_{rp_k} refers to the behavior of a reference point rp_k thereby removing the nested structure.

Our notion of behavioral types takes the intended semantics into account, i.e., the behavior in space and time. Different *syntactic type definitions* which may be grouped into equivalence classes may exist for the same space-aware behavioral type. For instance, by using the symmetry of the union operator in type composition or the symmetry of $\wedge$, we can construct syntactically different type definitions for the same type.

III. Semantics of Space-aware Behavioral Types

To facilitate the verification that objects occupy a certain geometric space in an area, we can use subtyping of the behavioral types of these objects. As described in Sect. II-A, verification of spatial properties can be performed based on both, overapproximation and underapproximation. This is considered by distinguishing subtyping between *overapproximation-refinement aspects* and *underapproximation-refinement aspects*. A space-aware behavioral type T' is a subtype of another type T if and only if the following conditions hold for each spatial aspect and each shared timepoint t:

- For overapproximation-refinement aspects, the space occupation at t specified in T' is geometrically included in T. Thus, overapproximation-oriented spatial proofs (e.g., collision avoidance) that were carried out for a physical component represented by T also hold for a "smaller" one described by T'.
- For underapproximation-refinement aspects, the space occupation at t specified in T is geometrically included in T'. So, underapproximation proofs (e.g., broadcast ranges) done for T hold also for a "larger" T'.
- For both, overapproximation-refinement and underapproximation-refinement aspects hold, that if T comprises unbound reference points, T' incorporates the same unbound reference points, which show an identical behavior.

Subtyping imposes a partial order relation between the space-aware behavioral types since according to our definition the following properties hold:

- *Reflexivity:* A type is its own subtype since an occupied space includes itself.
- *Antisymmetry:* For aspects refined by overapproximation holds that if the space occupied according to T' is

geometrically included in the one of T but not identical, then there is at least a point in space that is occupied by T but not by T'. Thus, the space of T is not included in the one of T' and, in consequence, T is not a subtype of T' with respect to overapproximation. The argumentation for underapproximation is analog.

- *Transitivity:* If T' is a subtype of T and T'' a subtype of T' with respect to overapproximation, then the occupied space according to T'' is included in the one defined by T' and that one is included in the one according to T. Thus, the occupied space defined for T'' is also included in the one specified in T such that T'' is also a subtype of T. An analogous deduction can be drawn for underapproximation.

It is possible to construct a lattice based on this partial order for a fixed number of aspects. The type $\bot$ is a subtype of all other types. Here, all overapproximation-refinement aspects are occupying zero space all the time, while underapproximation-refinement aspects occupy all the space all the time. In contrast, all other types are subtypes of the $\top$ element. Thus, underapproximation-refinement aspects occupy zero space all the time, while overapproximation-refinement aspects occupy all the space all the time.

IV. Behavioral Type Checking and Evaluation

In this section, we discuss means to decide the compatibility of system components based on their behavioral types.

A. Type Compatibility Checking Algorithm

For two space-aware behavioral types with cycle times ct_1 and ct_2, we perform space-aware behavioral type checking in the following way:

1) We calculate the least common multiple of ct_1 and ct_2 that we name ct.
2) For all time points t between 0 and ct we perform the following steps:
   a) Retrieve for both behavioral types all relevant spatial information expressed by the behavioral descriptions bd_1 and bd_2 at timepoint t.
   b) Decide possible overlappings between the behavioral descriptions bd_1 and bd_2 by regarding the possibly occupied space for all underapproximation-refinement aspects. Here, an overlapping must occur, for each spatial aspect. Otherwise, the types are incompatible.
   c) Decide additional possible overlapping between spatial information of bd_1 and bd_2 by regarding the possibly occupied space for all overapproximation-refinement aspects. Here, no overlapping must occur for any spatial aspect. Otherwise, the types are incompatible.

The algorithm is carried out using the checker BeSpaceD [10] that, depending on the geometry used, converts the spatial information and property into a SAT or an SMT problem. For that, BeSpaceD breaks the geometric constraints down into more fine grained verification conditions as we discuss in the following.

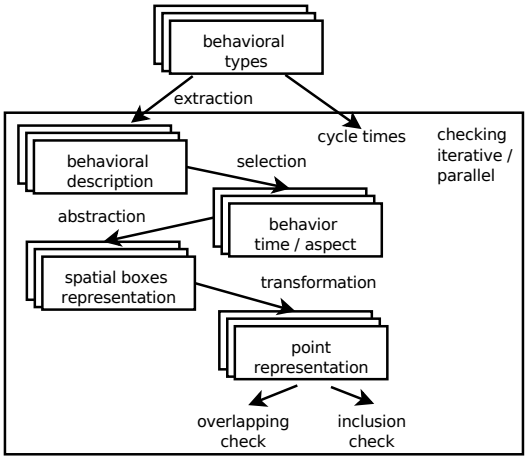

Fig. 4. Checking type compatibility and subtyping

B. Making Behavioral Descriptions Checkable

Our modeling style allows for very rich specifications describing quite complex systems. Checking these specifications would demand to treat a state space that would exceed the time and memory limits of the type checking algorithm introduced above. In the following, we present some steps allowing to abstract complex specifications into checkable ones such that our type compatibility checking and subtyping algorithms can be used. To guarantee that the abstractions do not falsify the verification results, they have to preserve the transitivity, reflexivity, and antisymmetry properties introduced in Sect. III. The abstraction consists of an order of operations that is depicted in Figure 4 (see also [10]):

1) *From time intervals to timepoints:* Time interval-based descriptions are transformed into timepoint-based descriptions by using safe approximations of geometric spatial behavior of adjacent time intervals at the timepoints.
2) *Extraction of relevant behavioral information:* BeSpaceD provides functions that are based on time and spatial aspects and provide sub-descriptions for the relevant behavior which are defined on the inductive structure of the behavioral descriptions.
3) *From segments to boxes:* Parts of robots may be described by segments or other geometric objects. Segments have a cylindric shape with a radius, a length, and an orientation. For fast and easy checking, we convert segments and other geometric objects into box-based approximations. Boxes are defined by an upper left front and a lower right rear coordinate that are both expressed by their respective x, y and z axes of the coordinate system. Figure 5 shows a variant of the second example from Sect. I-A in which the line representations of the three robot components are replaced by a number of boxes representing the space covered. As long as the

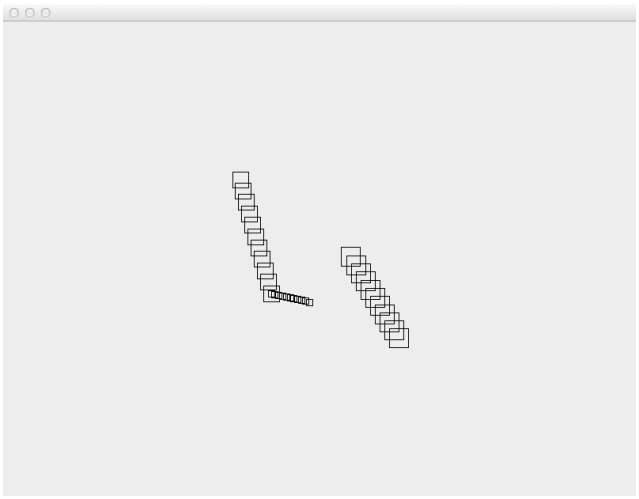

Fig. 5. Box-based abstraction of rotating robots

boxes cover all the space of the three components, this replacement is a safe overapproximation. (It would be a safe underapproximation if all space represented by the boxes was covered by the components.)

4) *Automata and spatial behavior:* The behavior of our components can be modeled using automata with a cyclic control flow. Here, we describe possible transitions and states encountered as events that are part of the behavioral description.
5) *From boxes to spacepoints:* Behavioral descriptions using geometric boxes can be broken down into descriptions that contain geometric points, so-called *spacepoints*. For example, a cube with a side length of 10 may be broken down into $10 \cdot 10 \cdot 10 = 1000$ spacepoints. In the behavioral description, each spacepoint is described using a predicate. In spite of this enlargement of the behavioral representation, we can check the spacepoints speedily since points from different behavioral descriptions are comparable without further interpretation.
6) *Checking of overlappings and inclusion with points:* We use hash-sets for checking overlappings and inclusion of two descriptions. For overlappings, we insert points from one description into the hash-set and check whether the points of the second description are already in the hash-set. For inclusion, we insert points from one description and check whether all points from the other description are indeed included in the hash-set.
7) *SMT and other approaches:* In addition to comparing geometric representations on a point level, we have developed SMT encodings of geometric objects that are more efficient for large sets of points [10]. Furthermore, checking of point-wise overlappings and inclusion can also be performed in BeSpaceD using a SAT solver.

C. Implementation

A first implementation of BeSpaceD and space-aware behavioral types exists. It is done in the functional programming language *Scala* which facilitates the break down and conversion of behavioral descriptions.

```
abstract class Invariant;

abstract class ATOM extends Invariant;

case class OR (t1 : Invariant, t2 : Invariant)
   extends Invariant;
case class AND (t1 : Invariant, t2 : Invariant)
   extends Invariant;
case class NOT (t : Invariant) extends Invariant;
case class IMPLIES (t1 : Invariant, t2 : Invariant)
   extends Invariant;
        ...
case class TimePoint [T](timepoint : T)
   extends ATOM;
case class TimeInterval [T]
 (timepoint1 : T, timepoint2 : T) extends ATOM;
case class Event[E] (event : E) extends ATOM;
        ...
case class Occupy3DBox
  (x1 : Int, y1: Int, z1 : Int,
   x2 : Int, y2 : Int, z2 : Int) extends ATOM;
case class OccupySegment3D
  (x1 : Int, y1 : Int, z1 : Int,
   x2 : Int, y2 :Int, z2 : Int, radius : Int)
   extends ATOM;
case class Occupy3DPoint (x:Int, y:Int, z: Int)
   extends ATOM
```

Fig. 6. Some Scala definitions

Behavioral descriptions are provided as abstract data types called `Invariant`. We chose this name since logical descriptions are supposed to capture the abstract behavior of a component during its entire lifetime. For look and feel, we provide an excerpt in Fig. 6. Some logical operators, predicates for time and events and geometric occupation of time are shown. The description language is more expressive than the subset used for space-aware behavioral types, e.g., time only needs to be a type with a partial order (parameter `T`) whereas in our semantics definitions above we used integers.

In the following, we discuss two features of the implementation:

Type system features: Using the type constructors above with the behavioral specifications, our type checking algorithm invoking the BeSpaceD tool allows us to check *(i)* space-aware behavioral type compatibility and *(ii)* whether a space-aware behavioral type is a subtype of another one. Note, that behavioral descriptions can look different, but may describe the same type. Our framework is able to decide both subtyping and type compatibility, since we exhaustively simulate possible behavior bounded by the a cycle time. In cases, where the behavioral descriptions use elements that we cannot check, we may still derive an order of types based on checkable spatial aspects. For all non-checkable aspects, we assume safe approximations. Hence, a type for which the behavioral specification is uncheckable for all aspects, is equivalent to $\bot$.

Speed of type checking: We implemented the space-aware behavioral types checking as described above. Checking can be done in acceptable time, e.g., checking two types with a cycle time of 1000 different timepoints and 15000 spacepoints

for the first resp. 20000 spacepoints for the second behavioral description was done in between seven and eight seconds on an Intel core i5 running 2.8 GHz with 8 GB RAM using Mac OS 10.8.4.

V. Conclusion

We presented behavioral types as a concept for space-aware systems facilitating the development, commissioning, maintenance, and refactoring of systems with cyber-physical characteristics. Using a robot system, we motivated, formally defined and discussed their applicability.

The approach is intended to be used in industrial automation. Facilities in the domain typically operate using cycles, after which behavior is repeated. For example, a robot in an assembly line may perform the same movement and operation on a workpiece over and over again with slight variations based on the color of a work piece. Our behavioral descriptions were designed with that kind of behavior in mind.

Moreover, we believe that the use of behavioral type-like specifications of cyber-physical systems is especially important for remote collaboration of engineering teams. Ongoing work in this direction comprises our collaborative engineering project [8] with a focus on remote handling of industrial installations in the Australian outback (such as mining sites) or for oil rigs.

References

[1] L. de Alfaro, T.A. Henzinger. Interface automata. Symposium on Foundations of Software Engineering, ACM , 2001.

[2] F. Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. Formal Methods for Components and Objects. vol. 2852 of LNCS, Springer-Verlag, 2003.

[3] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, Multiple viewpoint contract-based specification and design. Formal Methods for Components and Objects. Springer-Verlag, 2008.

[4] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier et al.. Contracts for System Design, INRIA, Rapport de recherche RR-8147, Nov. 2012.

[5] B. Bennett, A. G. Cohn, F. Wolter, M. Zakharyaschev. Multi-Dimensional Modal Logic as a Framework for Spatio-Temporal Reasoning. Applied Intelligence, Volume 17, Issue 3, Kluwer Academic Publishers, November 2002.

[6] J. O. Blech. Ensuring OSGi Component Based Properties at Runtime with Behavioral Types. Model-Driven Engineering, Verification, and Validation, 2013.

[7] J. O. Blech. Towards a Framework for Behavioral Specifications of OSGi Components. Formal Engineering approaches to Software Components and Architectures. Electronic Proceedings in Theoretical Computer Science, 2013.

[8] J. O. Blech, I. Peake, H. Schmidt, M. Kande, S. Ramaswamy, Sudarsan SD., and V. Narayanan. Collaborative Engineering through Integration of Architectural, Social and Spatial Models. *Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2014.

[9] J. O. Blech and B. Schätz. Towards a Formal Foundation of Behavioral Types for UML State-Machines. UML and Formal Methods. Paris, France, ACM SIGSOFT Software Engineering Notes, August 2012.

[10] J. O. Blech and H. Schmidt. Towards Modeling and Checking the Spatial and Interaction Behavior of Widely Distributed Systems. Improving Systems and Software Engineering Conference, Melbourne, 2013.

[11] M. Bravetti, G. Zavattaro. A theory of contracts for strong service compliance. Mathematical Structures in Computer Science 19(3): 601–638, 2009.

[12] L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. Theoretical Computer Science, Elsevier, 2008.

[13] L. Caires and L. Cardelli.A Spatial Logic for Concurrency (Part I). Information and Computation, Vol 186/2 November 2003.

[14] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). Theoretical Computer Science, 322(3) pp. 517–565, September 2004.

[15] L. Caires and H. Torres Vieira. SLMC: a tool for model checking concurrent systems against dynamical spatial logic specifications. Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2012.

[16] G. Castagna, N. Gesbert, L. Padovani. A theory of contracts for Web services. ACM Trans. Program. Lang. Syst. 31(5), 2009.

[17] S. Dal Zilio, D. Lugiez, C. Meyssonnier. A logic you can count on. Symposium on Principles of programming languages, ACM, 2004.

[18] J. L. Fiadeiro, A. Lopes. Consistency of Service Composition. Fundamental Approaches to Software Engineering (FASE), vol. 7212 of LNCS, Springer, 2012.

[19] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, O. Maler. SpaceEx: Scalable Verification of Hybrid Systems. Computer Aided Verification (CAV'11), 2011.

[20] S. Graf, R. Passerone, and S. Quinton. Contract-based reasoning for component systems with rich interactions. Embedded Systems Development, ser. Embedded Systems, vol. 20, pp. 139-154, Springer, 2014.

[21] R.H. Güting, R. Hartmut, and M. Schneider. Realm-based spatial data types: the ROSE algebra. The VLDB JournalThe International Journal on Very Large Data Bases 4.2 (1995): 243–286.

[22] F. Han, J. O. Blech, P. Herrmann, and H. Schmidt. Model-based Engineering and Analysis of Space-aware Systems Communicating via IEEE 802.11. In 39th Annual International Computers, Software & Applications Conference (COMPSAC), pages 638–646, IEEE Computer, 2015.

[23] F. Han, J. O. Blech, P. Herrmann, H. Schmidt. Towards Verifying Safety Properties of Real-Time Probabilistic Systems. Formal Engineering approaches to Software Components and Architectures, 2014.

[24] P. Herrmann, J.O. Blech, F. Han, H. Schmidt. A Model-based Toolchain to Verify Spatial Behavior of Cyber-Physical Systems. In 2014 Asia-Pacific Services Computing Conference (APSCC), IEEE Computer.

[25] D. Hirschkoff, É. Lozes, D. Sangiorgi. Minimality Results for the Spatial Logics. Foundations of Software Technology and Theoretical Computer Science, vol 2914 of LNCS, Springer, 2003.

[26] S. Kambhampati and L.S. Davis. Multiresolution path planning for mobile robots. Volume 2 , Issue: 3, Journal of Robotics and Automation, IEEE 1986.

[27] F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In *Model Driven Engineering Languages and Systems (MoDELS)*, LNCS 5795, pages 571–585. Springer-Verlag, 2009.

[28] F. A. Kraemer, V. Slåtten and P. Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. *Journal of Systems and Software*, 82(12):2068–2080, 2009.

[29] J.-C. Latombe. Robot Motion Planning. Kluwer Academic Publishers, 1991.

[30] E.A. Lee, Y. Xiong. A behavioral type system and its application in ptolemy ii. Formal Aspects of Computing, 2004.

[31] B. Meyer. Applying "Design by Contract". Computer, 25, 10, pp. 40–51, IEEE, October 1992.

[32] P. Nuzzo, H. Xu, N. Ozay, J. Finn, A. Sangiovanni-Vincentelli, R. Murray, A. Donz, and S. Seshia, "A contract-based methodology for aircraft electric power system design," IEEE Access, vol. 2, pp. 1-25, 2014.

