Appended paper D

Multi-Viewed Components

Magnus Persson, Frédéric Loiret, Jonas Westman, DeJiu Chen, Martin Törngren, Matthias Biehl

Technical report, Department of Machine Design, KTH, Stockholm, 2013,
TRITA MMK 2013:08, ISSN 1400-1179, ISRN/KTH/MMK/R-13/08-SE
Abstract

In this report, we present a concept for generic multi-viewed component models based on modular and (de)composable viewpoints. By decomposing systems not only along component but also viewpoint borders, a family of multi-viewed component models is proposed. Semantic relations between the viewpoints are modeled in order to be able to ensure e.g. consistency between separate but semantically partly overlapping views. We also characterize different types of relations between viewpoints in order to see how the choice of viewpoints in the component model influences the complexity of multi-viewed components. Finally we provide an example and describe how the concepts presented can be applied in practice.

Notation

Special notation for this report is briefly introduced below. In general, abstract concepts are given a calligraphic script (e.g. \( C, V \) or \( S \)), normal script lower case entities denote atomic data, upper case entities denotes composite entities (e.g. \( C, M \) or \( V \)) containing several other subparts of information. Bold font is used to denote sets of both atomic and composite entities.

\[ \begin{align*}
\cup & \quad \text{disjunct union} \\
\oplus & \quad \text{heterogeneous composition, i.e. for views of different viewpoint(s)} \\
\otimes & \quad \text{homogeneous composition, i.e. for views of the same viewpoint(s)} \\
\ltimes & \quad \text{arbitrary relation} \\
\models & \quad \text{models operator (i.e. abstracts)} \\
\emptyset & \quad \text{empty set} \\
\exists & \quad \text{exists} \\
\top & \quad \text{top element (index)} \\
\bot & \quad \text{bottom element (index)} \\
\setminus & \quad \text{set complement} \\
\in & \quad \text{is part of} \\
\forall & \quad \text{for all} \\
\rightarrow & \quad \text{semantically maps to} \\
\equiv & \quad \text{is defined as} \\
\perp & \quad \text{is orthogonal to (operator)} \\
\{1,2,3\} & \quad \text{set} \\
\langle1,2,3\rangle & \quad \text{tuple} \\
\langle1,2,3\rangle & \quad \text{sequence (ordered)} \\
0 & \quad \text{minimal element (index)} \\
a & \quad \text{set of attributes} \\
a & \quad \text{attribute} \\
A,B,C & \quad \text{component instances} \\
A_0,B_0 & \ldots & \text{trivial component, i.e. component without subcomponents or any attached views} \\
C_1,C_2 & \quad \text{subcomponents to \( C_0 \)} \\
C_\bot & \quad \text{component without views} \\
C_T & \quad \text{top component, i.e. with all views} \\
C & \quad \text{the group of subcomponents belonging to the top-level component \( C \)} \\
c & \quad \text{component model} \\
d & \quad \text{direction (for ports)} \\
E & \quad \text{edge (in a model)} \\
E & \quad \text{set of edges} \\
e & \quad \text{event (in a mode machine)} \\
g & \quad \text{mode group} \\
g & \quad \text{set of mode groups} \\
i,j & \quad \text{iterators} \\
m & \quad \text{mode} \\
m & \quad \text{set of modes} \\
M & \quad \text{model} \\
M & \quad \text{meta-model or other abstract syntax} \\
\overline{M} & \quad \text{complete transformation over the pivot meta-model \( M \).} \\
n & \quad \text{an arbitrary natural number} \\
n & \quad \text{name} \\
N & \quad \text{node (in a model)} \\
N & \quad \text{set of nodes} \\
n & \quad \text{an (arbitrary) property of a model}
\end{align*} \]
D.1 Introduction and Problem Illustration

The growing complexity of embedded systems poses a significant challenge to embedded software developers. Both product and process complexity are contributing to the challenge. Process complexity occurs from e.g. organizational constraints and tools used for development. At the same time, industrial challenges and scenarios put strong requirements on interoperability between different subsystems, on overall verifiability (around 50% of embedded software development costs can be spent on V&V activities), and on promoting concurrent design and modular design, in order to make it possible to distribute development efforts in large-scale systems geographically and organizationally. This is exemplified by the automotive sector, where development is distributed across multiple companies and countries [93, 96].

Since many of today's embedded systems are being developed by development teams with heterogeneous competences rather than singular developers or homogeneous development teams, system development will involve many different tools, each suited for a specific purpose and to be used by specialists. To make such tool chains be modular but still hold together is one of the main challenges of the embedded systems community today. [21, 22, 67, 91, 94, 98] Such modularity can come through different approaches, for example dividing the system by system borders (i.e. into components), or by different technical domains (i.e. into different viewpoints, as illustrated by figure D.1).
We refer to the use of multiple system descriptions as multi-view modeling, as inspired by the ISO 42010 standard [53]. A challenge is to build such descriptions in such a way that they inherently support system modularity. This implies handling that systems are described separately and thereafter composed, causing unintended side effects to occur. A common cause of non-expected integration side effects is the existence of emergent properties, i.e. properties of the system that are not easily understandable based on its constituent components. Emergent properties are usually different kinds of extra-functional properties of the system, such as for example timing, performance or safety. Such extra-functional properties are usually not explicitly modeled nor analyzed.

In a way, usage of multi-view modeling are an inherent property of today’s development environments, with multiple development teams, diverse tool environments and engineering competences, and possibly also development work distributed over multiple organizations. This diversity is a necessity due to the system properties, but also provides a source for significant integration problems [37, 81]. In practice this diversity is a challenge, since it becomes necessary to ensure that information stored in different modeling and development environments, i.e. different views, is kept consistent and coherent as efficiently as possible throughout the lifecycle of the system. This consistency can be semantic and/or syntactic. We informally define consistency as the absence of information conflicts, either within a view or within a group of views.

Several development approaches have been suggested to handle the increasing complexity in embedded systems, including architecture design [52, 53], component-based [28, 86] and model-based [93] development. Unfortunately, these approaches to a large extent have developed in parallel [92], and are also each represented by several fundamentally different solutions, giving a large group of potentially applicable formalisms for the development work, but only with loose coupling to each other.