[33] L. Pasquale, C. Ghezzi, C. Menghi, Ch. Tsigkanos, and B. Nuseibeh. Topology aware adaptive security. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 43–48. ACM, 2014.

[34] A. Platzer. Differential dynamic logic for hybrid systems. Journal of Automated Reasoning, vol. 41.2: 143–189, Springer, 2008.

[35] A. Platzer, J.-D. Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). International Joint Conference on Automated Reasoning, pp. 171–178, LNCS 5195, Springer, 2008.

[36] P. Rigaux, M. Scholl, and Agnes Voisard. Spatial databases: with application to GIS. Morgan Kaufmann, 2001.

[37] M. Wenger, J. O. Blech and A. Zoitl. Behavioral Type-based Monitoring for IEC 61499. To appear in Emerging Technologies and Factory Automation (ETFA), IEEE, 2015.

AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems

Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, Bernhard Schätz
fortiss GmbH
Guerickestr. 25
80805 Munich, Germany
Email: hoelzl@fortiss.org

***Abstract*—This paper presents tooling concepts in AUTOFOCUS 3 supporting the development of software-intensive embedded system design. AUTOFOCUS 3 is a highly integrated model-based tool covering the complete development process from requirements elicitation, deployment, the modelling of the hardware platform to code generation. This is achieved thanks to precise static *and* dynamic semantics based on the FOCUS theory [1]. Models are used for requirements, for the software architecture (SA), for the hardware platform *and for relations* between those different viewpoints: traces from requirements to the SA, refinements between SAs, and deployments of the SA to the platform. This holistic usage of models allows the provision of a wide range of analysis and synthesis techniques such as testing, model checking and deployment and scheduling generation.**

In this paper, we demonstrate how tooling concepts on different steps in the development process look like, based on these integrated models and implemented in AUTOFOCUS 3.

***Index Terms*—AutoFOCUS3, Seamless MBD, Model-Based Development, Embedded Systems, Tooling Concept, Tool**

I. INTRODUCTION

Embedded systems are increasingly developed in a model-based fashion facilitating constructive and analytic quality assurance via abstract component-models of the system under development. A variety of different tools claim to support a model-based approach; however, these tools mostly cover certain parts of the development process. In this paper, we demonstrate the advantage of integrated models and provide tooling concepts for various design steps. Both together leverage the benefits of a seamless model-based approach based on well-defined semantics. The objective of this paper is to present such tooling concepts in an entire tool (AUTOFOCUS 3) in its current advanced feature state, which is open source and freely available at `http://af3.fortiss.org`. The main contribution of this paper is to demonstrate the seamless integration of the integrated models.

AUTOFOCUS 3 is built on a *system model* based on the FOCUS theory [1] that allows to precisely describe a system, its interface, behavior, and decomposition in component systems on different levels of abstraction. To manage the complexity of describing such a system, different *views* are used for these aspects, providing dedicated description techniques for different structural aspects (like the decomposition of a component in a network of subcomponents, or the hardware platform the system is implemented on) as well as behavioral aspects (like an exemplary execution of a system, or its complete behavior). Since all of these views are projections of the underlying system model, these views are directly integrated. Furthermore, linking views allow an additional integration (like the mapping of a component behavior to a hardware element, or a required partial execution to a completely defined behavior). Besides allowing a manageable precise description of a system under development, the system model also enables different analysis and synthesis mechanisms (like the compatibility analysis of a partial and complete behavior or the deployment synthesis of components to a hardware platform). To support the different tasks in a development process, the views are furthermore organized in *viewpoints*. A viewpoint serves as a construct for managing the artifacts related to the different stakeholders of the development process [2, Chapter 3]. The AUTOFOCUS 3 viewpoints focus on the definition of the system requirements in a requirements analysis, the design of the software as a network of communicating components in form of a software architecture, and the realization of the system as scheduled tasks executed on networked processors in form of a hardware architecture.

The importance of integrated *models*, *views* and *viewpoints* is widely recognized and influenced the definition of many methods and frameworks in systems, software and enterprise engineering [2], [3] and provides the basis for this paper.

The objective of AUTOFOCUS 3 is to implement *tooling concepts* which demonstrate that such an approach is indeed feasible through a ready-to-use, open-source implementation in a pre-series quality. The current AUTOFOCUS 3 is a revised and improved version of earlier prototypes [4], [5] (the oldest dating back to 1996). Previous papers either report on particular aspects of the tool [6], [7], [8], [9], or on its use in the context of industrial case studies [10], [11], [12]. The underlying ideas of the current AUTOFOCUS 3 incarnation are presented in [13].

AUTOFOCUS 3 is not tied to a specific development process, but most developments done with AUTOFOCUS 3 would typically follow the following process or variations thereof:

1) **Requirements Analysis.** Requirements are elicited, documented as structured text, organized, analyzed and refined, and incrementally formalized. Test suites with coverage criteria can be generated from high-level specifications.

2) **Software Architecture.** The system is designed with a component-based language specifying the application software architecture and behavior of the system. The design is validated using *simulation*, *testing* (which can come as refinements from high-level generated tests) as well as *formal verification* based on model-checking.
3) **Hardware Architecture.** The software components are (possibly automatically) deployed on the platform, w.r.t. certain system requirements. Schedules optimizing one or more criteria are generated with the help of *SMT solvers.*

The code is *completely* generated out of the previous models, according to the deployment and chosen schedule. Furthermore, **Safety Cases** [14], which are documented bodies of evidence that provide a convincing and valid argument that a system is adequately safe for a given application in a given environment, can be modelled. A Safety Case may contain complex arguments that can be decomposed, corresponding to modular system artifacts which are generally *dependent on artifacts from different viewpoints*.

The paper is organized as follows: Section II presents briefly the main modelling viewpoints that are offered by AUTOFOCUS 3 (requirements, software architecture, and hardware architecture). Section III presents the transversal viewpoints which facilitate to make the connections between the main viewpoints and thus yield a seamless integration. We also present the benefits resulting of this integration: formal analysis and verification, scheduling, hardware-specific code generation. Section IV presents related work.

II. Model-based Tooling Concepts in the Development Process

In this section we present shortly the three modelling *viewpoints* supporting the process mentioned in the previous section.

A. Requirements

In AUTOFOCUS 3, requirements are specified model-based: requirements are not just documented as plain text; the tool provides templates with named fields to define, for instance, the title of a requirement, its author, a description, a potential rationale or a review status (see Fig. 1).

Furthermore, requirement sources and glossaries can be defined. Whenever they are referenced in a textual description of a requirement, the entries are automatically highlighted and the definition can be read in a pop-up. Requirements can be hierarchically grouped by packages and organized by trace links. Templates for scenarios and interface behavior help to detail requirements further.

Requirements can not only be documented as text, but also formalized and represented by machine-processable models. Message sequence charts (MSC), see Fig. 2, can be used to describe desired or unwanted interactions of actors.

Temporal logic expressions can be used to express desired and unwanted behavior of the system under development.

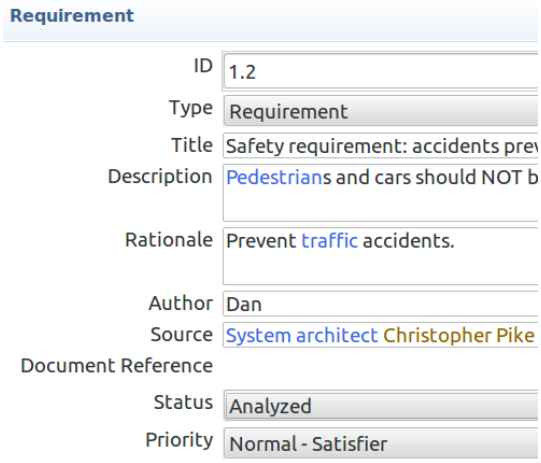

Fig. 1. Example structured requirement

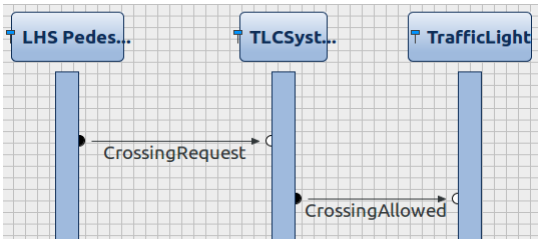

Fig. 2. Message sequence charts

As shown in Fig. 3, AUTOFOCUS 3 provides user-friendly templates (see also [15]) to specify temporal logic expressions.

Fig. 3. Temporal logic expression

Once the requirements analysis is sufficiently advanced, it is possible to express formalized behaviors, e.g., in the form of state automata. A state automaton typically covers a set of requirements rather than a single requirement.

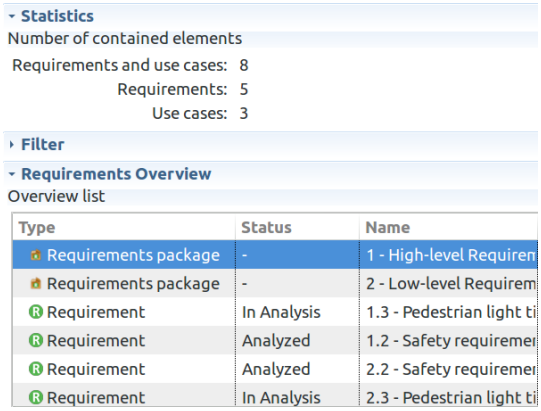

Fig. 4. Requirements statistics and reports

Tooling Support. AUTOFOCUS 3 supports the user in *analyzing* the requirements, for example through reports on the review status or statistics (Fig. 4). Simple queries on the requirements identify for example empty fields, duplicates and inconsistent status of requirements and their trace links. A report can be generated from AUTOFOCUS 3 that can be used for the *validation* of the requirements by the stakeholders of the system under development. State automata can be simulated; this is typically used in both the analysis and validation of requirements.

B. Software Architecture

The software architecture of a system under development can be described using a classical component-based language with a formal (execution) semantics based on the FOCUS theory [1]: components execute in parallel according to a global, synchronous, and discrete time clock.

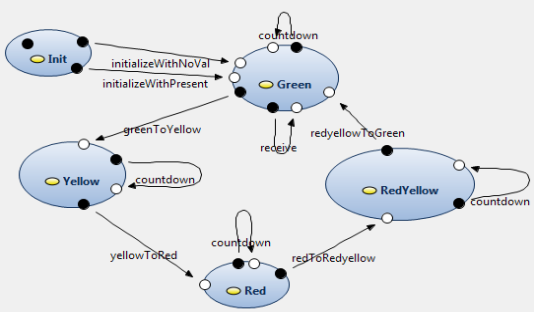

Fig. 5. State automaton

The behavior of atomic components can be defined by state automata (Fig. 5), tables (Fig. 6) or simple imperative code (Fig. 7). Components interact with each other through typed input and output ports which are connected by fixed channels (Fig. 8).

	Input					Output		
	mergeInButtonA	-	mergeInButtonB	-	true	-	+	ctrlInRequest	-	+
-	Present()		*		mergeInB...resent()			Pressed()		
-	*		Present()		mergeInB...resent()			Pressed()		
-	Present()		Present()		*			Pressed()		
-	NoVal		NoVal		*			Released( )		
+	Click to add a new rule...									

Fig. 6. Table

```
if (behaviorInState == 0) {
    behaviorOutTrafficSignal = Green( );
    behaviorOutPedestrianSignal = Stop( );
    behaviorOutIndicatorSignal = Off( );
    behaviorOutTime = -1;
    behaviorOutState = 1;
    return;
}
```

Fig. 7. Simple imperative code

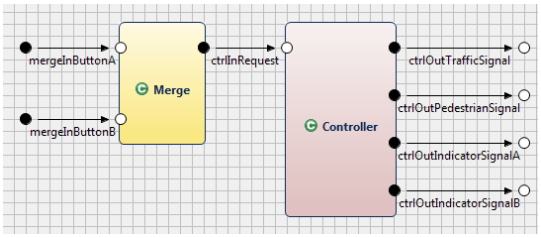

Fig. 8. Components and channels

Finally, components can be decomposed into sub-components to allow a hierarchically structured architecture.

Tooling Support. Due to the executable formal semantics of the component-based modeling language, AUTOFOCUS 3 facilitates the simulation of the software architecture at all levels, of a single state automaton (Fig. 9) as well as of composite components (Fig. 10) providing Model-in-the-Loop simulations. Test cases can be created and simulated (Fig. 11).

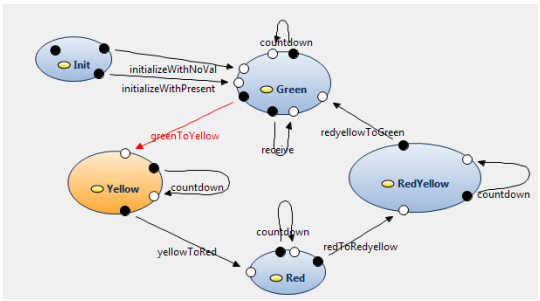

Fig. 9. Simulation of state automata

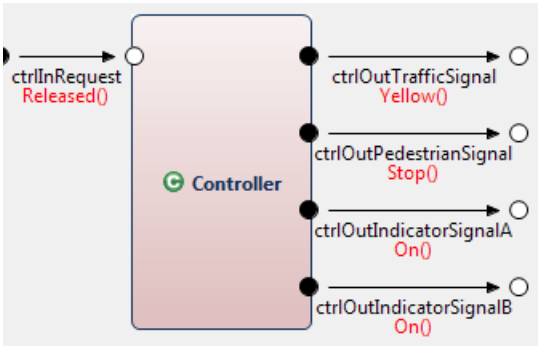

Fig. 10. Simulation of composite components

Furthermore, formal analyses like reachability analysis (see Fig. 12), non-determinism check, variable bounds checking and the verification of temporal logic patterns (see Fig. 13) are available: due to the formal semantics of FOCUS, AUTOFOCUS 3 can embed a precise translation of the software architecture into the language of the NuSMV/nuXmv [16] model checker. Note that this is all done in a complete transparent way: the translation and call to the model checker are all done in the background to provide user-friendliness

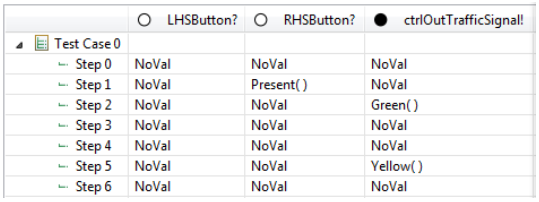

Fig. 11. Test suites

workflow. The results of the model checker, namely their counterexamples, are also translated back in the same way e.g., a counter example to a temporal logic property can be graphically simulated.

Fig. 12. Unreachable state

Fig. 13. Verification result

C. Hardware Architecture

AUTOFOCUS 3 allows to model the hardware: processors (or, in the automotive domain, ECUs – Engine Control Units), buses, actuators and sensors can explicitly be modeled as well as their connections (Fig. 14). Multi-core platforms with

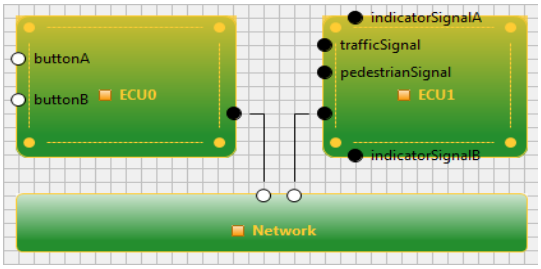

Fig. 14. Hardware architecture for generic ECUs

shared memory are available (Fig. 15), as well as specific *domain specific* hardware e.g., a *pacemaker* platform was built specifically for building and deploying models of a pacemaker. Similarly, automotive-specific hardware is supported via FIBEX import/export (an XML-based standardized format used for representing TDMA-networks).

Hardware architecture models actually deal with more than just hardware: they typically include a *platform* architecture

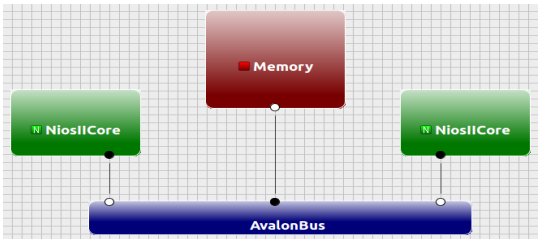

Fig. 15. Hardware architecture for multicore, with shared memory

which encompasses execution environments from bare metal hardware (e.g., chips and wires) over operating system environments up to higher-level runtime environments (e.g., Java virtual machine with remote method invocation mechanism). For instance, the aforementioned hardware model for FIBEX comes also with an implementation for the Flexray protocol ([17]), a time-triggered communication protocol in the automotive domain.

III. SEAMLESS INTEGRATION

An integration of all of the previously mentioned viewpoints in an integrated model, resp. tooling environment is an important asset for the user: it avoids the effort to integrate tools, defining their interfaces and dealing with conflicting, missing, or badly documented tool-interfaces, semantics and standards. However, if these viewpoints remain completely independent or *if the dependency between them remains informal* as it is the case with most existing tools, then only very few benefits result from its integration. Instead, AUTOFOCUS 3 allows for model integration leading to a consistent, integrated system design. In the following, we present these models as well as analysis and synthesis techniques that demonstrate the benefits of AUTOFOCUS 3 resulting from this strong integration.

A. Tracing Requirements to the Software Architecture

Traces. The integration of requirements (Section II-A) and the software architecture (Section II-B) can be achieved in various ways. A first simple integration is the use of informal traces: for each requirement, traces to components of the software architecture can be added (see Fig. 16). These traces

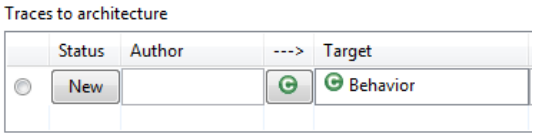

Fig. 16. Traces to the software architecture

indicate that the component(s) linked to the requirement shall fulfill the requirement. Such traces are automatically visible (and navigable) at the level of the component architecture (Fig. 17). These traces can be used to display information to the user. For example, a global visualization of the traces, both between requirements and between requirements and elements of the software architecture, is available and allows the user

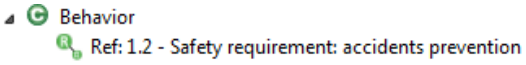

Fig. 17. Traces, seen *from* the software architecture

to have an overall picture of the intra- and inter-viewpoint relations (Fig. 18).

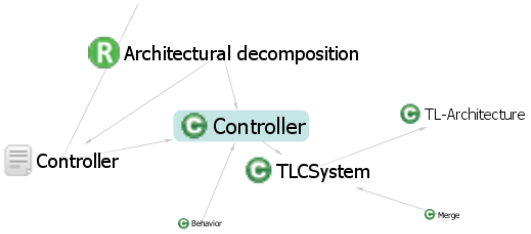

Fig. 18. Traces visualization

Refinements. Traces provide informal connections between model elements, which is to be expected since most requirements are (at least at first, or partly) informal. However, as explained in Section II-A, requirements elicitation can go far enough that a formalized behavior is obtained, e.g., that a state automaton is given. In such cases, traces to the software architecture can be enhanced into *refinements* describing not only a mere connection, but even expressing a formal relation between the requirements-level behavior description and a software-level implementation of it. A refinement is simply defined by expressing how values at the requirement level shall be transformed into values at the software level and vice versa (Fig. 19). Such refinements can then be used

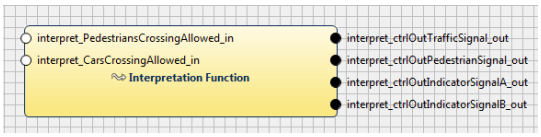

Fig. 19. Refinement definition

to automatically derive implementation-level test suites from requirements-level ones [18], [19] (which can be automatically generated according to some coverage criteria) or to verify by model checking that a component indeed implements a functionality.

Connecting MSCs to the Software Architecture. MSCs can be used in requirements in a semi-formal setting, i.e., the MSC entities represent actors identified in the requirements. Or they can be used in a completely formal setting: when the requirement elicitation is advanced enough, MSC entities can refer to components and the messages between entities can denote signals exchanged through channels. This can be expressed directly in the MSC editor, e.g., Fig. 20 shows how the properties of an MSC entity named "Merge Entity" refers to a component named "MergeComponent". The same

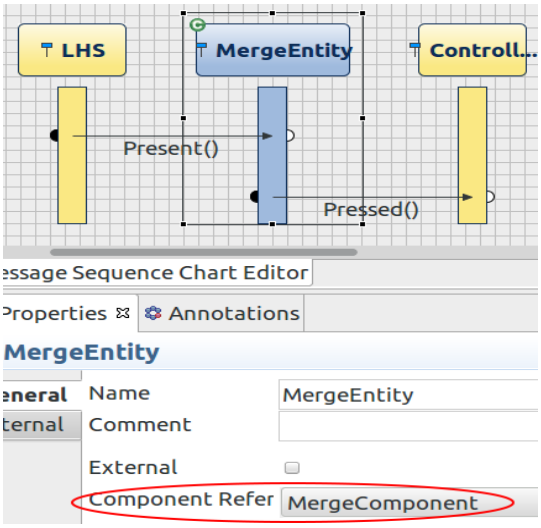

Fig. 20. Connecting MSCs to components

holds for the messages which can be connected to ports of the corresponding components.

Once these connections are provided, AUTOFOCUS 3 allows to verify, by translating the MSC into a temporal logic formula and by using model checking (in the style of [20]), that the given MSC is indeed feasible in the software architecture with the referenced components and ports. When feasible, the resulting run can be simulated in AUTOFOCUS 3.

B. Deployment of the Software Architecture on the Platform

Deployment of Software to Hardware Components. The integration of the software (Section II-B) and hardware architecture (Section II-C) is done by using deployment *models* that describe the integration between different viewpoints. Such deployments map components from the software viewpoint to the hardware viewpoint. Furthermore, the deployment model not only contains the mapping of components but also the allocation of logical ports of a software component to their corresponding hardware "ports", namely hardware *sensors* for sensor values and hardware *actuators* for actuator outputs. Fig. 21 illustrates this deployment. Note that the hierarchical structure of components makes it impossible to have such a simple GUI for deployments in AUTOFOCUS 3, making the actual interface different than the one presented in this figure.

Design Space Exploration for Model Synthesis. Once a deployment is defined, Design Space Exploration methods can be applied to define the *scheduling* of the software components on their respective hardware components. AUTOFOCUS 3 provides support to achieve this step by *automated synthesis*. Actually, even deployments can be automatically synthesized as well as complete hardware architectures, according to various system requirements, e.g. timing, safety, etc. To do so we use Design Space Exploration (DSE) techniques. In [21], we demonstrate how such a joint generation of deployments and schedules can be efficiently done for shared-memory multicore architectures. Each solution of such a synthesis process already

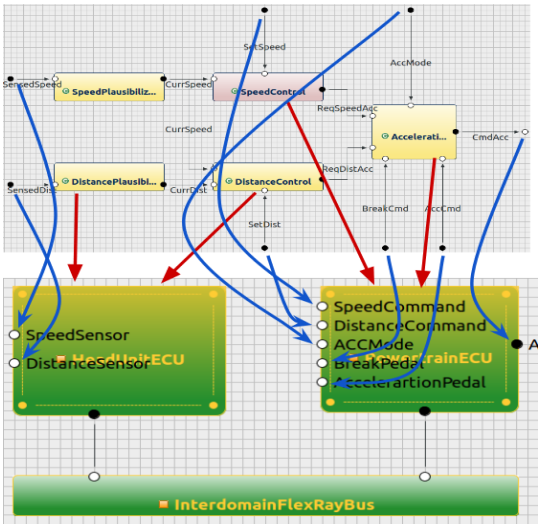

Fig. 21. Deployment (illustration)

Fig. 22. Design Space Exploration results

contains a possible deployment, which in turn already contains a valid schedule (cf. Fig. **??**). This reduces the effort and complexity in a the workflow for the identification of valid system designs.

Our approach relies on a symbolic encoding scheme, which enables to generate the desired models. The symbolic encoding is done by defining a precedence graph of components based on the software architecture as a set of tasks and messages and their connections including further information concerning predefined allocations to hardware architecture.

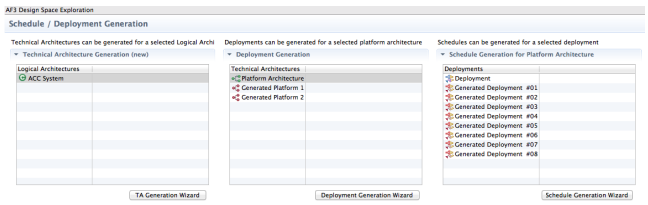

Fig. 23. Design Space Exploration Workflow

The proposed approach has proven to perform in a scalable fashion for practical sizes ([21]), as it relies on a symbolic formalization encoding the deployment synthesis as a satisfiability problem over Boolean formulas and linear arithmetic constraints. A state-of-the-art *satisfiability modulo theories (SMT)* solver, namely Z3 [22], is used to compute these solutions. Using Design Space Exploration techniques during system development involves the software engineer/designer itself. The system designer is often not just interested in an automatically synthesized solution, but even more in various solutions that can be compared. Therefore, visualization techniques [23] are part of a Design Space Exploration approach that leverages to guide the system designer through the solution space.

Furthermore, we propose a tooling concept that includes a Design Space Exploration Workflow (Fig. 23) enabling to use intermediate results for next optimization steps, e.g. a *Generated Deployment* or a *Scheduling Synthesis*.

C. Holistic Code Synthesis for Deployed Systems

Once the software architecture, the platform architecture, and a (manually defined or automatically synthesized) deployment model are defined, AUTOFOCUS 3 provides the possibility to have holistic code generation.

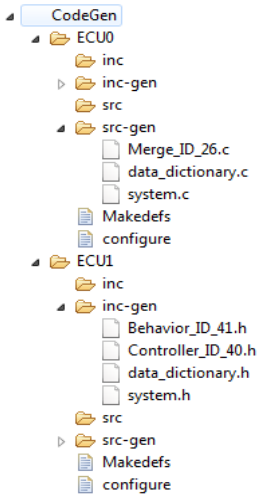

Fig. 24. Generated C code for the deployed system

The input to the generation facility is the mapping of software components to platform execution units. The result of the code generator is a full implementation of the system model including configuration files for the underlying operating system as well as bus message catalogs and compile and build environment configurations (see Fig. 24).

The code generator consists of two parts: the software architecture general purpose generator and the platform-specific target generator. The former translates the different types of software behavior specifications into an intermediate code model. From this intermediate representation the final system code (see Fig. 25) is generated by the ECU specific code generator using the ECU target language (e.g. C, Java, VHDL).

Note that these specific generators can ignore the intermediate implementation in cases the original behavior specification

```
void run_system(){
    read_input( );
    prepare_output_net( );
    perform_step_Merge_ID_26( );
    if (noval_mergeOutRequest_ID_38 == true) {
        set_noval_net_mergeOutRequest( );
    }
    else {
        write_net_mergeOutRequest(mergeOutRequest_ID_38);
    }
    finish_output_net( );
}
```

Fig. 25. Main loop of the ECU running the Merge component

can be implemented more efficiently when applying a transformation to the target language and/or hardware directly (e.g., a state-less computation component might be implemented efficiently on a FPGA sub-unit available to the ECU).

Every platform integrated in AUTOFOCUS 3 must provide its extension to the target generator as well as a justification that it upholds the semantics of the model of computation of the software architecture. Likewise, the software code parts must also be behaviorally equivalent to these formal semantics. Proving such semantic equivalences can be cumbersome [24], but is absolutely necessary in order to avoid breaking functional properties established by earlier validation and verification methods.

D. Safety Cases

To argue about the safety of systems, *Safety Cases* are a proven technique that allows a systematic argumentation. Safety Cases may contain complex arguments that can be decomposed corresponding to modular system artifacts which are generally *dependent on artifacts from different viewpoints*: e.g., requiring redundancy for safety has an impact both on software *and* on hardware architectures. Such assurance cases

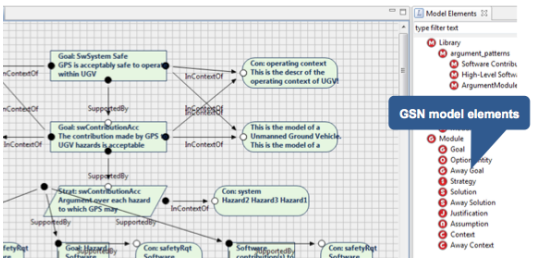

Fig. 26. GSN-based Safety Cases

are generally not well integrated with the different system models, resp. viewpoints. To provide a comprehensible and reproducible argumentation and evidence for argument correctness, we make use of the integrated system model. Since AUTOFOCUS 3 provides such integrated models at its core, it leverages the possibility to tightly connect these system models with safety case artefacts in order to form a comprehensive safety argumentation. In AUTOFOCUS 3 we provide safety case modelling based on Goal Structuring Notation (GSN) [25], as illustrated Fig. 26. Different safety case artifacts can be connected to their corresponding system artifacts (e.g., a safety goal to a requirement from the requirement viewpoint). This – for instance – enables to automatically guide the construction of the system architecture w.r.t. the safety claims, as we demonstrated in [26].

IV. RELATED WORK

There are many model-based tools which target the development/architecting of embedded systems, but none of them, to our knowledge, presents all the features of AUTOFOCUS 3.

Papyrus [27] with Moka[1] allows the execution of models based on the fUML [28] Semantics. Code generation is, as far as we know, only partly implemented, but considering the fast growth of Papyrus and Moka, this is should only be a question of time. A more significant difference to AUTOFOCUS 3 is that AUTOFOCUS 3 integrates all the modules into a unified software instead of being made of separate modules for diagram editing (Papyrus) and execution semantics (Moka). This has a significant impact on the verification (either testing or formal verification): in AUTOFOCUS 3, the semantics for execution and verification are intrinsically identical; in Papyrus additional work is required to synchronize the semantics of Moka and the verification tool, for example Diversity[2] – a verification tool typically used together with Papyrus.

The widespread commercial tool IBM Rational Rhapsody[3] has been offering for a long time a complete tool chain until code generation. Rhapsody has a precisely defined semantics [29]. It has even been used as a basis to provide integrated formal verification [30]. However, it is not as tightly integrated as AUTOFOCUS 3, not open source and is essentially used for commercial use and not as a platform for research experiments as AUTOFOCUS 3. The design space exploration viewpoint of AUTOFOCUS 3 is a research tooling concept which is a good example of such an experiment which differentiates AUTOFOCUS 3 from Rhapsody. Similar considerations hold for Bridgepoint (or xtUML)[4], LieberLieber Embedded Engineer[5] and the Enterprise Architect (EA)[6] that supports many modeling languages such as UML or SysML. ADORA (Analysis and Description of Requirements and Architecture)[7] is a research tool that supports an object-oriented method and a modeling language also called ADORA [31]. ADORA targets requirements and the software architecture of a system. Hardware is not included.

Ptolemy II [32] is similar to AUTOFOCUS 3 in the sense that it is based on formal semantics and provides code generation. Like AUTOFOCUS 3, it is an open source and academic tool which is used for research. However, Ptolemy targets only the software architecture: neither requirements nor the hardware are integrated. This arises from the fact that the

[1] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution
[2] http://projects.eclipse.org/proposals/diversity
[3] www-01.ibm.com/software/awdtools/rhapsody/
[4] https://xtuml.org/
[5] http://www.lieberlieber.com/en/embedded-engineer-for-enterprise-architect/
[6] http://www.sparxsystems.com/products/ea/index.html
[7] http://www.ifi.uzh.ch/rerg/research/adora.html

development of embedded systems is not the main focus of Ptolemy.

The SCADE Suite[8] is a commercial tool well-known in the development of control software, for example in avionics. While the SCADE Suite[9] offers a lot of functionality with respect of simulation, verification and code generation, at the moment it does not provide any support for requirements.

V. Conclusion

In this paper, we presented AUTOFOCUS 3 and the tooling concepts that it supports at different steps in the development process. AUTOFOCUS 3 is based on a completely integrated model-based development approach from requirements elicitation to deployment on the platform of code which allows to generate code *completely* (i.e., without further human modification) from the models. Based on well-defined semantics, AUTOFOCUS 3 demonstrates how integrated models are enablers for a wide range of analysis and synthesis techniques such as testing, model checking and deployment and scheduling synthesis. Tooling concepts in AUTOFOCUS 3 demonstrate how to make use of these techniques in a model-based development process.

References

[1] M. Broy and K. Stølen, *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[2] K. Pohl, H. Hönninger, R. Achatz, and M. Broy, *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer Publishing Company, Incorporated, 2012.

[3] U.S. Office of Management and Budget and U.S. Office of E-Government and IT, "A Common Approach to Federal Enterprise Architecture."

[4] F. Huber, B. Schätz, A. Schmidt, and K. Spies, "AutoFocus - A Tool for Distributed Systems Specification," in *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, ser. LNCS, vol. 1135. Springer Verlag, 1996, pp. 467–470.

[5] F. Hölzl and M. Feilkas, "Autofocus 3: A scientific tool prototype for model-based development of component-based, reactive, distributed systems," in *Proceedings of the 2007 International Dagstuhl Conference on Model-based Engineering of Embedded Real-time Systems*, ser. MBEERTS'07. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 317–322.

[6] S. Teufl, D. Mou, and D. Ratiu, "MIRA: A tooling-framework to experiment with model-based requirements engineering," in *21st IEEE International Requirements Engineering Conference, RE*, Rio de Janeiro-RJ, Brazil, 2013, pp. 330–331.

[7] A. Campetelli, F. Hölzl, and P. Neubeck, "User-friendly model checking integration in model-based development," in *24th International Conference on Computer Applications in Industry and Engineering (CAINE)*, November 2011.

[8] S. Voss, J. Eder, and F. Hölzl, "Design space exploration and its visualization in AUTOFOCUS3," in *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering*, Kiel, Deutschland, 25.-26. Februar 2014, pp. 57–66.

[9] T. Kelly, C. Carlan, and S. Voss, "Model-based safety cases in autofocus3," in *1st International Workshop on Assurance Cases for Software-intensive Systems (ASSURE)*, 2013, tool demonstration.

[10] M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, K. Scheidemann, M. Spichkova, and D. Trachtenherz, "A top-down methodology for the development of automotive software," Technische Universität München, Tech. Rep. TUM-I0902, January 2009.

[11] M. Feilkas, F. Hölzl, C. Pfaller, S. Rittmann, B. Schätz, W. Schwitzer, W. Sitou, M. Spichkova, and D. Trachtenherz, "A Refined Top-Down Methodology for the Development of Automotive Software Systems: The KeylessEntry System Case Study," Technische Universität München, Tech. Rep. TUM-I1103, Februar 2011.

[12] W. Böhm, M. Junker, A. Vogelsang, S. Teufl, R. Pinger, and K. Rahn, "A formal systems engineering approach in practice: an experience report," in *1st International Workshop on Software Engineering Research and Industrial Practices, SER&IPs*, Hyderabad, India, June 2014, pp. 34–41.

[13] B. Bernhard Schätz, "Model-based development of software systems: From models to tools." Habilitation Thesis, Technische Universität München, 2009. [Online]. Available: http://www4.in.tum.de/ schaetz/papers/Habiliationsschrift.pdf

[14] P. Bishop and R. Bloomfield, "A methodology for safety case development," in *Safety-Critical Systems Symposium*. Birmingham, UK: Springer-Verlag, ISBN 3-540-76189-6, Feb 1998.