**D.1.1 Elaboration of Problem and Requirements Analysis**

A basic assumption during multi-view modeling is that the different views are consistent. The assumption may not need to be met at every single point in time and may only apply to certain groups of views. Consistency between views relies on a simple basic (but not necessarily fulfilled) criterion: each view is internally consistent. We demonstrate this challenge with a simple example: Consider a real-time system where components have three main views: A structural one, a behavioral one (e.g. code), and a timing-related one (including e.g. worst-case execution time). Since there is a need to ensure consistency between the different views, it is necessary to ensure for example that the structural view is consistent with the behavioral one (same components and same way of composing the ports) or that the behavioral and timing view are consistent (i.e. the worst-case execution time given in the timing view is valid and not for example outdated). This view synchronization needs to happen even if these separate views are stored separately. Further challenges include handling effect of composition of different parts and aspects of the system, especially for extra-functional properties of the system.

The example, the discussion above and other sources [43] highlight common requirements in multi-view modeling. The following requirements apply to a prospective component model modularized through separate (de)composable viewpoints:

- Explicit semantic consistency handling between views of the same component. The handling may be manual or automatic, but basic support for consistency handling
needs to be integrated into the component model.

• Being able to handle scalability issues as the size of the modeled system and/or the number of applied viewpoints. There should specifically be room for future extensibility e.g. later adding additional views and viewpoints to an already existing system model.

• Composability of relevant properties in the components and viewpoints is one prerequisite in achieving such scalability – without it, it is not possible to ensure internal properties of components being preserved under composition. It also provides clearer division of concerns.

• Incremental rollout – it should be feasible to roll out the approach in an incremental manner, such that a “big bang” implementation strategy where a tool is rolled out instantly in the entire development organization does not become necessary. Many larger organizations don’t see that as a feasible or even wanted option.

• Modeling environments will be heterogeneous and multi-formalism. The modularization needs to be designed for translation between distinct, but partly overlapping descriptions of the same system, that are expressed in different formalisms.

The problem of finding a good way of splitting a system in viewpoints is an old one, mentioned already in one of the first papers on viewpoints [69] and was approached already before the concept of viewpoints was formulated [68]. There are several ongoing projects [9, 99] that have targeted or is targeting the tool integration issue in embedded software development and hence also semantic consistency between different viewpoints, both for existing and upcoming system data/component models. The approaches of these projects have been slightly differing, including:

• Using point-to-point tool integration, typically supplying only the transformation and not explicitly taking the need for structural and/or semantic transformation into account.

• Trying to come up with a single overall shared meta-model.

• Some hybrid approach, usually coming up with a relatively small core set of concepts, and making other concepts part of optional extensions, which are usually chosen only based on established design patterns(such as using several abstraction layers and separating different concerns more informally), and not on any type of formal analysis of the relations between different viewpoints. Examples for this are EAST-ADL, OSLC or iFest.

The first two approaches do not scale well, as illustrated in figure D.2. Point-to-point solutions do not scale since they imply a number of transformations in $O(n!)$. The second approach does not have that issue. The number of transformations scales by $O(n)$, however, it has a different problem unless major overlap between the different formalisms is assumed. Although the number of transformations is in this case in $O(n)$, this is a simplified view of the complexity growth: The complexity of the shared meta-model itself grows as well, since (1) it will cover a larger and more diverse number of concepts, and (2) it is likely that each of the transformations gets more complex with each added meta-model as well (since the expected distance between a meta-model and the shared meta-model is likely to grow with the number of involved meta-models is). [50, 85] The complexity growth of both the point-to-point approach and the single shared meta-model are partially caused artificially, due to an assumption about the tool environment that does not generally hold:
That transformations, preferably semantically complete, should be feasible between any pair of formalisms. To put the requirement this way is stricter than reality, thanks to many of the interactions being localized. Real-world tool chains usually have a certain degree of pipeline structure, employing loops and chains as common design patterns [19]. Further in many places, only a partial semantic translation is really needed.

The key to making a solution that scales to larger numbers of views, even custom ones, lies in performing a good implementation of the third approach. This however raises the question about which principles should be used for the modularization, since it could be done in different ways. There is a need to handle multiple views both along subsystem boundaries and along technical domain boundaries [62]. Several proposed component models have already introduced the concept of views, but without explicitly and formally considering the relations between different viewpoints. These component models, however, do not efficiently deal with the challenge of providing consistency between the different views due to syntactic and semantic data entanglement between them. Entanglement of data in different viewpoints makes composition of viewpoints potentially having non-composable effects on properties and prone to emerging effects. Further, some of these component models only allow pre-defined formalisms in views, effectively hindering the usage of other, possibly more expressive or otherwise useful formalisms.

D.1.2 Contribution and Scope

The explicit aim of this report is to find well-founded and formalized principles for building component models that are modularized into (de)composable views, in such a manner that information consistency between views is ensured and hence also along tool chains in a predictable and analyzable manner. We have applied the reasoning in a general manner, i.e. not for a specific set of viewpoints but for arbitrary combinations. It is not the aim of this report to build a concrete set of viewpoints that is only applicable in a certain development context, but to provide general principles and guidelines that are applicable in different contexts.
development contexts. This report presents a number of key contributions as listed below:

- It constitutes a step towards merging concepts from the model-based and component-based design paradigms, by considering composition and composability not only along subsystem/component borders but also along viewpoint borders.

- The concept of viewpoint decomposition is introduced, making it possible to handle views as combinations of composable subviews. Completely overlapping concepts will, thanks to this semantic modularization, only be represented once, either in a shared meta-model, or in a respective extension.

- The principle of explicitly describing the relations between the content of viewpoints in a formalized manner, forming a semantic lattice of viewpoints. This helps the designer of the view framework understand the semantic relations between different viewpoints and get an understanding of what constitutes “well-formed” design of view frameworks.

- Finally the report provides a formalization of the concept of composability for both properties of components and viewpoints.

View frameworks [1, 65] need to explicitly take into account the needs of synchronization of information between views. By providing sound rules for the interaction between different views, it becomes possible to "construct" a suitable component model from a number of modular and (de)composable viewpoints.

By providing sound rules for the interaction between viewpoints, it becomes possible to have separate viewpoints that are made consistent with each other only depending on the relevant shared part(s) of the meta-model, e.g. structure only, and that become composable. The approach is illustrated in figure D.3. It is important to note that views may still have partly the same coverage. Hence information sharing cause the different viewpoints not to be completely orthogonal, but rather have relations each other. These relations can be more easily managed with the concepts proposed in this report.

The structure of the report is as follows:

- Chapter 2 gives background on key concepts from model- and component-based design needed to understanding the starting point of this report.

- Chapter 3 describes the formalization of multi-view components proposed in this report.

- Chapter 4 covers the description of an applied example, built around a windshield wiper system.

- Chapter 5 discusses different ways to apply the formalization.

- Chapter 6 covers related work.

- Chapter 7 summarizes, concludes and discusses venues for future work.

**D.1.2.1 Delimitations**

Although the same problems to a lesser degree exist also in systems where the main emphasis is on hardware (e.g. mechanical or electronic), the nature of these areas tend to have a larger emphasis on prototype building as part of the development process, and hence have an additional safeguard against design conflicts as those mentioned in section D.1. Multi-view modeling problems are less pronounced in these systems and hence not the
focus of this report. Mechanical systems typically also have a lower number of potential dependencies between its subsystems, based on their physical design.

The example given towards the end of the report is not intended to show the full potential of the approach, or serve as a specification of actually useful definitions of viewpoints, but rather to illustrate the principles defined in this report and make them more understandable.

The focus of this report is further mainly on components with an at least partially formal definition of their component model, and mainly those where quantitative description of extra-functional properties (behavior, timing, reliability, safety, . . . ) is feasible. Many currently used component models are not at this level, but rather work as “code wrappers” of either source code or compiled code.

The proposed formalization in this report is intended to be applicable regardless of development process. This means that no assumption on the applied viewpoints has been made, for example through assuming a specific set of used viewpoints (the viewpoints and the process influence each other) nor any assumption in which order the views are constructed.
D.2 Preliminaries

To support the following sections we here provide background knowledge on concepts from architecture, model-based and component-based design.

D.2.1 Architecture Design: Viewpoints and Views

In system architecting [52, 53], views are used to describe different aspects of a system. Viewpoints are specifications guiding the construction and use of concrete views, as given by the following definitions:

- **view**: A representation of a whole system from the perspective of a related set of concerns.
- **viewpoint**: A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

Views are mainly a way to reach separation of different concerns. It is important to note that they all cover the same system architecture, i.e. viewpoints should, at least in theory, be sufficiently consistent with each other as they are all models of the same real-world system(s) . There is however no standardized way to ensure view consistency. Even though ISO 42010 [53] does mention the need to synchronize view contents, it gives no detailed guidance on how to do it or what kind of tools are useful in the process. Informally, consistency is the absence of contradictions between different views, i.e. the views together describe a realizable system.

The choice of viewpoints, and hence also which properties of the systems that are possible to model, is a key step of system development. Viewpoints that can be commonly relevant may be e.g. structural viewpoints, real-time and performance viewpoints, and allocation viewpoints. A couple of concrete examples of simple viewpoints are given later in the windshield wiper example in section D.4. The possibility to extend system descriptions with additional views and viewpoints is a vital need. Of course custom viewpoints can be used together with architecture frameworks or multi-viewpoint libraries, but many different proposals for more or less formalized standard sets of viewpoints have been suggested. Few of these proposals give sufficient formalization of the views as such, in order to be able to ensure view consistency, and some even consider views as simple subsetting of the full system model. Unfortunately, this implies creating a large all-encompassing system model, which is only possible for smaller systems. To achieve such consistency, a sufficiently formal definition of both the syntax and semantics of the views in question are required.

D.2.2 Model-Based Development

Model-based development [32, 83] is an approach to system development putting models as the central concept. Models are abstractions of the system to be built, and a wide variety of different model-based approaches exist. [73] Some try to build formal models that are used to prove certain system properties, other focus on organizing data about the system in a suitable manner, and yet other are built in order to effectively and efficiently build executable code, either for simulation or parts of the system itself.
One common approach is based on the paradigm of building custom-built modeling languages, either separate domain-specific languages (DSLs) or as a specialization of generic languages such as UML. Most of these approaches build on the concept of meta-modeling, the process of building meta-models to define modeling languages. Typically, a hierarchy of meta-modeling formalisms is used, like for example the OMG 4-level one where the top-level, meta-meta-modeling languages, refer to languages that are sufficiently expressive to model themselves. Model transformations [16] are programs used to translate between different models to each other.

D.2.3 Components and Component Models

Component-based design is a commonly suggested solution to several issues in the software engineering community. The often touted vision is that software-based systems will be as easy to build “as Lego”, from a cognitive point of view making it easier to reason about the system. Part of this vision is that previously developed software components are to be reused in new contexts, which makes this vision harder to achieve in practice - as extra-functional concerns and variants of functionality will differ between different deployment environments. With this introduction, we define the term “software component”:

• “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [89]

This definition is partially contended [27, 29], however for our purposes, it is sufficiently good. It is also abstract - however, based on the flora of approaches to components available in the wild, it is hard to make a more concrete one. Still, due to the contention of the definition, we propose a slight variant of the classical Szyperski definition that will better suit how we use the term in this report:

• “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only among the aspects covered by its component model. A software component can be deployed independently and is subject to composition by third parties.”

The slight difference to the Szyperski definition is important to note, and deals with the main criticism of the definition: There are always some aspects that a component model does not (fully) handle. If for no other reason due to computational intractability, exact component models will not be possible to use in any reasonably sized system. Hence, we introduce the vital delimitation that component models only handle certain aspects. With a classical model-based design wording, the components are an abstract representation of the corresponding real system, and only certain aspects of the real system carry over to the models. Further, such models may have different levels of detail.

As there are many types of components available, it is unfortunately very obvious that all components are not possible to compose with each other. Typically, components that are supposed to be usable together are both based on some common basis, i.e. the same principles for building components and assembling them with each other. Such a basis is called a component model.