[15] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE*, 1999.

[16] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *Computer Aided Verification*. Springer International Publishing, 2014, pp. 334–342.

[17] F. Consortium, "Flexray communications system protocol specification, version 2.1, revision A," URL http://www.flexray.com, 2005.

[18] D. Mou and D. Ratiu, "Binding requirements and component architecture by using model-based test-driven development," in *Twin Peaks of Requirements and Architecture (Twin Peaks)*, 2012.

[19] J. O. Blech, D. Mou, and D. Ratiu, "Reusing test-cases on different levels of abstraction in a model based development tool," in *MBT*, 2012, pp. 13–27.

[20] S. Li, S. Balaguer, A. David, K. Larsen, B. Nielsen, and S. Pusinskas, "Scenario-based verification of real-time systems using uppaal," *Formal Methods in System Design*, vol. 37, no. 2-3, pp. 200–264, 2010.

[21] S. Voss and B. Schätz, "Scheduling shared memory multicore architectures in AF3 using Satisfiability Modulo Theories," in *MBEES*, 2012, pp. 49–56.

[22] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[23] S. Voss, J. Eder, and F. Hölzl, "Design space exploration and its visualization in AUTOFOCUS3," in *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering*, Kiel, Deutschland, 25.-26. Februar 2014 2014, pp. 57–66. [Online]. Available: http://ceur-ws.org/Vol-1129/paper33.pdf

[24] F. Hölzl, "The AutoFocus 3 C0 Code Generator," Technische Universität München, Tech. Rep. TUM-I0918, 2009.

[25] T. Kelly and R. Weaver, "The goal structuring notation – a safety argument notation," in *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[26] S. Voss, C. Cârlan, B. Schätz, and T. Kelly, "Safety case driven model-based systems construction," in *EITEC*, 2015.

[27] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic, "Papyrus: A UML2 tool for domain-specific language modeling," in *Model-Based Engineering of Embedded Real-Time Systems - International Dagstuhl Workshop*, Dagstuhl Castle, Germany, November 4-9 2007, pp. 361–368, revised Selected Papers.

[28] T. O. M. Group, *Semantics of a Foundational Subset for Executable UML Models (FUML)*. Pearson Higher Education, 2013. [Online]. Available: http://www.omg.org/spec/FUML/1.1

[29] D. Harel and H. Kugler, "The rhapsody semantics of statecharts (or, on the executable core of the uml)," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, vol. 3147, pp. 325–354.

[30] I. Schinz, T. Toben, C. Mrugalla, and B. Westphal, "The rhapsody uml verification environment," in *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, ser. SEFM. Washington, DC, USA: IEEE Computer Society, 2004, pp. 174–183.

[31] M. Glinz, S. Berner, and S. Joos, "Object-oriented modeling with adora," *Inf. Syst.*, vol. 27, no. 6, pp. 425–444, Sep. 2002.

[32] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[8] http://www.esterel-technologies.com/products/scade-suite/

[9] http://www.esterel-technologies.com/products/scade-suite/

Introduction to WUCOR (1st International Workshop on UML Consistency Rules)

Damiano Torre (Primary Contact)
Carleton University
Software Quality Engineering Laboratory
Ottawa, Canada
University of Castilla-La Mancha
ALARCOS Research Group
Ciudad Real, Spain
dctorre@sce.carleton.ca

Yvan Labiche
Carleton University
Software Quality Engineering Laboratory
Ottawa, Canada
labiche@sce.carleton.ca

Marcela Genero
University of Castilla-La Mancha
ALARCOS Research Group
Ciudad Real, Spain
marcela.genero@uclm.es

Maged Elaasar
Carleton University
Software Quality Engineering Laboratory
Ottawa, Canada
melaasar@gmail.com

INTRODUCTION

The Model Driven Architecture (MDA) [1] is an approach to the development of software systems that promotes the use of transformations between successive models from requirements to analysis, to design, to implementation, and to deployment [2]. Much attention has been paid to MDA by academia and industry in recent years [3], which has resulted in models gaining more importance in software development. The Unified Modeling Language (UML) [4] is the Object Management Group's specification most frequently used and is the de-facto standard modeling language for object-oriented modeling and documentation [5]. It is the most commonly used modeling language to implement the MDA although it should not be used in every single software development project [6]. The UML provides 14 diagram types [4] that can be used to describe a system from different perspectives (e.g., structure, behavior) or abstraction levels (e.g., analysis, design), which helps deal with complexity and distribute responsibilities between stakeholders. Those diagrams help support many software development activities, such as: transforming an analysis model into a design model, transforming a design model into an implementation, generating documentation, model-driven testing, model-driven validation and verification, performance estimation, and schedulability analysis. Since the various UML diagrams describe different perspectives of one, and only one, software under development, they strongly depend on each other and hence must be consistent. To be successful, any software development activity that consumes a UML model made of diagrams, such as the ones mentioned earlier, requires that those diagrams be consistent. As UML is not a formal notation, inconsistencies may arise in the UML specification of a complex software system when such specification requires multiple diagrams to describe different perspectives of the software [7]. When UML diagrams portray contradicting or conflicting meaning, the diagrams are said to be inconsistent [8]. Such inconsistencies may be a source of faults in the software system [9]. It is therefore paramount that they be detected, analyzed and fixed [10], which requires that consistency between the diagrams of a UML model be first specified. One can find some UML diagram consistency specifications in the UML standard itself, where they are often referred to as well-formedness rules. As discussed in the literature, one can reason about consistency according to different dimensions: Horizontal vs. Vertical vs. Evolution Consistency, Syntactic vs. Semantic consistency, and Observation vs. Invocation consistency [11]. One can find consistency specification in the UML standard itself. One can also imagine consistency specification that is specific to a domain (e.g., telecom, aerospace), to an organization, to a project or a team. Even though there is a need for UML diagram consistency, even though there exist different ways to reason about consistency rules, one can observe from the literature [11] that: 1) there is no well-accepted set, as complete as possible, of consistency specification rules, or simply rules, for UML diagrams (beyond the small set of well-formedness rules in the standard specification); 2) many researchers have proposed, explicitly or implicitly, rules to detect inconsistencies, without any effort to validate those rules; 3) the majority of the consistency rules target a small subset of the UML diagrams (mostly, class, sequence, and state machine diagrams); 4) a non-negligible set of consistency rules are provided over and over again by researchers (instead of, for instance, referring to an accepted list of such rules); 5) a non-negligible set of consistency rules presented by researchers are actually included in the UML standard itself; 6) the UML standard is far from providing a comprehensive set of consistency rules; 7) the vast majority of consistency rules are horizontal and syntactic (other dimensions are barely used in those rules). These observations motivated WUCOR, during which we sought the opinion of experts about the consistency rules researchers have been defining in the literature, and the

rules that may be missing. The goal of this workshop has been to gather community input and feedback on UML consistency rules in general. WUCOR provided an opportunity for researchers who have been working on UML consistency, or whose (research) activities require consistent diagrams, to engage with each other in a highly interactive venue so that the group could validate the rules that have been collected and pave the path for future initiatives. The objective of the workshop has been to bring together any one, either from the industry or academia, interested in consistency rules between UML diagrams of a given model, and to provide a platform for discussions, interactions and collaborations regarding this topic. One of the starting point for the discussion groups was the set of 190 unique consistency rules we have coalesced in our work [12]. We also asked for expert opinion about a subset of those rules that are deemed paramount, and should therefore always be enforced, and other rules that can be considered optional. The final program of the WUCOR is presented in TABLE I.

TABLE I. SCHEDULE OF WUCOR

Time	Duration	Activity
8:45am	5min	Welcome to WUCOR
8:50am	25min	Bernhard Hoisl and Stefan Sobernig. Consistency Rules for UML-based Domain-specific Language Models: A Literature Review
9:15am	25min	Dan Chiorean, Vladiela Petrascu and Ioana Chiorean. Proposal for Improving the UML Abstract Syntax
9:40am	40min	1st Actvity about dimensions of UML Consistency
10:20am	25min	Coffe Break
10:45am	1hr	2st Actvity about UML diagrams involved in UML Consistency
11:45am	1hr15min	Lunch Break
1:00pm	10min	Introduction to UML Consistency Rules
1:10pm	1hr50min	3rd Activity about UML consistency rules in Model-Driven Development
3:00pm	20min	Coffe Break
3:20pm	1hr25min	Discussion and Presentation of Results
4:45pm	15min	Conclusion, Summary and Next Steps

The WUCOR proceedings collect the two papers presented at the workshop (shown in TABLE I). Those submitted papers were peer-reviewed by three independent reviewers. The two accepted papers discuss 1) a review about UML-based Domain-specific Language Models, and 2) a proposal for Improving the UML Abstract Syntax; both papers were considered very related to UML Consistency rules issues.

We would like to thank the authors for submitting their papers to WUCOR. We are also grateful to the members of the Program Committee and to the MODELS 2015 organizers for their support during the workshop organization. For more information about WUCOR please visit the workshop website at https://wucor.wordpress.com. The Program Committee was composed by :

- Steve Cook, Hidden Symmetry Ltd, UK
- Alexander Egyed, Johannes Kepler University, Austria
- Kenn Hussey, Committerati Consulting, Canada
- Zbigniew Huzar, Wroclaw University of Technology, Poland
- Robert Karban, Jet Propulsion Laboratory, USA
- Florian Noyrit, CEA LIST, France
- Richard Paige, University of York, UK
- Gianna Reggio, Università di Genova, Italy
- Nicolas Rouquette, Jet Propulsion Laboratory, USA
- George Spanoudakis, City University London, UK
- Mehrdad Sabetzadeh, University of Luxembourg, Luxembourg
- Miroslaw Staron, University of Gothenburg, Sweden

ACKNOWLEDGMENTS

This work has been funded by the SIGMA-CC project (Ministerio de Economía y Competitividad and Fondo Europeo de Desarrollo Regional FEDER, TIN2012-36904) .

REFERENCES

[1].Mukerji, J., Miller, J.: Overview and guide to OMG's architecture. Object Management Group (2003), http://www.omg.org/mda/
[2].Thomas, D.: MDA: Revenge of the modelers or UML utopia? IEEE Software 21, 15–17 (2004)
[3].Lucas, F.J., Molina, F., Toval, A.: A systematic review of UML model consistency management. Information and Software Technology 51, 1631-1645 (2009)
[4].OMG: OMG Unified Modeling LanguageTM - Superstructure Version 2.5. Object Management Group (2013)
[5].Pender, T.: UML Bible (2003)
[6].Petre, M.: UML in practice. Proceedings of the 35th International Conference on Software Engineering, pp. 722-731. (2013)
[7].Ibrahim, N., Ibrahim, R., Saringat, M.Z., Mansor, D., Herawan, T.: Consistency rules between UML use case and activity diagrams using logical approach. International Journal of Soft. Engin. and its Applicat. 5, 119-134 (2011)
[8].Simmonds, J., Straeten, R.V., Jonkers, V., Mens, T.: Maintaining Consistency between UML Models using Description LogicZ. RSTI – LMO'04 10, 231-244 (2004)
[9].Muskens, J., Bril, R.J., Chaudron, M.R.V.: Generalizing Consistency Checking between Software Views. Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, pp. 169-180. (2005)
[10].Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Chang, S.K. (ed.) Handbook of Software Engineering and Knowledge Engineering, pp. 329-380. (2001)
[11].Torre, D., Labiche, Y., Genero, M.: UML consistency rules: a systematic mapping study. (EASE 2014). (2014)
[12].Torre, D., Labiche, Y., Genero, M., Elaasar, M.: A systematic identification of consistency rules for UML diagrams. Carleton University (2015), http://goo.gl/TFMgnE

Consistency Rules for UML-based Domain-specific Language Models: A Literature Review

Bernhard Hoisl and Stefan Sobernig
Institute for Information Systems and New Media
Vienna University of Economics and Business (WU Vienna)
{bernhard.hoisl, stefan.sobernig}@wu.ac.at

***Abstract*—The Unified Modeling Language (UML) has become a popular implementation vehicle for domain-specific modeling languages (DSMLs). A UML-based DSML is typically defined by multiple specification artifacts, i.e. inter-related models, describing different views on the DSML. These separate, yet inter-related models are potential sources of specification inconsistencies which bear a high risk of affecting all subsequent DSML development phases (e.g., platform integration). In a large-scale literature review of more than 8,000 publications, we collected evidence on consistency-rule usage for 84 UML-based DSML designs. In this paper, we report on the identified patterns of consistency-rule usage (e.g., rule formalization, rule scopes, and supported development activities) and specification defects which challenge the use of consistency rules in DSML specifications.**

I. Introduction

Domain-specific modeling languages (DSMLs) are specialized modeling languages tailored primarily for graphical modeling tasks in a particular application domain to support the model-driven development (MDD) of software systems for this domain. As a special kind of domain-specific languages (DSLs), DSMLs provide end users with at least one graphical or diagrammatic concrete syntax—in contrast to, for example, textual or form-/table-based DSLs (see, e.g., [1], [2]).

In recent years, developing DSMLs based on the Meta Object Facility (MOF [3]) and integrated with the Unified Modeling Language (UML [4]) has become a widely adopted option (see, e.g., [5], [6]). A UML-based DSML tailors its host language (i.e. the UML) to the needs of a particular domain (e.g., by introducing domain-specific model elements or by restricting the semantics of existing UML elements). These domain-specific aspects are specified on the level of a DSML's language model, which captures all relevant domain abstractions and specifies the relations between these abstractions (see, e.g., [7]). To tailor a DSML's language model, language-model constraints are employed, for example, specified by informal textual annotations (e.g., UML comments [4]) or in a formal language (e.g., OCL [8]).

In the DSML context, consistency rules are devised to ensure that the different artifacts of a UML-based DSML do not contradict each other due to conflicting syntax and semantics specifications (see, e.g., [9]–[11]). A DSML specification covers also the phases of defining the DSML's concrete syntax, behavior, and platform integration [7]. The result of such a DSML specification are multiple interdependent specification artifacts. For example, DSML-specific constraints—as part of a DSML's language model—need to be enforced for all instance models to ensure compliance with their respective metamodel (i.e. the DSML's language model). Furthermore, the UML provides 14 different model and diagram types to specify different (structural and behavioral) concerns of a software system [4]. DSMLs can build on multiple model and diagram types at the same time, therefore, putting emphasis on inter-model consistency.

In a recent systematic literature review (SLR), we extracted design decisions from UML-based DSMLs and collected the corresponding DSML specification artifacts [12]. The review is a data source for two aspects of consistency rules for UML-based DSML specifications. First, we extracted data on consistency-rule usage in DSML specifications. From 84 DSML designs, we retrieved details on employed consistency-rule formats, DSML language-model formalizations, consistency-rule scopes, supported software-engineering activities, the underlying UML model and diagram types, and supporting software tools. This complements the work on consistency rules by [10], [11] from the perspective of DSMLs realized as UML extensions. Second, the review spotted critical specification defects for UML-based DSMLs. These defects in the UML formalization of a DSML's language model (e.g., incomplete and insufficient specification of UML profiles, incorrect use of constraint-language expressions) result in issues for defining consistency rules.

In summary, the key contributions of this paper are the extraction, analysis, and discussion of consistency-rule usage in UML-based DSML designs. This complements the work by Torre et al. ([10], [11]) which focusses on UML in general. In addition, the paper highlights challenges specific to UML-based DSMLs when it comes to providing an infrastructure for defining consistency rules, including recommendations to avoid commonly observed pitfalls in DSML development. On top, we provide descriptive findings on (extended) UML usage (e.g., UML diagram types) adding to the current body of empirical research on UML (see, e.g., [13], [14]).

The remainder of the paper is structured as follows. Section II summarizes important background information with respect to DSML development, SLR procedure, and specification consistency in this context. Results of the data-extraction process are presented in Section III, limitations of the SLR in Section IV. Section V puts the extracted data on consistency-rule usage in DSMLs into perspective and discusses the role

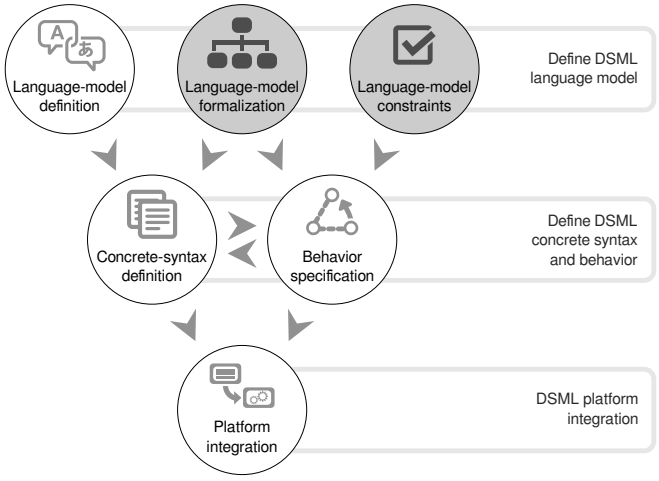

Fig. 1. Language-model driven DSML development process.

of specification defects in this context. Our study is compared to related work in Section VI and Section VII concludes the paper.

II. Background

The following three sections recap details of developing DSMLs (Section II-A), conducting the SLR (Section II-B), and evaluated UML consistency aspects (Section II-C) important to interpret the results presented in Section III.

A. DSML Development

DSML development is an exploratory, iterative process. A process view (such as [7]) treats DSML development as a complex flow of characteristic development activities (e.g., language model definition, constraint specification etc.). We focus on a language-model driven DSML development activity which discriminates between the following development phases [7]: define DSML language model, define DSML concrete syntax and behavior, and DSML platform integration (see Fig. 1).