We will use this definition in this report, although the delimitation is not exclusively to software components, but rather to components merely in systems with a substantial software part. The definition is however easily adaptable also to hardware components in that context.
• “A component model specifies the standards and conventions that components must follow to enable proper interaction.” [21, section 10.2]

A further important concept in the world of the components is composition, i.e. how different subcomponents can be assembled. How composition is defined in a component model is guided by a composition model, i.e. the description of the composition operation acting on the components, for example when composition is valid, and what the result is. The composition model defines how components can be composed and the semantics of the composition, i.e. the principles in which components are composed to larger systems. In particular, this implies also having model abstractions for what happens with the components’ extra-functional properties, if they are a part of the component model. Composition may be defined both in a more hands-on approach by merging the code in a certain manner, or analytic, i.e. doing analysis on the components. [61]

A property is composable if system integration will not invalidate this property, once the property has been established at the component level (adapted from [58]). Composability is a basic assumption of component-based development, but if not explicitly handled by the component model, especially extra-functional properties of components may sometimes not compose in a composable manner. In other words, a property of a component model is non-composable if there are possible unexpected (i.e. unanalyzable) emerging effects of adding new components.

There are many types of component models, and it is not an intention of this publication to describe all of them. Still, component models can be built according to quite different principles, and with quite different goals. Some are focused mainly at providing benefits during the development time, while others are useful even during runtime. There are component models that are built both around source code and compiled code, and even others that cover neither, and rather focus on earlier development stages and provide simulatable models at a higher level of abstraction. There are component models that rely on more or less formally described and standardized interfaces, and other that are built around more ad-hoc approaches.

D.2.3.1 Interfaces

An important concept related to components are interfaces. Interfaces specify a component's interactions with its environment, such as environmental constraints and expectations on surrounding components. Interfaces are externally visible representations that only expose a part of the full component implementation. In other words, an interface is an abstract model of the complete component. The aim is that the interface formalization should be sufficient to perform suitable component composition in the intended usage environment, without needing to consider components’ internal properties.

A basic assumption on interfaces is that interfaces are composable. However, as can be noted from the new definition introduced in this report, components and interfaces all come with a certain component model, only describing certain aspects of the system, and a certain composition model, which may further only define composition mechanisms for a subset of the interfaces. Hence, interfaces are never composable for all component aspects, but only those explicitly handled by the interface theory, i.e. a theory explaining the interplay between the interfaces of components to be composed. For example, many interface theories completely disregard timing, making this an emerging property of the composed system.
Examples of interface formalisms range from the less formal ones, such as IDLs [71], to the fully formal ones, including component and interface algebras [5], interface automata [2], contracts such as HRC \(^2\) [11, 13, 30], resource interfaces [24], game-theoretic approaches [4], modal interfaces [76], just to mention a few formally oriented ways to define interfaces. Different interface theories [5] are at different levels of formalization, and different interface theories also formalize different aspects of the interface. Of course, just as the choice of viewpoints in an architectural model, the choice of interface theory for a component model defines which aspects are possible to model and analyzable for a specific component model. If there is no explicit analysis support for a specific extra-functional property, there is a big risk that composability is not achieved for that property.

Interface theories are closely related to component models. Typically, a certain interface theory provides the formal foundation for a component model, typically by providing a formal definition of both the component model and its composition model.

D.2.3.2 Implications for Modularity of Components and Viewpoints

As can be seen, composability is not primarily a property of the actual implementation, but rather a property of the abstractions used for the implementation/modeling of a component/system. Any system, given sufficiently detailed modeling and disregarding execution time constraints for the analysis, will be composable for any analyzable property.

In this report we consider composability along two dimensions. Traditionally, component models try to achieve composability along component/subsystem boundaries. However, both of these concepts can also be applied to the composition of different viewpoints, i.e. for system modularization along system stakeholder concerns instead of hierarchical structure.

This makes for an interesting note: depending on the component model, the operations of viewpoint composition and component composition may not be congruent, i.e. the result will depend on in which order they are applied. The ideal case is illustrated in figure D.4. Of course, this is not a suitable trait for “well-behaved” component models. This situation is however easily avoidable, by requiring that all viewpoints should be complete in regards to composability, i.e. they do not rely on data which are not part of the viewpoint.

D.3 Multi-Viewed Components: Proposed Formalization

In this report, we represent all components and viewpoints as finite directed attributed and typed graphs. This is a common way to represent models in general, indeed suggested also by other sources [15, 54]. It also maps well to common representation formats of models, e.g. XML. It is however not the only formalization possible. It should be possible to transfer the following results also to other model representations, either through encoding the models as directed attributed and typed graphs, or by using definitions similar to the ones used below. Without further reduction of generality, we can also assume that the viewpoints can be explicitly represented as a part of such a structure.

We will introduce the following main concepts, as illustrated in figure D.5, where the main relationships between these concepts are also shown:

- Models and meta-models, \(M\) and \(\mathcal{M}\)

\(^2\)Heterogeneous Rich Components
Figure D.4: This picture illustrates the ideal case – the operations ⊕ (composition of views from different viewpoints of the same component) should be congruent with ⊗ (composition of views from the same viewpoint but different components).

- Views and viewpoints, V and V
- Components and component models, C and C
- Semantics (of a concrete view) and semantic domain (of a viewpoint), S and S

A view V is intended as an abstraction of a system σ. Since we deal with a model-based approach in this report, we can further assume the view V is represented by its model M. Formally, we look upon the view V as a mapping of its model M to a certain semantics $S = \{\sigma_1, \sigma_2, \ldots\}$, i.e. a set of systems. The set of systems can be specified either explicitly or through constraints on the system in the model M. A group of views together form a multi-view component C. Different views of the same component compose (⊕) and views of the same viewpoint of different components also compose (⊗). Meta-models, viewpoints, component models and semantic domains are each abstract meta-level equivalents to these concepts at the instance level.

A model M is a directed, attributed and typed graph according to a meta-model$^3$ M. The model M consists of typed elements $N \cup E$, constituting of nodes $N = \{N_1, N_2, \ldots\}$ and edges $E = \{E_1, E_2, \ldots\}$ where $E_i = (s_i, t_i)$. Nodes are only used as start- and endpoints for edges. An edge may also use other edges as start- or endpoints. We refer to $s_i \in s \subseteq N \cup E$ as the source element of each edge and $t_i \in t \subseteq N \cup E$ as the target element of each edge.

$^3$Of course, not all kinds of model assume a meta-model as the basis for modeling syntax, but in this report we make the assumption that other approaches, such as different types of grammars, can also be represented with a meta-model for their abstract syntax. Other possible examples include for example syntax-free grammars.
Figure D.5: Relations between the main concepts of the formalization. The gray elements refer to the concept of semantics as understood in traditional language engineering.

An edge may not refer to itself, i.e. $s_i, t_i \neq E_i$. We explicitly allow that $s_i = t_i$, i.e. edges may have the same source and target elements in the typed graph. Please note that we allow edges to connect edges and not only nodes, unlike some other formalisms. Both edges and nodes are typed $T(N)$ and $T(E)$ respectively. Each node $N_i \in N$ further has a set of attributes $a = \{a_1, a_2, \ldots\}$. An attribute $a$ is formalized as a typed name-value pair, i.e. $a = (n(a), v(a)), v(a) \in T(a)$. We use the notation $n(a)$ as a shorthand for the name of the attribute and $v(a)$ for the value of it. Each node $N$, attribute $a$, and edge $E$ has a type $T$ according to a meta-model (type graph) $M$ which discriminates valid models from invalid ones, as is known from literature formalizing meta-modeling, for example [55].

**D.3.1 Components are a Type of Models**

A component $C$ can now without significant loss of generality be represented as the composition of its views as illustrated in figure D.6; a model of the following structure:

- A top node $C_0$ of type $C$. This makes the model a rooted one.
- A view root node $V_i$ of viewpoint $V_i$ for each view. The two views in the figure have the root nodes $V_1$ and $V_2$. 
Figure D.6: Structure of a model of the component \(C\) with a top node \(C_0\) and two views, \(V_1\) and \(V_2\). The two viewpoints \(V_1\) and \(V_2\), here exemplified by a state machine formalism and a data-flow structure.

- Submodels \(M_i = \{N_i, E_i\}\) for each view \(V_i\).
- “View link” edges to make the model connected:
  - for each view \(V_i\) an edge \(\{C, V_i\}\) of type contains-view.
  - for all model elements (nodes and edges) in a view, corresponding containment links of type contains-element, i.e. the set \(E(V_i) = \{(s, t) : s = V_i, t \in E_i\}\).

A component \(C\) is hence represented by its \(n\) views \(V_1 ... V_n\), which are defined in accordance to the viewpoints \(V_1 ... V_n\) respectively, as illustrated in figure D.6. The viewpoints are not necessarily unique (\(V_i = V_j \implies i = j\)); several views may be based on the same viewpoint where each only covers a part of the full system. Each of the views describe the real-world implementation of the component in an abstract way, i.e. it is an abstraction of the full component (i.e. \(\forall i \in 1...n : V_i \models C\)) \(^4\)

\(^4\)The operator \(\models\) (usually pronounced “models”) indicates that the left operand is an abstraction of the right one. For example, given \(X \models Y\), this implies that any semantic claims about \(Y\) also holds for \(X\), i.e. in other words, \(X\) is a more abstract model of \(Y\).
Each component is governed by a component model $C$. A component model $C$ consists of a meta-model $M_i$ for each viewpoint $V_i$ and a semantic mapping $V_i \rightarrow S_i$ to its respective semantic domain $S_i$. By applying the semantic mapping to $S_i$ on a view instance $V_i$, we get the semantic meaning $S_i$ of the view. Of course, this semantic mapping will relate to the elements suggested in $M_i$, but we make the distinction between a meta-model and its semantics within a given viewpoint. By making such a distinction between the meta-model and its semantic mapping, it is possible to allow the same syntax to have more than one semantics, depending on context.

For example, in the example in section D.4, we use the possibility of using the same syntax, but giving them separate semantics, by giving a generic boxes-and-lines type of formalism two separate semantics: One where the boxes denote software components, and one where they denote hardware components. This is an important distinction compared to traditional language engineering [56], where the assumption is that the semantics is related to the languages themselves ($S_M$ for a model and $S_M$ for a meta-model respectively), as shown in gray in figure D.5. Alternatively, the mapping $V_i \rightarrow S_i$ can be seen as the union of the mappings to semantics of all individual models built according to the viewpoint, i.e. $V \rightarrow S \equiv \bigcup_{V_i} V_i \rightarrow S_i$. Both the mapping and the semantic domain $S_i$ may be implicit.

Each viewpoint may also be implicitly linked to a separately defined composition model $V_i \otimes V_i \rightarrow V_i$. If a view is intended as an interface, having a composition model for it is a design requirement. This composition model is used to produce a new composed view at the supercomponent level, i.e. $V_A \otimes V_B = V_A \otimes V_B$. The composition model may take additional information in order to perform the composition. Formally, we consider that to be a form of ternary operation on three views: two views with the information to be composed, and a third view with instructions on how to compose them, for example port mappings.

A semantic domain $S$ is a set used to give meaning to the models, i.e. it contains all possible understanding of models in a viewpoint $V$, with its associated meta-model $M$. A concrete view $V \in V$ has the meaning $S \in S$. It is quite common that the meaning $S$ is multi-valued. For example, traditional automata have multivalued semantics in the form of sets of regular languages. A semantic domain does not necessarily need to be finitely large, though it can be represented with a finite model, specifically, one or several finite views. Examples of typical semantic domains for common viewpoint formalisms would be sets of possible structural buildups, sequences of allowable output events and/or active modes (both possibly timed). Mappings to semantic domains can either be denotational or operational, and rely either on formal definitions or on informal understanding.

A relation exists between viewpoints if it exist for every group of views of the same system that are built according to the viewpoint. Formally: if $V_1 \in V_1$ and $V_2 \in V_2$ are views of the same system $s$, $V_1 \bowtie V_2 \implies V_1 \bowtie V_2$, where $\bowtie$ is the relation/relationship pattern under consideration.

### D.3.2 Semantic Relations Between Views and Viewpoints

Traditionally, viewpoints are chosen simply to reflect the interests of the system stakeholders as closely as possible [52, 53, 57]. It is the opinion of the authors of this report that it is insufficient to build integrated tool environments by choosing the applicable viewpoints for the development of a system in such a way. In addition to addressing the stakeholder concerns themselves, also the relation between the concerns, and hence the resulting concerns between viewpoints, need to be considered. In many cases, separate
stakeholders have system concerns that to a large degree overlap. With a classical approach to viewpoint choice, such overlap is not systematically detected. We therefore propose that the relations between viewpoints to be characterized, depending on the semantic relation between the viewpoints.