In our work of evaluating consistency rules for UML-based DSMLs, we target the phase of the domain-specific language model definition (see Section I and Fig. 1). In this first phase of a language-model driven DSML development activity, a core language model and the corresponding language model constraints for the selected target domain are defined. By following a domain analysis method, such as domain-driven design (see, e.g., [15]), domain abstractions are identified and form the language model of a DSML. This initial *language-model definition* may not be UML-compliant (e.g., textual descriptions, informal models) and need to be turned into a *formal language model*. By formal model, we refer to a realization of the language model using a well-defined metamodeling language such as the MOF/UML metamodeling infrastructure. A metamodeling language is itself based on a well-defined and well-documented language model (i.e. CMOF for the UML metamodel [3]) and provides at least one well-defined and well-documented concrete syntax to define an own language model (e.g., the CMOF diagram syntax to specify a UML metamodel extension).

Because the language model often cannot (or only insufficiently) capture all restrictions and/or semantic properties of the DSML elements, *language-model constraints* are added. These language-model constraints prevent the language model to be formalized incomplete, ambiguous, and/or inconsistent with other DSML artifacts. As such, language-model constraints form the basis for the definition of consistency rules and are specified, for example, by employing special-purpose constraint languages (such as the OCL [8]) or unstructured textual annotations.

After the definition of a DSML's language model, the *concrete syntax* of a DSML is defined (i.e. suitable notation symbols as well as composition and production rules) which serves as a DSML's user interface (see Fig. 1). In parallel, the *behavior* of DSML language elements is specified to produce the behavior intended by the DSML designer. In the last phase, all artifacts defined for a DSML are *integrated* into a selected software platform to produce platform-specific (executable) models (e.g., by employing model transformations to generate source code [16]).

B. Systematic Literature Review

We performed a systematic literature review (SLR) to distill generic design decisions from UML-based DSML design documents for the different development phases discussed in the former section. Here, we briefly summarize the process and the results of the SLR; details are published in [12] and in [17]. The main goal of the SLR was to identify a maximum number of high-quality scientific publications which document design decisions on UML-based DSMLs as primary sources.

The SLR was performed in three steps. First, to provide a basis for evaluation of the search procedure, we established a corpus of reference publications as *quasi-gold standard* (QGS [18]). In essence, a QGS is a set of hand-picked publications considered relevant for a specific SLR. In the end, the constructed QGS corpus consisted of 37 publications (24 journal and 13 proceedings articles). Based on these QGS publications, the relevant search engines for the automated search were identified (SpringerLink, IEEE Xplore, Scopus, and ACM Digital Library) and a search string for the automated search was constructed (the query expression represented 544 unique pairs of search terms).

Second, we performed the actual *engine-based publication search* using the search string developed in the previous step on the four selected search engines. The search execution yielded 5,778 search hits split into four result sets, one for each of the search engines. After enforcing the QGS-based capping, having evaluated the papers based on our selection criteria and having completed the quality assessment, 73 papers representing 1.3% of the original search hits remained. For this final publication set, we extracted the publication-specific data (15 metadata items for each paper, including bibliographical entries, selection decision, and decision-mining entries).

Third, based on the bibliographical records extracted from the 73 publications selected up to this point, we then performed a *backward-snowballing search*. Backward snowballing is the practice of manually identifying additional publications for selection from the reference lists (citations) of a given set of publications [19]. Via the backward snowballing search, we reviewed a total of 2,337 references. After evaluation and quality assessment of the papers, eight were included into the paper corpus (0.3%). From these additional publications, we extracted publication-specific data in the same way as was done for papers retrieved by the main search.

We considered a total of 81 articles as relevant: 73 from main search plus eight from snowballing. To complete the paper corpus, we re-considered the QGS publications not retrieved by the main and the snowballing searches for inclusion based on the selection criteria. This way, we classified two QGS journal articles and one QGS conference article as relevant. We so arrived at a paper corpus of *84* publications (the complete list of publications is provided in [20]). The corpus was composed of 54 conference articles (64%) and 30 journal articles (36%).

C. Consistency in UML-based DSMLs

In this paper, we investigate six aspects of model-level consistency in UML-based DSML designs in line with [10], [11].

Language-model formalization: After the identification of language-model concepts, the corresponding definitions serve as input for the phase of formalizing the domain constructs into a MOF/UML-compliant language model (see Section II-A). As we focus on consistency rules at the level of a DSML's language model, we establish how the domain abstractions are formalized using the MOF and/or the UML. Available options are *UML M1 structural model* (e.g., UML class models), *UML profile definition* (i.e., extending UML metaclasses with stereotypes), and *UML metamodel extension* (i.e., adding new metaclasses and/or new associations between metaclasses to the UML metamodel) [20].[1]

Consistency-rule formats: A DSML's language model formalization is limited by the expressiveness of the MOF/UML (e.g., part-of relations). Semantic variation points in the MOF/UML may render a DSML's language-model specification incomplete and/or ambiguous. This risks introducing inconsistencies across different DSML modeling artifacts ([4], [20]). Therefore, we assess whether consistency rules are provided for a DSML to cover such variation points [10]. If so, we document the choice of rule representation (e.g., OCL expressions [8]).

Consistency-rule scopes: We record whether consistency rules target a *single model* only (e.g., to resolve ambiguities in the definition of a model) or whether the rules relate multiple models. For inter-model scenarios, *horizontal* consistency refers to consistency between different, but complementing models at the same level of abstraction (e.g., between different platform-independent models). *Vertical* consistency refers to consistency between models at different levels of abstraction (e.g., between platform-independent and platform-specific models). *Evolution* consistency refers to consistency between different versions of the same model (e.g., between an input and an output model of a model transformation [10]).

Software-engineering activities: Model-level consistency rules are employed in support of different software-engineering activities. Observed activities are *refinement* ("semantics-preserving changes applied to a model, to reduce non-determinism" [11]), *verification* ("determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase" [11]), *checking constraints* ("of models according to consistency rules and producing a list of violations" [11]), *transformation* ("describe the application of mapping rules on one model to create a new model" [11]), and *heuristics* (rules that are written as plain text "for solving a UML consistency problem without the exhaustive application of an algorithm" [11]).[2]

Model and diagram types: We document which of the 14 structural and behavioral UML model and diagram types [4] are actually tailored by the DSMLs. These are, therefore, the model and diagram types for which consistency rules are defined for various rule scopes ([10], [11]).

Tool support: We evaluate whether consistency rules (in a given representation) can be automatically processed and validated. In addition, we provide an inventory of supporting software tools for rule processing and validation (e.g., constraints evaluators [10]).

III. Extracted Data on DSML Consistency

This section presents the data on the six consistency aspects in the corpus of 84 DSML designs collected via the SLR ([12], [17]). For data extraction, we studied the corresponding publications as the primary design documents for cues on each of the six consistency criteria. 52 out of 84 DSML designs (62%) explicitly specified consistency rules at the level of the DSML's language model. For 32 DSML designs (38%), we did not find any documentation hints of consistency-rule definitions.

Table I shows the frequency of UML-based language-model formalization options identified for the 52 DSML designs. The majority of DSMLs (84%) employ UML profiles to formalize their language model. Only one DSML defines its UML-based language model via an M1 structural model. The language model of four DSMLs is specified by using a combination of a UML profile and a UML metamodel extension.

To quantify the specification size of these 52 DSML designs, we evaluated the size of their core language-models. Depending on the different, underlying UML language-model formalization options, the specification size was established differently. For 47 DSMLs defining their language models

[1]We only discuss formalization options actually observed for DSMLs in our SLR study (see Section III). A complete list of language-model formalization options is documented in [20].

[2]Again, our analysis is limited to activities observed in our SLR (see Section III). The complete list of relevant software-engineering activities is available from [11].

TABLE I
UML-BASED LANGUAGE-MODEL FORMALIZATION OPTIONS.

Language-model formalization	Frequency
UML profile definition	47 (84%)
UML metamodel extension	8 (14%)
UML M1 structural model	1 (2%)
Total	56 (100%)

TABLE II
EMPLOYED FORMATS TO SPECIFY CONSISTENCY RULES.

Consistency-rule format	Frequency
Unstructured text	36 (50%)
OCL	33 (46%)
Mathematical expressions	2 (3%)
ATL	1 (1%)
Total	72 (100%)

TABLE III
IDENTIFIED SCOPES OF CONSISTENCY RULES.

Consistency-rule scope	Frequency
Single model consistency	45 (79%)
Horizontal consistency	7 (12%)
Vertical consistency	3 (5%)
Evolution consistency	2 (4%)
Total	57 (100%)

TABLE IV
RELATING CONSISTENCY RULES TO SOFTWARE-ENGINEERING ACTIVITIES.

Software-engineering activity	Frequency
Heuristics	36 (32%)
Verification	35 (32%)
Constraint checking	33 (30%)
Transformation	4 (4%)
Refinement	3 (3%)
Total	111 (100%)

using UML profiles, we counted the stereotype definitions and the corresponding, distinct base UML metaclasses. In this group, we find a median of 14 ± 9.6[3] stereotype definitions per DSML. A typical profile extends a median of 5 ± 3 distinct base metaclasses per DSML. For the eight DSMLs using a UML metamodel extension, we collected the number of newly introduced UML metaclasses. A typical DSML adds a median of 19.5 ± 11.9 UML metaclasses. For one DSML defining its language model using a UML structural model at level M1, we were unable to count the number of UML classes due to its incomplete design documentation.

The 52 DSML designs containing consistency rules adopted four different rule formats (see Table II). 33 DSMLs (63%) use one (either unstructured text, OCL, or mathematical expressions), 18 DSMLs (35%) use two (both, unstructured text and OCL), and one DSML three different formats (unstructured text, OCL, and ATL [21]) to specify consistency rules. There are nearly equal shares of DSMLs adopting unstructured (informal) text (50%) and (formal) OCL expressions (46%). Mathematical and transformation-language expressions (e.g., in ATL) are rarely used with three DSMLs only.

The majority of 52 DSMLs (79%) apply consistency rules for the scope of a single model (see Table III). Inter-model consistency rules for a horizontal scope (i.e., at the same abstraction level) were found for seven DSMLs (12%). A minority of three DSMLs define consistency rules spanning different abstraction levels (vertical consistency). Consistency rules between different versions of a language model (evolution consistency) were reported for two DSMLs only. In five DSMLs, consistency rules had mixed scopes: single model/vertical consistency (2 DSMLs), single model/evolution consistency (2), and horizontal/vertical consistency (1).

As for software-engineering activities supported by the consistency rules, almost equal shares of DSMLs relate to three activities of heuristics (32%), verification (32%), and constraint checking (30%; see Table IV). In turn, rules rarely target model transformation and refinement activities with only four and three cases, respectively. In 17 DSMLs (33%), rules are employed for one software-engineering activity only (15x heuristics, 2x verification). Mixed usage is reported for 16 DSMLs (31%) with two supported activities (14x verification/constraint checking, 1x heuristics/transformation, 1x heuristics/refinement), and for 15 DSMLs (29%) with three activities (heuristics/verification/constraint checking). More than three supported activities are limited to a minority share of four DSMLs.

One DSML is unspecific about the UML model and diagram types it is tailoring and is therefore omitted in Table V. For the remaining 51 DSMLs, class diagrams are ranked first with a 35% share, followed by activity (12%), and component as well as package diagrams (each 11%). No DSML tailored communication, profile and timing diagrams. Given that each of the 51 DSMLs can build on multiple model and diagram types, a total of 95 tailored UML diagram types were identified. 65 (68%) are structural and 30 (32%) are behavioral diagram types. Typically, a DSML tailors more than one UML diagram type. There exists 28 unique combinations of different diagram types tailored by the DSMLs. Most DSMLs build on either class diagrams only (8 DSMLs, 16%) or class diagrams in combination with package diagrams (8). Five DSMLs adopt activity diagrams only and three DSMLs combine class and object diagrams. All other combinations of diagram types are employed by at most two DSMLs each; and are omitted for brevity.

Software tools for processing and enforcing consistency rules are shown in Table VI. In total, we identified relevant tool support for 22 out of 52 DSML designs (42%; 16 unique tools). The majority of DSMLs (58%) did not document any tool usage. Five DSMLs (23%) use the OCL project of the Eclipse Model Development Tools (MDT) and three DSMLs

[3]We report the variance in terms of the *median absolute deviation from the median* using the $\pm$ notation along with the median value.

TABLE V
TAILORED UML DIAGRAM TYPES. AN ASTERISK (*) DENOTES A STRUCTURAL, ALL OTHERS ARE BEHAVIORAL DIAGRAM TYPES [4].

UML diagram type	Frequency
Class*	33 (35%)
Activity	11 (12%)
Component*	10 (11%)
Package*	10 (11%)
State machine	7 (7%)
Composite structure*	6 (6%)
Use case	6 (6%)
Sequence	5 (5%)
Object*	4 (4%)
Deployment*	2 (2%)
Interaction overview	1 (1%)
Communication	0 (0%)
Profile*	0 (0%)
Timing	0 (0%)
Total	95 (100%)

TABLE VI
EMPLOYED TOOLS TO VALIDATE CONSISTENCY RULES.

Tool	Frequency
OCL project of the Eclipse Model Development Tools (MDT)	5 (23%)
IBM Rational Software Architect	3 (14%)
CompSize	1 (5%)
Eclipse Atlas Transformation Language (ATL)	1 (5%)
Eclipse EMF Compare	1 (5%)
EIS plug-in	1 (5%)
Gentleware Poseidon for UML	1 (5%)
ITEM ToolKit	1 (5%)
Kent Modeling Framework (KMF)	1 (5%)
LTSA	1 (5%)
No Magic MagicDraw	1 (5%)
Oclarity	1 (5%)
Octopus	1 (5%)
Telelogic Tau (G2)	1 (5%)
TOPCASED	1 (5%)
WebRatio	1 (5%)
Total	22 (100%)

(14%) IBM's Rational Software Architect (RSA) to validate consistency rules.[4] The remaining 14 software tools are each deployed in a single DSML project only.

IV. SLR LIMITATIONS

SLRs have the general problem of finding a representative set of relevant primary studies. We closely followed established guidelines on designing and conducting SLRs available from research on evidence-based software engineering to avoid any pitfalls ([18], [19], [23]). However, we may risk having missed further relevant primary studies on UML-based DSMLs. For example, by extracting data from our paper corpus, we did not find empirical evidence for every consistency-rule format proposed by related work, such as, code annotations or constraining model-to-text transformations [20]. Nevertheless, we addressed this threat right from the beginning, by building our review procedure around the principle of continuous search validation and search refinement driven by a QGS as a recommended practice [18].

We intentionally limited our SLR exclusively to DSMLs embedded into UML 2.x [4], thereby excluding DSMLs based on former UML versions and other metamodeling infrastructures (e.g., Kermeta or MetaGME). We only considered consistency rules specified on the level of UML-based DSML language models; i.e. we restricted data extraction to the DSML development phases of formalizing a UML-compliant language-model and defining accompanying language-model constraints (see Section II-A). Therefore, on the one hand, we excluded rules applied on non-UML artifacts (e.g., non-UML platform-specific models generated during the platform integration phase). On the other hand, we also excluded consistency rules relating to other UML models besides a DSML's language model (e.g., UML M1 behavioral models as part of a DSML's behavior specification). Furthermore, we excluded exemplary as well as application-specific consistency rules found in DSML reports (e.g., as part of a case study exemplifying the application of a DSML).

We exclusively report on tools used to enforce consistency rules on DSML language models. We do not consider tool support for other phases in DSML development, such as, language-model editors, generators for concrete-syntax editors, model-execution engines, model-transformation engines, or orchestration engines.

V. DISCUSSION

First, we elaborate on the relevance of the extracted data presented in Section III. Against this background, we reiterate over frequently reoccurring specification defects in UML-based DSML language models, as revealed by [12], [17].

A. Interpretation of Review Data

Language-model formalization: The preponderance of UML profiles might partly be explained as they are the native UML extension mechanism [4], their application is known to modelers, and plenty of supporting tools exists (e.g., language-model and concrete-syntax editors). UML profiles provide for packaging and for scoping intra-model consistency rules (i.e., OCL expressions) as part of a DSML's language-model formalization. To this date, portability of such OCL consistency rules between different evalution engines remains limited due to the OCL/UML language specifications leaving critical details to language and tool implementers (e.g., navigation semantics between extension and extended model elements; see [24] for an overview).

Another critical issue pertaining to (esp. formally specified) consistency rules in multi-level and shallow-instantiation-based metamodeling environments such as MOF/UML is their confinement to direct instantiations (e.g., M1) of model elements (e.g., M2). Consider as an example a DSML language model defined at level M2 which must enforce consistency

[4]We separately report the Eclipse MDT/OCL project and IBM's RSA because RSA bundles a couple of the MDT/OCL plugins deviating these in unknown ways from the official Eclipse OCL plugins [22].

rules at level M0, i.e. the occurrence (instance) level of DSML models. This requirement is documented for DSMLs in the business-process modeling domain in which consistency conditions are stipulated for the scope of business-process instances (see, e.g., [25], [26]). To date, there are certain conventions (e.g., escaping to informal rule definitions [20]), implementation idioms (e.g., prototypical concept pattern [27]), and alternatives to metamodeling based on shallow instantiation (e.g., potency and deep instantiation [28]) to work around or to address this limitation. However, no comprehensive solution has yet become available in the family of MOF/UML/OCL languages as a DSML development infrastructure.