Two viewpoints $V_i$ and $V_j$ are semantically overlapping if their semantic domains are overlapping, i.e. if $S_i \cap S_j \neq \emptyset$. This means that they are semantically interacting with each other, and that the data shared between the views should be managed. In practice, this is a harder problem than this simple formalization implies: often the semantics is only informally or semiformally defined, and will depend on the actual usage of the languages rather than the formal language definition, and hence is hard to reason about.

It is also important to note that it is not a hard requirement to have a fully formal definition of two viewpoints to at least informally reason about their semantic relationship. For example, even if using fully informal semantics, it may be possible for viewpoint designers to deduce semantic orthogonality informally, for example by observing that they cover completely separate aspects (e.g. software vs. hardware).

We can differ between four main cases of semantic relations between viewpoints as illustrated in figure D.7:

- **Orthogonal/Independent** means that it is known $V_1, V_2$ have no direct semantic relationship, i.e. $S(V_1) \cap S(V_2) = \emptyset$. We also write $V_1 \perp V_2$. Indirect relationships, going through a third view, may be possible but no direct ones. Many approaches to multi-view modeling have an implicit but usually broken assumption that views are orthogonal if nothing else is mentioned.

- **Syntactic overlap** means that two views $V_1, V_2$ have some information that is shared syntactically, but may also some information that is unique to each view that is not sharing semantics. This means that $S(V_1 \setminus V_2) \perp S(V_1 \cap V_2) \perp S(V_2 \setminus V_1)$, and that an syntactic overlap $M_1 \cap M_2$ can be identified.

- **Semantic overlap** is used to classify all remaining cases where it is known (or possibly only assumed) that there is a semantic relation between two views $V_1, V_2$, i.e. $S(V_1) \cap S(V_2) \neq \emptyset$. There may still be an explicit way to handle the relation, e.g. through a suitable operation or by explicitly modeling the semantic mapping for both involved viewpoints.

We distinguish one special case: Two views are **semantically equivalent** if they have the same semantics, i.e. $S(V_1) = S(V_2)$.

- **Refinement/Abstraction** is a relation between views where one is a more abstract view of the system under description. Formally: A view $V_1$ is an abstraction of another
view $V_2$ if its semantics is a superset of the other, i.e. $S(V_1) \supseteq S(V_2)$. We may also say that $V_2$ is a refinement of $V_1$ or write $V_1 \models V_2$. For a detailed discussion of different variants of abstraction see [84]. Refinement may both be homogeneous (using the same formalism, providing a more detailed description) or heterogeneous.

We distinguish one case especially: filter/subset, as a relation where the more abstract view $V_1$ is a strict syntactical subset of the refined view $V_2$, i.e. $M_1 \supseteq M_2$ and $S(M_1) \perp S(M_2 \setminus M_1)$. An example may be to filter a specific concern, e.g. to only mechanics from a model covering both mechanic and electric properties. In the opposite direction we speak about extension, where a view is extended by additional information.

By encoding these relations between viewpoints, it is possible to formally describe the semantic lattice that the viewpoints are spanning together, as a basis for analysis. In the case of semantic overlap only, it can be characterized to different extents. It may only be known that there is overlap, an explicit analytic procedure to check the consistency for views may be known, or a semantically complete mapping may be known, e.g. a model transformation.

An interesting question is how to handle viewpoints that have been marked as semantically overlapping without known mapping. This implies that they are potentially conflicting with each other, or at least they have not been confirmed to be conflict-free. A safe assumption would of course be to assume that viewpoints who haven’t had their relation investigated are normally conflicting. Depending on development approach, such combinations could be completely disallowed, be allowed but imply a mandatory manual consistency check, or it may be decided to only ensure consistency between these views as far as there is formalization of the shared semantics (i.e. the most detailed shared node in the semantic lattice that is available for usage as a pivot meta-model).

By combining these semantic relations of viewpoints with the idea to split viewpoints in smaller subviewpoints. This viewpoint decomposition makes it possible to build more fine-grained descriptions of the semantic relations of more complex cases than the above mentioned ones.

Do note that we nowhere speak about equality of models, at least in the case that they are not having identical semantic domains, we refer to them having the same semantics instead. There are potentially more than one way to model the same thing. We call such views semantically equivalent. In practice, it is normally hard to decide if two views are semantically equivalent, since the semantic mapping is in today’s practice very seldomly explicitly modeled. Even if the syntactic level is entirely formal, if the semantic domains are not matching, model equality is not a very interesting concept either. In the extreme, semantic equality may depend on the modeling detail level, e.g. if the full semantics is explicitly given or not. Equality is not an absolute concept in this context [66]. In practice, equality will depend on how well the relations between the different viewpoints are known and to what level they are needed to be consistent. This level will depend on the level of formality of the semantics of the models, the latter which of course depends a lot on the development process employed.

A first example on this is from algebra – the expressions $(a + b)(a - b)$ vs $a^2 - b^2$ are in normal algebra equivalent: depending on the level of semantics formalization, this may or may not be apparent. It may also in some cases be relevant which is chosen, e.g. for numerical stability issues.

A second example comes from the domain of model-based development, where diagrams are often used to represent more formal information. Many such modeling
languages (e.g. UML) put very little emphasis on the actual looks and layout of a diagram since they don't convey any formal meaning, but depending on the development process, companies may want to enforce layout consistency between different models of the same component (i.e. in order to make engineers have a sense of familiarity across different views).

A third example comes from the area of model integration: [87] presents a mapping between Simulink and UML. Despite being semantically incomplete, such a partial mapping is still very useful in practical work, in order to ensure that at least structure is consistently defined.

D.3.3 Composition of Viewpoints

We further allow there to be a defined composition function between combinations of different views of the same component, $V_i \oplus V_j \rightarrow V_i \oplus_j, i \neq j$. The nature of this composition function may differ. In the simplest case, completely orthogonal viewpoints, it is simply the disjunct union of the two viewpoints, i.e. $\oplus V_i \rightarrow V_j \oplus_j = V_i \cup V_j$. In the case where no such composition function is defined between two viewpoints, then the two are not composable with each other and can not be handled in an automated manner.

Direct heterogeneous composition, i.e. composition of different viewpoints from different components ($V_1(A) \otimes V_2(B)$), is by principle discouraged in the suggested approach. The reason is that it leads to unclean composition models, e.g. direct composition between hardware and software components from hardware- and software-specific views respectively. However, the framework allows indirect heterogeneous composition, in that each relevant viewpoint is transformed to the composed viewpoint before the composition between the components:

$$V_1 \oplus_2 (V_1(A) \otimes V_2(B)) = (V_1(A) \oplus V_2(A)) \otimes (V_1(B) \oplus V_2(B))$$

In the case of semantically non-overlapping viewpoints this simplifies to:

$$(V_1(A) \oplus V_2(\emptyset)) \otimes (V_1(\emptyset) \oplus V_2(B))$$

For viewpoints that are semantically overlapping, i.e. where the semantic domains are overlapping ($S_i \cap S_j \neq \emptyset, i \neq j$), other more intricate ways of defining the composition are possible, such as unions (if both syntactically and semantically consistent), synchronous composition (for automata) and so on. If the set of viewpoints is complete (i.e. they completely describe the component), then $C = \bigoplus_{i=1}^{n} V_i$. This is not the common case; during the development of a component the description is normally incomplete. Further it is common that subcomponents are not fully represented, either for reasons of intellectual property rights, or simply due to abstraction, scalability or lack of documentation.

D.3.4 Semantic Lattice of Viewpoints

By letting all views refer to each other and explicitly model their relations in terms of expressibility/abstraction of each other, a semantic lattice is formed. Such a lattice would

---

5This of course assumes that sufficiently many such operations have been defined, such that composition over all views is properly defined.
have a bottom element $C_\perp$ which consists only of the hierarchical composition tree without any internal details, and a top element $C_\top$ which includes all available viewpoints (whether or not composition functions are explicitly defined). Common branches in between would contain meta-models with expressivity in between, with all data included in them potentially shared by multiple meta-models at leaf level. The semantic lattice will hence build on the principle of open but formal semantics at the root and lower levels. The more common information is had, the closer to each other the formalisms would be placed in the lattice. In order to reach such a hierarchical build-up in formalisms, without conflict with the requirement on well-founded (formal) semantics, it is important to choose the formalism of the root of the viewpoint lattice suitably.

The principle of open but formal semantics implies that although the component relies on formally defined semantics, the formalization is quite weak for the root model and is only introduced gradually as further extensions are added. An illustration is given in the later figure figure D.12. This means that the core formalism needs to be really barebone, providing in the very most flexible case only minimal semantics (e.g. component names and basic composition structure), but no further semantics in itself, rather, leaving those aspects for extensions that further detail the description of the component. Depending on the tool environment (i.e. if the tools use similar formalisms), of course, the more of the extensions building up the component model need to be shared.

A key idea is to explicitly deconstruct viewpoints of each component/system into appropriate subviews instead of the common current practice of viewing them as monoliths. It is clear that many viewpoints rely on a common core of information, however, the content and extent of this common core will depend on which viewpoints are to be made consistent with each other. By deconstructing viewpoints into subviews, it will be easier to describe the relations between different viewpoints, as different parts of the viewpoint will be smaller in size, and potentially have simpler relationships to other viewpoints. The viewpoints will also form a less complicated semantic lattice of formalisms. By the classical model-based development motto: “all models are approximations, but to different extents”. The more specialized viewpoints will be closer approximations for a specific concern.

With composable viewpoints and the principle of open but formal semantics, it is made more easy to choose a pivot viewpoint/meta-model that is as inclusive as possible for information contained in both the source and target viewpoint(s).

Pivot meta-models are used for the translation between other meta-models rather than in their own right. This means that they are normally used to couple together several (at least two but usually more) meta-models over their common semantic foundation. Pivot meta-models may also be used in a transformation (one example is ), when this is not complete with regards to the shared semantic basis but information is lost. The design of pivot meta-models has so far been rather ad-hoc. Given information on the semantic loss in the translation to the pivot meta-model, it becomes feasible to explicitly analyze any data loss (due either to being out of scope for the target view, or due to lacking transformation availability). This however requires that viewpoints are built in a manner which make them composable, i.e. with predictable side effects of composition.

### D.4 Example

In order to illustrate the concepts described in the previous section, we give a simple example in form of a simple rain-sensor-equipped windshield wiper system. The example
is based on a set of four main semantic domains/perspectives: structural breakdown (hardware and software), mode switching and allocation of software to hardware, and show how they combine to a multi-viewed component model. For each semantic domain we define a few different relevant viewpoints at different level of detail. The example has been illustrated with a very simple system design, based on a windshield wiper system with a rain sensor, informally illustrated in figure D.8.

The example is purposefully simplified compared to real-world examples, in order to increase its understandability, and shall not be considered representative of real development work or as a suggestion for viewpoints to be actually used for actual development, except possibly at a very high abstraction level only. The example should only be sufficiently complex in order to showcase the principles behind the formalization of relations between viewpoints.

### D.4.1 Structural Viewpoints

The first viewpoint of the example is the physical and logical structure of the system, i.e. how hardware and software is connected. This is usually one of the key parts of a system, and a common choice for a “main” viewpoint that most other viewpoints partially or completely will depend on. Many of today’s simpler component models (e.g. CCM [72]) have the structural view as the single exposed interface view, or at least the only one having a formally defined semantics. The structure of a system is easily decomposable into several separate subparts. Given a common structure, where each component has ports, and ports are connected by connectors, the components expose some ports to be visible outside the supersystem too. Both the hardware and software structure can, at an abstract level, be represented with the following viewpoints, all with informal semantics:

- **$M_t$ – Hierarchical breakdown tree.** The component is built of subcomponents $C = \{C_1, C_2, \ldots, C_n\}$. All components $C_i$ except the root one ($C_0$) have a relation “part” to a different component, i.e. there are edges $E_{\text{part}} : \{(C_1, C_0), \ldots, (C_i, C_j) : 1 \leq i, j \leq n\}$

---

6Informal semantics can be formally represented by using the equivalence relation as a semantic mapping, i.e. $\mathcal{S}_i = M_i$ and the mapping $M_i \rightarrow \mathcal{S}_i$ is the identity function.
Figure D.9: The complete meta-model $\mathcal{M}_p$, excluding the constraint that the components form a tree. Black elements are introduced as part of $\mathcal{M}_t$, blue as part of $\mathcal{M}_ep$ and red with $\mathcal{M}_b$. The HW and SW specializations of components are also indicated in gray.

and the graph formed by $E_{\text{part}}$ is a tree, i.e. it contains no loops, and $C_0$ is reachable from all other components $C_i$. The relations are of type $\text{part}$. We finally denote all reorderings of components $C_1, C_2, \ldots$ to be equivalent as long as the above constraints are met.