We found that a language model can been realized by multiple formalizations (e.g., a combination of a UML profile and a UML metamodel extension as observed four times). This bears a double risk. On the one hand, consistency rules must be defined for the scope of two different artifacts, metamodel and profile packages, causing ambiguity in rule specification and possible rule conflicts. On the other hand, such a mixed DSML language model can potentially be used in different configurations (e.g., different profile and metamodel compositions). As an extreme, when integrating two or more DSMLs which are realized as (otherwise independent) UML extensions, reconciling the original consistency rules becomes a challenge [20].

Consistency-rule formats: We observed a comparatively high frequency of unstructured text and OCL expressions employed in combination (in 37% of the DSMLs). This is partly explained by the reporting needs of a scientific publication, requiring a certain level of elaboration on otherwise formal constraint expressions. Another driver might be that consistency rules expressed in some constraint-expression language must be complemented with textual explanations when applied beyond the context of a single model. To express consistency conditions between two or more models (horizontally and vertically), missing any direct and/or navigable inter-model links, alternative approaches (e.g., constructs in model-transformation languages, non-standard constructs in constraint-expression languages such as in the Epsilon Validation Language, EVL [29]) must be evaluated for adoption on a case-by-case basis. In doubt, complementary textual explanations (as we found in this study) are a viable option.

Similarly, in the context of evolution consistency, one DSML [30] specified consistency rules in a combination of OCL expressions evaluated in ATL-based model transformations (ATL can be used to define constraints on models [21]). The authors of [30] present an approach for model execution by a series of model transformation steps (exemplified by an evolving state machine diagram). In this case, OCL expressions are still employed to ensure the consistency of a single model. However, with the combination of ATL transformations and, thus, different model versions on which the OCL expressions are evaluated against step-by-step, the consistent evolution of a model is ensured.

Consistency-rule scopes: Intra-model consistency rules for DSML language models are the most frequent rule scope identified by our SLR. As for inter-model consistency, consistency was ensured by using (at least) unstructured textual artifacts. For example, in [31] textual rules are defined for horizontal consistency between composite structure and activity models in support of a heuristic activity. Vertical consistency was always observed in combination with a refinement activity. In [32], an abstract user-interface (UI) class model is refined into a UI deployment model. Consistency rules integrated with model transformations for evolution support have already been given as an example above [30].

Software-engineering activities: According to our definitions, we classified consistency rules formulated as unstructured texts as related to the software-engineering activity "heuristics" and OCL expressions as related to the constraint-checking activity (see Section II-C). This data-extraction convention explains the closely aligned figures reported for these consistency-rule formats and the corresponding software-engineering activities. Furthermore, we classified constraint checking as a verification technique (i.e. as part of the verification activity).

We did not find any evidence for the management, validation, and maintenance software-engineering activities as defined in [11]. The reason for their absence may be that these are not primary activities in the process of designing a research-driven DSML (see Section II-A). Managing consistency, evaluating the satisfaction of user requirements, or maintaining interdependencies between platform-independent models and platform-specific implementations may not be of high priority when developing scientific UML-based DSMLs (and, thus, are postponed).

Model and diagram types: In this study, we put emphasis on consistency rules formulated at the level of a DSML's language model (M2 level). These rules are enforced on instance models of a DSML (M1 level). A DSML's language model was frequently found formulated as a UML profile (in 84% of the DSMLs). Overall, multiple combinations of tailored diagram types could be observed (28 unique combinations). This combinatorial variety indicates the domain-specific application requirements matched by the diagram types adopted by each DSML found by our SLR.

Tool support: All of the 16 software tools found support the automatic evaluation of consistency rules. Because of diversified tool usage, we could not identify repeated occurrences except for the Eclipse MDT/OCL project (5 times, 23%) and IBM's RSA (3 times, 14%). Nevertheless, the small number of tooling support found (no consistency-enforcing tool was mentioned in 58% of the DSMLs) does not necessarily indicate that in these cases consistency rules are evaluated manually. In particular, we found OCL expressions being documented for 33 out of 52 DSMLs (63%), which can in principle be automatically evaluated.

B. DSML Specification Defects

Our SLR exposed six defect kinds in DSML specifications for 31 reviewed design documents ([12], [17]). As an extreme case of a DSML specification defect, metamodel and/or profile

definitions were found entirely missing (e.g., in [33]). Rather, we found that stereotypes are often applied in UML instance models without a proper profile definition ([12], [17]). In such cases, any kind of consistency rule lacks the foundation of a valid interpretation. However, there are also less obvious sources of challenges.

Such defects often reveal misconceptions about UML extension techniques. In addition, they pose particular challenges to formulating consistency rules and prevent consistency rules, if any, to serve their intended purpose. In this section, we reiterate over relevant defects related to consistency rules on DSML language models defined using the UML.

Defects in metamodel definition: The DSML's language model definition does not reference a corresponding metamodel specification, therefore, essential details about the semantics of DSML-specific metaclasses and their relationships are omitted ([12], [17]). In at least five DSML designs, we found an underspecification of metamodel elements. For example, newly introduced metaclasses did not inherit from well-defined base metaclasses (e.g., in the case of a UML metamodel extension, from metaclasses of the UML specification [4]). In such cases, any consistency rule defined for the scope of the underspecified metamodel elements remains ambiguous. This is because it is potentially redundant, restating consistency conditions already available for base metaclasses; or it is potentially conflicting with the latter.

Missing mappings between language model and profile: A frequently observed problem is that a MOF-based or modeling-language independent metamodel is implicitly aligned to a corresponding UML profile. Nevertheless, in at least seven cases, the mapping between metamodel and profile was not documented explicitly ([12], [17]). The lack of explicit documented correspondences lets the reader assume a 1:1 mapping between, for example, non-UML-compliant elements of an initial language-model definition and equally named stereotypes of a UML profile formalization.

Such an implicit mapping, often only based on simple name matching between metamodel and profile, raises the issue of porting any consistency rules from one to the other, which is often not straight forward. A possible approach is to define such mappings between (non-UML-compliant) metamodel and UML profile definitions explicitly. For example, elements of a MOF-based metamodel can be mapped to stereotypes of a UML profile in the form of model-to-model transformations expressed in ATL to ensure their consistency (see, e.g., [20]). This would allow for rendering intended semantics UML-compliant or, if not possible (e.g., semantics of UML stereotypes would contradict the UML specification), providing an explicit trace back to the semantics of the originating metamodel elements. This way, there would also be clear hints how to interpret any consistency rules defined for one artifact (metamodel) in the context of the other (profile).

Defects in profile definition: We identified 21 cases where the definition of a UML profile does not adhere to the UML specification ([12], [17]). Semantic defects encountered included stereotypes inheriting from non-stereotype classes, multiplicity declarations on stereotype extensions, composite aggregation between stereotypes, or inheritance cycles between stereotypes ([12], [17]). These defects introduce semantic variation points (e.g., the possibility of multiple behaviors), which carry over to the interpretation of consistency rules defined over these elements.

Vendor- and tool-specific extensions: In at least three cases ([12], [17]), language models are defined using vendor-specific extensions to OMG specifications that are built into a particular modeling tool (e.g., undefined visibility properties [34]). On the one hand, this raises the issue of defining non-portable consistency rules. On the other hand, to provide consistency between these proprietary additions and elements of the UML metamodel, we recommend specifying their precise semantics in the same way as was done in the UML specification (see, e.g., semantics sub-clauses in [4]).

Defects in constraint-language expressions: We encountered numerous syntactic and semantic defects, including logical errors, calling undefined functions, missing keywords, unbalanced parentheses, and misspelled metamodel elements ([12], [17]). It is obvious that consistency can only be ensured and automatic evaluation can only be provided if formal rules (e.g., OCL expressions) are defect-free. Therefore, increasing the documentation quality of constraint-language expressions is key. Documentation guidelines which require authors to check syntax and semantics of OCL expressions with dedicated tools is a starting point. Table VI provides a non-exhaustive overview of available tools.

VI. Related Work

Our data extraction criteria are closely aligned to the ones presented in [10], [11], in which the authors present a systematic mapping study identifying consistency rules for UML diagrams. The main difference to the approach of Torre et al. is that we focus on domain-specific language models extending the UML, instead of general-purpose UML diagrams. Therefore, our work can be seen as complementary. However, a key difference is that we do not strive for providing an exhaustive collection of concrete consistency rules for UML diagrams in the sense of [10], [11]. Most of the consistency rules for DSMLs are inherently specific to one application domain and to one design of a corresponding language model. As this prevents their general applicability (e.g., to the UML metamodel in general), we did not compile a catalog of consistency rules.

When comparing the collected data with the original work in [10], [11], we can confirm the predominance of consistency rules defined as unstructured text and as OCL expressions, both targeting a single model and multiple models at the same abstraction level (horizontal consistency), for the reviewed DSMLs. Verification and constraint checking are also frequently employed, although we rank heuristics activities first unlike in [10], [11]. This may be due to our data-extraction process, in which we classified each text-based consistency rule as related to the heuristics software-engineering activity as specified by the definition in Section II-C.

Regarding UML model and diagram types, a majority of empirical studies report UML classes as the most exhibited one (see, e.g., [13], [14]). At the same time, we cannot confirm the previously reported high frequency of sequence and state machine diagrams. Similarily, in the review by [10], [11] the otherwise reported frequent adoption of activity, component, and package diagram types is not confirmed. A further confirmatory finding to recent empirical studies (see, e.g., [11]) is the large amount of unique combinations of different diagram types (28). There is also an important overlap regarding supporting tools (Eclipse-based projects, No Magic MagicDraw etc.), but the small tool-specific study population at our side prevents drawing robust conclusions.

VII. Conclusion

In this paper, we analyzed consistency aspects extracted from 84 UML-based DSML designs collected via a SLR [12]. We exclusively focused on consistency rules defined on the level of a DSML's language model. For the evaluation of UML consistency aspects, we adopted criteria from close related work ([10], [11]). By interpreting extracted consistency-related data, we discussed frequently identified defects in UML-based DSML language models. Results of our study show that a UML-based DSML language model is predominantly formalized via profile definitions which tailor mostly class, activity, component, and package diagrams. Textual descriptions and the OCL are most frequently used in combination to define consistency rules on a single model for verification purposes. In the majority of cases, the DSML reports do not document any tool support for enforcing these rules. Results of our study partly confirm findings from as well as add to the observations by related work.

References

[1] D. Spinellis, "Notable design patterns for domain-specific languages," *J. Syst. Softw.*, vol. 56, no. 1, pp. 91–99, 2001.

[2] U. Zdun and M. Strembeck, "Reusable architectural decisions for DSL design: Foundational decisions in DSL development," in *Proc. 14th Europ. Conf. Patt. Lang. Prog.*, 2009.

[3] Object Management Group, "OMG meta object facility (MOF) core specification," Available at: http://www.omg.org/spec/MOF, 2015, version 2.5, formal/2015-06-05.

[4] ——, "OMG unified modeling language (OMG UML)," Available at: http://www.omg.org/spec/UML, 2015, version 2.5, formal/2015-03-01.

[5] L. Nascimento, D. L. Viana, P. A. M. S. Neto, D. A. O. Martins, V. C. Garcia, and S. R. L. Meira, "A systematic mapping study on domain-specific languages," in *Proc. 7th Int. Conf. Softw. Eng. Adv.* IARIA, 2012, pp. 179–187.

[6] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proc. 33rd Int. Conf. Softw. Eng.* ACM, 2011, pp. 471–480.

[7] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," *Softw. Pract. Exper.*, vol. 39, no. 15, pp. 1253–1292, 2009.

[8] Object Management Group, "Object constraint language," Available at: http://www.omg.org/spec/OCL, 2014, version 2.4, formal/2014-02-03.

[9] J. Simmonds, R. V. D. Straeten, V. Jonckers, and T. Mens, "Maintaining consistency between UML models using description logic," *RSTI – L'Objet*, vol. 10, no. 2-3, pp. 231–244, 2004.

[10] D. Torre, Y. Labiche, and M. Genero, "UML consistency rules: A systematic mapping study," in *Proc. 18th Int. Conf. Eval. Assess. Softw. Eng.* ACM, 2014, pp. 6:1–6:10.

[11] D. Torre, Y. Labiche, M. Genero, and M. Elaasar, "A systematic identification of consistency rules for UML diagrams," Available at: http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-15-01.pdf, Carleton University, Tech. Rep. SCE-15-01, 2015.

[12] S. Sobernig, B. Hoisl, and M. Strembeck, "Extracting reusable design decisions in UML-based domain-specific languages: A multi-method study," submitted.

[13] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius, "Empirical evidence about the UML: A systematic literature review," *Softw. Pract. Exper.*, vol. 41, no. 4, pp. 363–392, 2011.

[14] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Sci. Comput. Program.*, vol. 89, Part B, pp. 144–161, 2014.

[15] E. Evans, *Domain-driven Design: Tackling Complexity in the Heart of Software*, 1st ed. Addison-Wesley, 2004.

[16] T. Mens and P. v. Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, 2006.

[17] S. Sobernig, B. Hoisl, and M. Strembeck, "Protocol for a systematic literature review on design decisions for UML-based DSMLs," Available at: http://epub.wu.ac.at/4467/, WU Vienna, Tech. Rep. 2014/02, 2015.

[18] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Inform. Softw. Tech.*, vol. 53, no. 6, pp. 625–637, 2011.

[19] S. Jalali and C. Wohlin, "Systematic literature studies: Database searches vs. backward snowballing," in *Proc. ACM-IEEE Int. Sym. Empir. Softw. Eng. Meas.* ACM, 2012, pp. 29–38.

[20] B. Hoisl, S. Sobernig, and M. Strembeck, "A catalog of reusable design decisions for developing UML/MOF-based domain-specific modeling languages," Available at: http://epub.wu.ac.at/4466/, WU Vienna, Tech. Rep. 2014/03, 2015.

[21] J. Bézivin and F. Jouault, "Using ATL for checking models," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 69–81, 2006.

[22] A. S.-B. Herrera, E. Willink, R. Zanzeebarr, A. Igdalov, C. W. Damus, and N. Boldt, "OCL/FAQ – Eclipsepedia," Available at: http://wiki.eclipse.org/OCL/FAQ, 2015.

[23] B. Kitchenham, "Procedures for performing systematic reviews," Keele University & National ICT Ltd., Joint Tech. Rep. (Keele University Tech. Rep., NICTA Tech. Rep.) TR/SE-0401, 0400011T.1, 2004.

[24] P. Langer, K. Wieland, M. Wimmer, and J. Cabot, "EMF profiles: A lightweight extension approach for EMF models," *J. Object Technol.*, vol. 11, no. 1, pp. 1–29, 2012.

[25] S. Schefer and M. Strembeck, "Modeling support for delegating roles, tasks, and duties in a process-related RBAC context," in *Proc. Int. Worksh. Inform. Syst. Secur. Eng.*, ser. LNBIP, vol. 83. Springer, 2011, pp. 660–667.

[26] S. Schefer-Wenzl and M. Strembeck, "An approach for consistent delegation in process-aware information systems," in *Proc. 15th Int. Conf. Bus. Inform. Syst.*, ser. LNBIP, vol. 117. Springer, 2012, pp. 60–71.

[27] C. Atkinson and T. Kühne, "Processes and products in a multi-level metamodeling architecture," *Int. J. Softw. Eng. Know.*, vol. 11, no. 6, pp. 761–783, 2001.

[28] ——, "A tour of language customization concepts," *Adv. Comput.*, vol. 70, pp. 105–161, 2007.

[29] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige, "The Epsilon book," Available at: http://www.eclipse.org/epsilon/doc/book/, 2015.

[30] E. Cariou, C. Ballagny, A. Feugas, and F. Barbier, "Contracts for model execution verification," in *Model. Found. Applicat.*, ser. LNCS. Springer, 2011, vol. 6698, pp. 3–18.

[31] T. Schattkowsky, J. Hausmann, and G. Engels, "Using UML activities for system-on-chip design and synthesis," in *Model Driven Eng. Lang. Syst.*, ser. LNCS. Springer, 2006, vol. 4199, pp. 737–752.

[32] J. Bergh and K. Coninx, "CUP 2.0: High-level modeling of context-sensitive interactive applications," in *Model Driven Eng. Lang. Syst.*, ser. LNCS. Springer, 2006, vol. 4199, pp. 140–154.

[33] N. Ubayashi, T. Tamai, S. Sano, Y. Maeno, and S. Murakami, "Model compiler construction based on aspect-oriented mechanisms," in *Gen. Program. Compon. Eng.*, ser. LNCS. Springer, 2005, vol. 3676, pp. 109–124.

[34] E. Soler, J. Trujillo, E. Fernández-Medina, and M. Piattini, "Building a secure star schema in data warehouses by an extension of the relational package from CWM," *Comp. Stand. Inter.*, vol. 30, no. 6, pp. 341–350, 2008.

Proposal for Improving the UML Abstract Syntax

Dan Chiorean, Vladiela Petraşcu, Ioana Chiorean
Babeş-Bolyai University, Cluj-Napoca, Romania
chiorean@cs.ubbcluj.ro, vladi@cs.ubbcluj.ro, ioana@math.ubbcluj.ro

***Abstract*—Different types of consistency of UML models have been described in the literature. The consistency of UML models with the UML static semantics, usually referred as well-formedness, and the consistency between different versions of the same model are among the most cited. The UML models' well-formedness is a precondition for any other kind of consistency, being described by means of the UML abstract syntax. Unfortunately, this abstract syntax specification is bogus. As a consequence, checking UML models' consistency is not a natural practice, as it should be. Beginning with 2000, there have been several papers reporting this state of facts, but without any visible consequences on the state of practice. In this paper, the authors propose a new approach (including specification techniques and a process) meant to overcome this specification drawback. Our proposal is based on a long practice of improving the specification of UML Well-Formedness Rules in the OCLE tool.**

***Index Terms*—UML model consistency, UML abstract syntax, Well-Formedness Rules**

I. Introduction

The purpose and relevance of the abstract syntax specification of MOF-based metamodels are explicitly mentioned in all OMG documents and acknowledged by the entire community of modelers. As stated in [11], "The abstract syntax defines the set of UML modeling concepts, their attributes and their relationships, as well as the rules for combining these concepts to construct partial or complete UML models." The same document claims that "Relative to UML 1, this revision of UML has been enhanced with significantly more precise definitions of its abstract syntax rules and semantics, a more modular language structure, and a greatly improved capability for modeling large-scale systems." Unfortunately, the state of facts does not live up to these claims. The paper [14] proposed by Wilke and Demuth at the OCL 2011 Workshop is a relevant proof of this, through both its title and contents, even though it has been written before [11]. The errors identified within the abstract syntax specification and the solutions proposed for some of them concern all UML and MOF releases, with no exception (see [1] - [8], [13], [14]). Although there have been enhancements in the abstract syntax specification from one version of the standards to the other, things could and should be seriously improved given the fact that: the reported errors, except compilation errors, have not been eliminated yet, new errors have emerged, the reading and analysis of the UML 2.x specification is much more tedious compared to its 1.x version. Thus, the understanding of a single 2.x well-formedness rule (WFR) generally requires a detailed analysis of several class diagrams and additional operations (AOs). The fact that the standard abstract syntax specifications fail to reach a stable and correct version is symptomatic and triggers the necessity of a change of attitude in writing such specifications. The approach proposed here is driven by the paramount importance of a complete, non-ambiguous informal specification, accompanied by relevant examples and by the need of a thorough validation of all specifications, based on adequate test cases. The technical aspects concerning the OCL specifications reported by previous papers, as well as the need of a testing-oriented OCL specification (meant to support an efficient error detection and diagnosis [5]) are also considered.

Within the abstract syntax specification, the role of the informal (natural language) descriptions is twofold. On the one side, they are used for detailing the structure described by class diagrams, by providing additional information concerning the concepts, attributes and associations involved. On the other, they describe the constraints that should be fulfilled by the modeling concepts, as well as the additional operations used for querying the model or needed in the specification of WFRs. This second role influences both the design of OCL specifications and their validation.

Unlike previous work on the topic, this paper introduces a natural approach concerning the specification of a static semantics. Although the term "approach" may seem a bit pretentious, we claim it is the most adequate, since it refers to a process in which the involvement of the OMG standards' authors is a must. The gaps and ambiguities residing in the standard specifications cannot be fixed in absence of their authors or in the absence of an explicit, unequivocal description, due to the risk of altering the original intentions. Our proposal takes into account the major differences among the 2.x and 1.x versions of the UML standard.

The reading of the UML 2.x specifications is more demanding and tedious as compared to the 1.x versions. This is due to the fact that most of the concepts are progressively described in several packages, their understanding requiring the investigation of various diagrams and associated textual descriptions [2]. The amount of newly-introduced concepts and their dispersed presentation are a strong argument towards the adoption of a complete, rigorous and clear description style, as proposed by this paper.

II. UML 2.x Abstract Syntax - From Goals to State of Facts

The primary purpose of an abstract syntax definition is that of providing a complete, non-ambiguous and rigorous description of a modeling language. These requirements are mandatory for both the correct understanding and consistent

use of the modeling language, and for enabling conformance verifications of user-models against it. Failure to fulfill these requirements triggers inability to fully validate models and the risk to encounter different interpretations of the same specifications. Moreover, it compromises the chance to ensure a safe and predictable model transfer among tools, as required by [11]: "One of the primary goals of UML is to advance the state of the industry by enabling object visual modeling tool interoperability. However, to enable meaningful exchange of model information between tools, agreement on semantics and notation is required."

Modeling languages share far more commonalities than differences with programming languages. Therefore, similar to programs, compilability is a mandatory requirement on models as well. However, the similarity among the two types of languages does not only involve compilability. Similar to programs, models should be executable, and the results should conform to the requirements [6]. The formal model specifications should be preceded by informal equivalents. The informal description should be complete and non-ambiguous, since, according to Kristen Nygaard "Programming is understanding". Therefore modeling, similar to programming, cannot be imagined in the absence of a thorough problem understanding. Mathematical proofs of specifications' correctness are only seldom realized; testing remains therefore the best alternative, at least in the current context. Similar to program development, ensuring model compilability is mandatory only at some key points of model development (usually, prior to transforming the models or prior to performing simulations). Such a requirement is best captured in [10]: " during model editing, the model will frequently be syntactically incorrect, and the tool needs to be able to allow for syntactical incorrectness in this mode.". Thus, it is easy to understand the reason why the static semantics is described exclusively by means of invariants (WFRs), without pre/post-conditions, as promoted by Design by Contract. Despite this, some of the additional operations encountered in UML have preconditions. We judge this practice as right and useful, since the additional operations are not employed exclusively in the specification of invariants. An AO being targeted to model navigation, the fulfillment of its precondition guarantees that its evaluation is meaningful, while the fulfillment of its postcondition ensures the correctness of the evaluation results. As regarding the result of a model compilability check, this should provide more than a simple yes/no message. In case of non-compilability, it is essential to be provided with meaningful information enabling efficient error diagnosing and allowing a real-time model adjustment.

The informal specifications included in the UML 2.x documents (for both WFRs and AOs) fail to comply with the quality requirements mentioned in the beginning of this section. This is a high-priority issue, in our view. In the last UML specification, 2.5, WFRs are compilable - a step forward compared to the previous specification, 2.4.1. However, the runtime testing and debugging issues are much more tedious than the compilability ones, as unanimously acknowledged by software developers. An important cause of the existing errors is the inappropriate informal specification. We will deal with these issues in Section 5 of this paper.

III. Related Work

Beginning with 2000, several papers have focused on the specification and usage of WFRs. In the following, we summarize the ones considered to be the closest to our approach.

In [13], the authors have given a first quasi-exhaustive analysis of the WFRs specified in UML 1.3. The work has focused on the Foundation::Core package (31 classes and 27 associations) that has been specified in USE, in order to check the corresponding 43 WFRs. Also, 28 Additional Operations were tested. Errors have been found in 39 out of 71 tested expressions. Four categories of errors have been identified: syntax errors, minor inconsistencies, type checking errors, and general problems. The paper was the first to draw an alarm with respect to the quality of the UML WFRs specifications. The following statement worth mentioned: "For future work we plan to extend the analysis to the complete UML metamodel including all of its wellformedness rules and making it available in USE. This might not only be useful for improving the state of the standard but also implies another very nice application: in principle, any UML model can be checked for conformance to the UML standard.". In [2], authors from the same research team present a similar analysis performed with USE, this time for the UML 2.0 Superstructure.

In [8], the second published paper on this topic, the authors claim having tested the entire set of WFRs specified in the context of the UML 1.3 metamodel. They report 450 errors of three kinds: non-accessible elements, empty names, and miscellanea. The proposed solutions for fixing the reported problems seem a bit bizarre. Namely, they suggest to "Take the empty names into account in every rule of the metamodel (296 errors). Consider access and contents as two different concepts (138 errors). Avoid two opposite association ends with the same name (18 errors)".

In [3], the authors present two techniques for checking UML models. One, implemented in Rational Rose, that enables to navigate and check the contents of the UML metamodel by means of an appropriate VBA specification, and the other by means of OCL AOs and WFRs. Some AOs and WFRs are analyzed both with respect to the identified bugs to the actions undertaken for correcting them. In [4], the same authors analyze different kinds of errors identified in the OCL specification. The focus is on proposing "good practices" meant to support "a correct, clear and efficient specification". The consistency among the formal and informal specifications, the clearness of OCL expressions, the fact that evaluating OCL specifications instead of only compiling them is imperative, are among the proposed and exemplified practices. The paper [5] is focused on describing OCL specification patterns intended to support a specification style targeted at an easier model debugging. In [6], the focus is on the similarities between programming and modeling languages. The paper emphasizes

the fact that, in the context of the model-driven paradigms, producing compilable models is a must, not an option.

[14] is focused on the study of UML 2.3 Superstructure WFRs. As acknowledged by its authors, there are many similarities between the topic and results reported in this paper and those of [2]. The differences concerns the metamodels (UML 2.3 in this last analysed paper and UML 2.0 in the previous case) and the tools employed (Dresden OCL toolkit in the last paper and USE in the other).

The common feature of papers published by the teams from Bremen and Dresden is their focus on the compilation phase. As regarding the papers published by our team, the analysis overpasses the mere compilation. The runtime results and their conformance to the informal specifications are also considered.

In [9], the authors present coherence rules grouped on metamodel elements and diagrams. Although the idea looks nice, there are some drawbacks. Firstly, the rules are presented exclusively in an informal manner (in spoken language); moreover, for some rules the semantics is not clear enough. Secondly, there are no comments about incorrect rules and about authors' proposal for improving the existent semantics.

Finally, in [7] and in some other papers on the same topic, A. Egyed presents "an approach for quickly, correctly, and automatically deciding when to evaluate consistency rules." As the title of the paper suggests, the author's work is focused on doing an automated quick evaluation. In the experience presented, only 24 rule were evaluated - some being WFRs and others defined by model designers. There are no mentions about the correctness of the evaluated rules. From this point of view, the approach is significantly different from ours. However, the author is convinced about the importance of consistency checking in case of UML models.

IV. The Proposed Approach for Specifying the Abstract Syntax

The state of facts in specifying the abstract syntax of UML, together with a thorough analysis of the published literature on the topic allow us to argue that a significant amount of all the existing specification errors are due to failure in obeying to a number of elementary requirements, validated by the software engineering practice. Given our experience in the field, we propose conforming to the following rules when specifying the abstract syntax of UML/MOF.

1) A complete and non-ambiguous informal equivalent of all OCL specifications (both WFRs and AOs) is the first and the most important of these rules. Moreover, there should exist a full conformance among the informal specification and its formal correspondent. It would be helpful if the informal specification would be accompanied (possibly in an attached document) by examples illustrating cases of validation and invalidation of each rule, as well as exceptional cases that may arise throughout the evaluation of AOs.
2) Runtime validation of formal specifications on significant data sets (models) is mandatory. Mere compilability is not enough.
3) The formal specifications of WFRs must be testing-oriented [5]. Accomplishment of this requirement supports an easy error diagnosing of models that do not comply with the WFRs in question.
4) Choosing the appropriate context for the specification of WFRs that refer to features of several metaclasses is also an important issue [6].
5) For both efficiency and clarity, the use of OCL specification patterns is recommended, whenever the case.
6) Indented and syntax highlighted OCL expressions enable an easier lecture of specifications. Prefixing the formal specifications with a short informal description of requirements (similar to comments in programming) is useful, as well.
7) The WFRs which are specified in an informal manner exclusively should be complemented by relevant examples of their fulfillment or failure to be fulfilled, possibly accompanied by an overview of how the authors imagine the validation process. Obeying to this requirement allows a better understanding of the rules and provides support in finding appropriate specification and validation solutions.

Except for the forth recommendation, which seems to be obeyed by almost all specifications of the UML, all the others are not met. As illustrated in the following, taking them into account will help in increasing the quality of the standard specifications.

V. Analyzing Import Relationships on UML Namespaces

The import in a namespace of elements from different namespaces is one of the most important relationships in both programming and modeling languages offering modular development support. This allows the imported elements to be directly referred by their name or by an alias, whenever there is no name conflict among the imported elements and the elements belonging to the namespace which performs the import. In case of conflict, the use of a qualified name is mandatory. The programming languages come with clear specifications concerning the import relationship. Thus, we argue that any differences in the import rules specified for UML and MOF compared to the programming languages should be clearly justified. Even more, examples are needed to illustrate the manner in which various cases tolerated by modeling languages can be coded in a programming language (direct engineering) or the reverse (reverse engineering). Otherwise, the support offered by modeling languages to the MDA, MDE and MDD paradigms remains only in statements.

Similar to programming languages, the UML allows two types of import:

1) *explicit*, by defining an individual import relationship for each imported element (which should be a `PackageableElement`). Such an import relationship is modeled by the `ElementImport` concept (direct descendent of `DirectedRelationship`), that has two attributes: `visibility` and `alias`. Graphically,

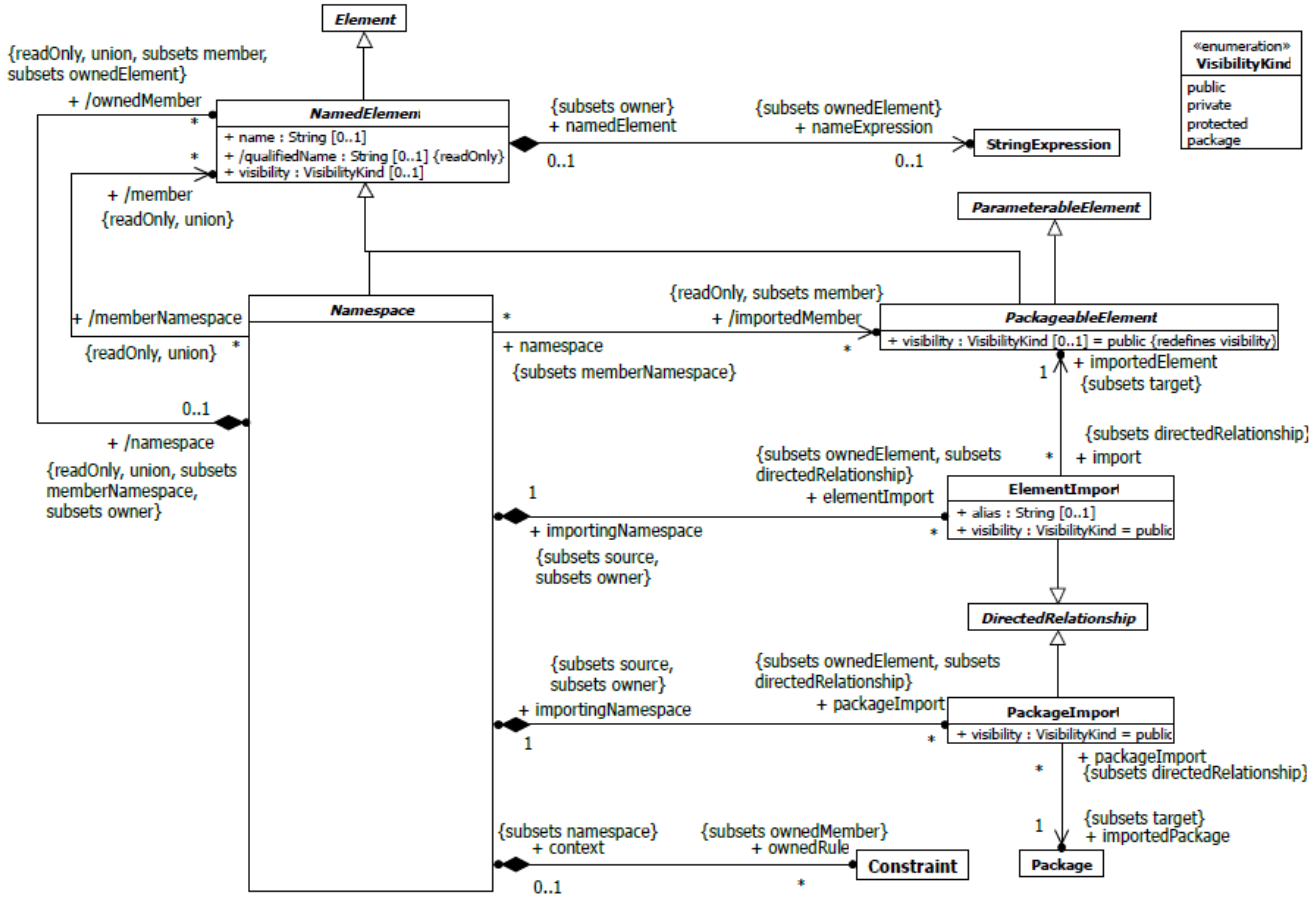

Fig. 1. The `Namespaces` diagram of the `Constructs` package (Figure 7.5 of [11])

`ElementImport` is represented by a "a dashed arrow with an open arrowhead from the importing namespace to the imported element. The keyword `import` is shown near the dashed arrow if the visibility is public, otherwise the keyword `access` is shown to indicate private visibility. If an element import has an alias, this is used in lieu of the name of the imported element. The aliased name may be shown after or below the keyword `import`."

2) *implicit*, by means of an import relationship among the importing namespace and the imported package. "Conceptually, a package import is equivalent to having an element import to each individual member of the imported namespace, unless there is already a separately-defined element import." Similar to the case of an element import, this relationship is modeled by the `PackageImport` metaclass, having an analogous graphical representation. It "is shown using a dashed arrow with an open arrowhead from the importing package to the imported package. A keyword is shown near the dashed arrow to identify which kind of package import that is intended. The predefined keywords are `import` for a public package import, and `access` for a private package import."

The concepts involved in the import relationships and their interconnections are illustrated in Figure 1. Even if not explicitly stated, the set `importedMember` is needed when computing the set of potential servers of a model element or when checking if an element is legally imported.

In the following, we will analyze the formal OCL specifications (and their informal descriptions) regarding the import relationship defined between a `Namespace` and a `PackageableElement` or a `Package`.

In the `Namespace` context, the AO `getNamesOfMember()` previously specified in the `NamedElement` context "is overridden to take account of importing. It gives back the set of names that an element would have in an importing namespace, either because it is owned; or if not owned, then imported individually; or if not individually, then from a package."

In our opinion, the last part of the second phrase (marked by the underlined words) is a bit confusing. A clearer and more explicit statement (at least for a non-native speaker) could be: either because it is owned, or imported individually or by a package import.

Following, there is the corresponding OCL specification, as

provided in the standard.

```
Namespace::getNamesOfMember(element:NamedElement):String[0..*];
getNamesOfMember =
 if self.ownedMember->includes(element)
 then Set{element.name}
 else let elementImports:Set(ElementImport)=self.elementImport
        ->select(ei|ei.importedElement=element) in
      if elementImports->notEmpty()
      then elementImports->collect(el|el.getName())
      else self.packageImport->select(pi|
       pi.importedPackage.visibleMembers()->includes(element))
       ->collect(pi|pi.importedPackage.getNamesOfMember(
                                              element))->asSet
      endif
 endif
```

In [11], the type returned by the observer defined is written as `String[0..*]`, notation not accepted in OCL 2.4 (the current specification [12]) and previous specifications. The AO `visibleMembers()`, used by composition in the above specification is bogus, as we will prove in the following. As a consequence, the result returned by the AO `getNamesOfMember()` will be incorrect in some cases.

In the `Package` context, the query `visibleMembers()` identifies those members of a `Package` that can be accessed outside it. The specification provided for the AO visibleMembers() in [11] is:

```
body: member->select(m|m.oclIsKindOf(PackageableElement)
      and self.makesVisible(m))->
          collect(oclAsType(PackageableElement))->asSet()
```

Analysing figure 12.1 Packages, pp. 29 from [11] we wonder why the OMG has not proposed: `self.packagedElement->select(pe | self.makesVisible(pe))`. In this case, the condition `oclIsKindOf(PackageableElement)`, the cast at `PackageableElement`, and the conversion `asSet` are redundant.

The query `makesVisible()` specified itself in the `Package` context, "defines whether a `Package` makes an element visible outside itself. Elements with no visibility and elements with public visibility are made visible.".

```
Package::makesVisible(el:Namespaces::NamedElement):Boolean;
pre: self.member->includes(el)
makesVisible =
  -- the element is in the package
  (ownedMember->includes(el)) or
  -- it is imported individually with public visibility
  (elementImport->select(ei|ei.visibility =
  VisibilityKind::public)->collect(importedElement.oclAsType(
        NamedElement))->includes(el)) or
  -- it is imported through a package with public visibility
  (packageImport->select(pi|pi.visibility =
  VisibilityKind::public)->collect(pi|
  pi.importedPackage.member->includes(el))->notEmpty())
```

As regarding this specification, there are some things we would like to analyze.

Firstly, the informal specification states that "Elements with **no visibility** and elements with **public visibility** are made visible." As no explanation is offered regarding why elements with no visibility are visible outside the package, this requirement seems strange to us. Especially since in programming languages (Java, for instance) only public members of a package can be explicitly exported or referred by their qualified name outside the owner package. That is why, at the beginning of this section, we have emphasized the necessity of an explicit description of the rationale behind certain decisions.

Secondly, it is easy to notice that the formal specification does not comply with the informal one, since only the visibilities of `elementImport` and `packageImport` are considered, without taking into account the visibility of the element itself (irrespective if it being owned by the package, imported individually or by means of a package import). Moreover, the authors of the OMG specification say nothing with respect to what happens in particular cases, such as the one in which the same element is imported both individually (with `visibility = VisibilityKind::private`) and by means of a `packageImport` with `visibility = VisibilityKind::public`. In this case, evaluating the specification above, `makesVisible` will be evaluated to true, even if it is stated that the individual import has priority compared to package import.

In order to exemplify our reasoning, let us consider the model shown in Figure 2. As illustrated there, between packages `P2` and `P1` there is an `<<import>>` relationship. The package `P1` owns the class `A` having `visibility = VisibilityKind::private`. In the context of the `P1` package, we are interested to see if `el = A` is visible outside its owning package. The class `A` is a member of `P1`, so we have to evaluate the `makesVisible()` AO. `P1.ownedMember->includes(A)`, therefore `makesVisible() = true`. Thus, due to the `packageImport` relationship, `A` is added to the `P2` namespace and can be accessed by name, in case there are no name collisions between `A` and other elements of the `P2` namespace. This result is incorrect, since the element in question has private visibility.

Thirdly, when the element is imported in the package by `packageImport`, the result returned will be wrong if the visibility of the element transmitted by parameter (`el`) is `VisibilityKind::private`. A sample situation is illustrated in Figure 3. In the package `P2`, the private class `A` is imported by the `importPackage` relationship stereotyped `<<import>>`. In `P3`, private class `A` is imported by means of the `importPackage` relationship, also stereotyped `<<import>>`, between `P3` and `P2`. In the OCL specification of `Package::makesVisible` (see above), this corresponds to the OCL expression following the second **or** (imported through a package with public visibility). Similar to the previous case, the result is wrong, due to the visibility of class `A`.

Concluding, we notice that even though the `makesVisible()` AO is compilable, its returned results do not fully comply with the informal specification. Even more, the usefulness of **elements with no visibility** has not been explained and taken into account in the formal specification. Apart of these, there could be particular cases, like those mentioned above, when the results are debatable.

A possible solution would be to include in the precondition the restriction regarding the visibility of `el`.

Fig. 2. Import of a private class through a single package import

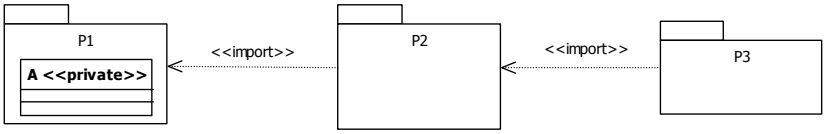

Fig. 3. Import of a private class through double package import

```
Package::makesVisible(el:Namespaces::NamedElement):Boolean;
pre: self.member->includes(el) and
     el.visibility = VisibilityKind::public
```

or

```
Package::makesVisible(el:Namespaces::NamedElement):Boolean;
pre: self.member->includes(el) and
     (el.visibility=VisibilityKind::public or
      el.visibility.oclIsUndefined
     )
```

Since we have no idea about the semantics of `el.visibility.oclIsUndefined` in this case, and due to other particular cases, our opinion is that the first thing to do is to clarify the informal specification.

In a namespace, a `NamedElement` is valid if it is distinguishable from any other element owned by the namespace [11], [6]. The WFR checking this requirement uses the AO `isDistinguishableFrom(p1,p2)`. This operation is firstly defined within the `NamedElement` context, and redefined in the `BehavioralFeature` context. As stated in the [11] (pp. 73), "...By default, two named elements are distinguishable if (a) they have unrelated types or (b) they have related types but different names."

```
context NamedElement::isDistinguishableFrom(n:NamedElement,
                                            ns:Namespace):Boolean
def: isDistinguishableFrom(n:NamedElement,
                           ns:Namespace):Boolean =
 if self.oclIsKindOf(n.oclType) or
    n.oclIsKindOf(self.oclType)
 then ns.getNamesOfMember(self)->intersection(
      ns.getNamesOfMember(n))->isEmpty()
 else true
 endif
```

The formal specification fully complies with the informal requirements. However, stating that two elements having unrelated types are distinguishable could cause unpleasant situations, such as the one in which a package contains both a class and an enumeration having the same name (e.g `Test`). In this case, `enumeration.oclIsKindOf(class.oclType)` and `class.oclIsKindOf(enumeration.oclType)` are always evaluated to `false`, irrespective of the `enumeration` instance of the metaclass `Enumeration` and `class` instance of the metaclass `Class` (see Figures 4 and 5). By consequence, the type of an attribute `testKind:Test` belonging to a different class of the same package is uncertain. This is due to the fact that we cannot distinguish between these two types having the same name. In order to fix this bug, the simplest solution would be to remove "part a)" from the above requirements. However, the best decision would be to provide a clearer specification, including suggestive examples of models with both distinguishable and not distinguishable named elements.

The `visibleMembers()` AO analyzed at the beginning of this section is used by composition in computing the set of elements imported in a namespace.

"The `importedMember` property is derived from the `ElementImport`s and the `PackageImport`s."

```
importedMember = self.elementImport.importedElement->asSet()
  ->union(self.packageImport.importedPackage->collect(p |
  p.visibleMembers())->asSet())
```

Here, the drawback is due to previously-discussed `visibleMembers()`. Another problem is that both in case of a direct `elementImport` and `packageImport`, we have to consider also the elements having `visibility = VisibilityKind::private` and marked with the stereotype `<<access>>`, not only those marked with `<<import>>`. That is why, the importedMember in the `P3` namespace of Figure 6, will return `Set{A, C, B, D, E}`, so the `visibleMembers()` has a negative influence by means of `<<access>>` `packageImport` also.

Since in case of name-clashes the imported elements can be referred only by means of their `qualifiedName`, let us take a short look at the derived attribute `qualifiedName`, specified in the `NamedElement` metaclass. In the standard it is stated that this attribute "is constructed from the names of the containing namespaces starting at the root of the hierarchy and ending with the name of the `NamedElement` itself." The specification of the `qualifiedName():String` AO is:

```
body:
 if self.name<>null and self.allNamespaces()->select(ns|
                                  ns.name=null)->isEmpty()
     then self.allNamespaces()->iterate(ns:Namespace;
          agg:String = self.name |
          ns.name.concat(self.separator()).concat(agg))
     else null
 endif
```

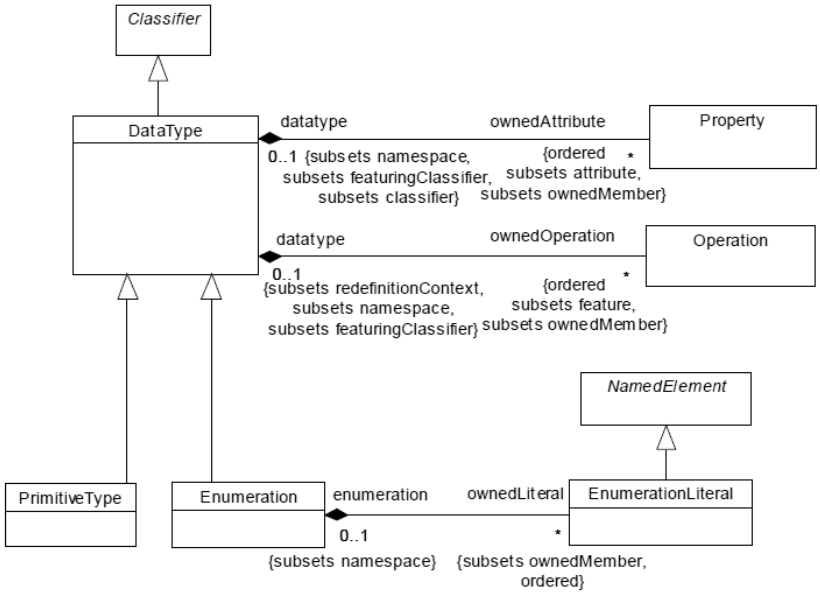

Fig. 4. The classes defined in the DataTypes diagram - from [11], Figure 11.18

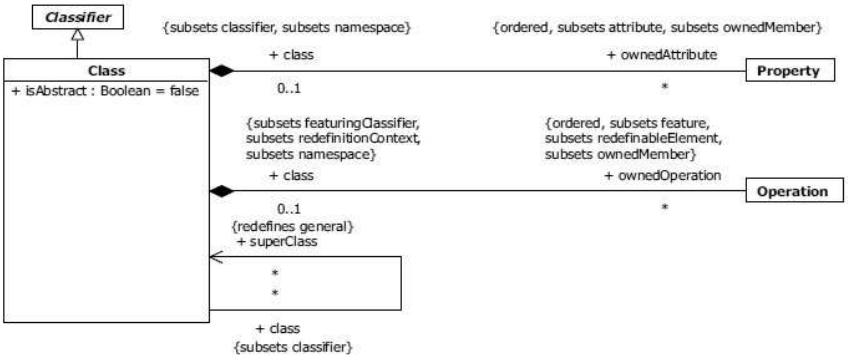

Fig. 5. The Classes diagram of the Constructs package - from [11], Figure 11.15

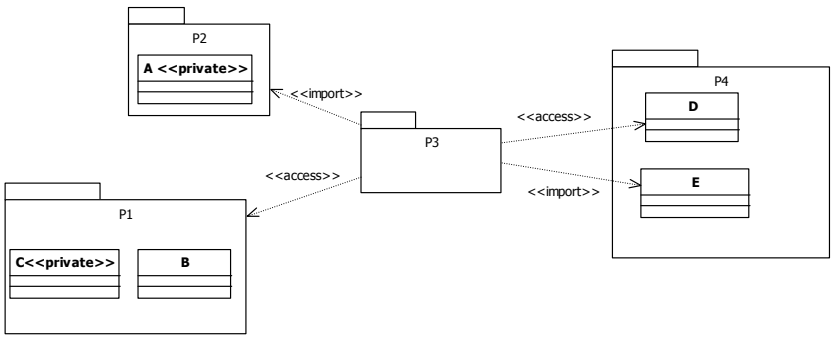

Fig. 6. Package import

While this specification is compilable and roughly correct, there can arise a little problem, because an empty `String` is not forbidden as a name value, and `self.''->notEmpty() = true`, the value computed for `qualifiedName` in such cases, is meaningless. That is why, such cases must be forbidden. More general, we consider that rules similar to those applied in programming languages must be used in modeling language as well.

The aspects analyzed in this section prove that, even in simple cases, the specifications must be realized carefully, not in a superficial manner. The reccomendations made in the previous section must be taken into acount.

VI. Conclusions

The purpose of this paper has been to propose a change of attitude with respect to the definition of the UML's abstract syntax, expected to positively affect the quality of the standard specifications. This improvement is a "sine qua non" condition for attaining the target of model-driven technologies and paradigms.

Our proposal is argued by means of meaningful examples taken from the latest UML specification [11]. The first requirement to be accomplished concerns the quality of the informal specifications: they have to be complete, accurate and clear. Once this precondition is accomplished, the associated postcondition is that the formal OCL specifications must fully conform to their informal equivalents. Our experience has proved that this conformance is achievable through an iterative process. The results obtained by evaluating the formal specifications must be compared to the informal ones and should trigger a synchronization among the two, if needed. This is an important contribution through which the OCL specifications may increase the quality of abstract syntax definitions, in general.

Another important message is that the mere compilability of formal specifications does not value much if these specifications are not validated on comprehensive models. Technical aspects related to the particularities of the specification language and the support that the formal specification style brings in achieving compilable models are important as well. In this respect, we recommend the adoption of a testing-oriented specification style, as introduced in [5].

Apart from the advice related to the specification style, all the others have been validated in Software Engineering. That is why, noticing that so much good specification practice has not been considered comes as an unpleasant surprise.

Achieving a good specification of MOF-based languages is a tedious process, requiring a quality feedback both from scientists and users. Our intent has been to make a first step, by proposing a set of "good practices" to be considered in the process, as well as a number of examples supporting our proposal. Hoping that our proposals will be analyzed, improved and extended by the OMG and thus a better abstract syntax specification will support a more efficient and widespread usage of modeling languages.

References

[1] Bauerdick, H., Gogolla, M., Gutsche, F. - Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. - In Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J., eds.: UML 2004 - The Unified Modelling Language. Volume 3273 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2004) pp. 188-196

[2] Fabian Bttner and Martin Gogolla - On Generalization and Overriding in UML 2.0 - in UML'2004 Modeling Languages and Applications. UML'2004 Satellite Activities, Springer 2004

[3] Chiorean, D., Carcu, A., Pasca, M., Botiza, C., Chiorean, H., Moldovan, S. - UML Model Checking in Studia Informatica vol XLVII (2002) pp. 71-88

[4] D. Chiorean, A Carcu, C Botiza, etc. Ensuring UML models consistency using the OCL environment - Electronic Notes in Theoretical Computer Science - ENTCS/102, 2004, pag. 99-110, http://dx.doi.org/10.1016/j.entcs.2003.09.005

[5] D. Chiorean, V. Petrascu, I. Ober. Testing-Oriented Improvements of OCL Specification Patterns. In Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics - AQTR. Volume II, pp. 143-148. IEEE Computer Society, 2010

[6] D. Chiorean, V. Petrascu. Towards a Conceptual Framework Supporting Model Compilability. In Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010). Volume 36(2010), ECEASST

[7] Alexander Egyed, Automatically Detecting and Tracking Inconsistencies in Software Design Models, In IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 188-204, 2011.

[8] J. M. Fuentes, V. Quintana, J. Llorens, G. Genova, R. Prieto Diaz. Errors in the UML metamodel? ACM SIGSOFT Software Engineering Notes 28(6):3-3, 2003.

[9] Hugues Malgouyres, Jean-Pierre Seuma-Vidal, Gilles Motet, Regles de coherence UML 2.0 - Version 1.1 - INSA - Toulouse, online at: http://www.lesia.insa-toulouse.fr/UML/CoherenceUML_v1_1_100605.pdf

[10] Michael Moors - Consistency Checking; Rose Architect, Spring Issue, April 2000, http://www.therationaledge.com/rosearchitect/mag/index.html

[11] Object Management Group (OMG) - OMG Unified Modeling Language (OMG UML) Version 2.5, 2015, http://www.omg.org/spec/UML/2.5/PDF

[12] Object Management Group (OMG) - Object Constraint Language version 2.4 - formal/2014-02-03, http://www.omg.org/spec/OCL/2.4

[13] M. Richters, M. Gogolla. Validating UML models and OCL constraints. In Evans et al. (eds.), UML 2000 The Unified Modeling Language. Advancing the Standard: Third International Conference Proceedings. Lecture Notes in Computer Science 1939, pp. 265-277. Springer, 2000.

[14] Claas Wilke and Birgit Demuth - UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure - paper proposed at: OCL 2011 Workshop