Composition: Composition of a component $B$ into $A$ implies making $B$ into a new subtree of $A$, i.e. $A \otimes B \equiv \{N_A \cup N_B, E_A \cup E_B \cup \{(B, A), \text{par t}\}\}$. Of course, it is feasible to introduce a new, empty, component if two components are needed to be composed into a completely new structure, i.e. the composition is $C_0 \otimes A \otimes B$, using the new empty component $C_0 = \{(N_0), \emptyset\}$ as a new root node.

- $\mathcal{M}_{ep}$ – Structure with exposed event ports. Each component, additionally to the content of the previous viewpoint, has a list $p = \{p_1, p_2, \ldots\} \in \mathcal{N}$ of exposed ports. In addition to a unique name $n(p_i)$, each port has a direction attribute $d(p)$ (either input or output). All ports which are not event ports are kept internal and not exposed in this viewpoint. There may only be one internal port of of each name mapped to the outside.

Composition: Composition is defined as for hierarchical breakdown trees, additionally the port lists are merged as follows: $p_{A \otimes B} = (p_A \cup p_B) \setminus \{p_a \in p_A : \exists p_b \in p_B, n(p_a) = n(p_b), d(p_a) \neq d(p_b)\} \cup \{p_b \in p_B : \exists p_a \in p_A, n(p_b) = n(p_a)\}$. This means that ports which share names and have differing direction are connected to each other, and ports that are not given any such connection are exposed to the outer world (in a single copy if the same port name is used in both components). An additional constraint is added which makes the composition invalid if there is more than one output port with the same name: $\exists p_a \in p_A, p_b \in p_B : n(p_a) = n(p_b)$ and $d(p_a) = d(p_b) = \text{output}$, i.e. it is not allowed to connect two output ports to each other.

- $\mathcal{M}_p$ – Complete port structure. The structure is extended also with ports which are not of the event type, i.e. a larger set $p' = \{p_1', p_2', \ldots\}$ extends the port set from the previous viewpoint (i.e. $p' \supseteq p$).

Each component port is further assigned a type system which we chose arbitrarily for the example: $T(p') = \{\text{int, real}\}$, a temporality $\tau(p') \in \{\text{continuous, discrete, event}\}$ and a direction $d(p') \in \{\text{input, output}\}$. Event ports exposed in the previous
viewpoint must all be event or discrete and must match direction and temporality, i.e. \( p' \in p \) and \( n(p) = n(p') \implies \tau(p') \in \{\text{event, discrete}\} \).

Composition: Just as for the previous viewpoint, ports are exposed by a port merge operation \( p'_{A \odot B} \equiv p'_A \cup p'_B \setminus \{p'_a \in p'_A, p'_b \in p'_B : n(p'_a) = n(p'_b), d(p'_a) \neq d(p'_b)\} \). These mappings, just as the ports, need to be consistent (i.e. \( p_{A \odot B} \subseteq p'_{A \odot B} \)). We finally also here require that the type, temporality and direction are equivalent and not inconsistent between the two connected components (i.e. \( n(p'_a) = n(p'_b) \implies d(p'_a) = d(p'_b) \)).

\[ D.4.1 \text{ Hardware and Software Viewpoints} \]

In the above, meta-models for structural description have been proposed. However, their semantics does not clearly tell if it refers to hardware or software. We hence specialize the structural views, we consider two viewpoints that share the already introduced syntax, i.e. they use identical copies of the same syntax, not the exactly the same one. They however have one important difference in semantics: one considers the hardware structure and the other the software structure.

For the hardware aspects, the structure with event ports makes little sense so we only use the first and last viewpoint for hardware, so we have two viewpoints \( V_{VHW} \) and \( V_{VHP} \), based on the modeling languages \( M_t \) and \( M_p \) respectively. As main nodes, we consider in the hardware view ECUs, each corresponding to a single computer system. Each connecting network is also viewed as a type of node, and we add a connection for each connection of node to network.

In the software view, nodes are software components, connected by software component connectors as specified by the implementation of the component framework (i.e. direct connections such as function calls, event input and outputs, input/output buffers and so on). In the example we have three relevant software viewpoints: \( V_{VSWt}, V_{VSWEP} \) and \( V_{VSWp} \), based on the modeling languages \( M_t, M_{ep} \) and \( M_p \) respectively.

\[ D.4.2 \text{ Allocation Viewpoint: Software to Hardware} \]

We further introduce an allocation viewpoint \( V_{alloc} \), illustrated in figure D.10, which maps software to the hardware it is allocated on, i.e. a mapping from software nodes to the hardware nodes they are allocated to. Formally, this means that there is an implicit consistency checking viewpoint \( V_{HW@SW@alloc-check} \), based on \( V_{HW} \cup V_{SW} \cup V_{alloc} \). The viewpoint only introduces a set of additional edges, i.e. \( M_{alloc} = \emptyset, E_{alloc} \), where \( E_{alloc} = \{(N_{SW1}, N_{HW1}), (N_{SW2}, N_{HW2}), \ldots \} \).

We further need to introduce consistency constraints in this implicit viewpoint, that can not be ensured in \( V_{alloc} \) alone: with the level of abstraction used in this example, we do not constrain or even analyze the allocation of remote communication between software
components over the network (e.g., in terms of communication load on the network). We do impose a minor constraint on the allocation though: There has to be some physical connection between the involved hardware nodes, i.e. software which is connected may not be allocated to two separate “hardware islands”. This constitutes an implicit viewpoint \( V_{\text{alloc-check}} \) which is never explicitly given but only encapsulated in the checking of the consistency constraint \( \forall E \in E_{\text{alloc}} : N_{SWi} \text{ is reachable from } N_{SWj} \Rightarrow N_{HWi} \text{ is reachable from } N_{HWj} \). Further we require all software components to have an allocation, i.e. \( \forall N_{SW} \in E_{\text{alloc}} \).

Of course, with a more detailed model of the system, additionally detailed analysis would be feasible, for example to find out if any bus would risk becoming overloaded or not.

Composition for this viewpoint is done by simple merging of the allocation lists, i.e. \( E_{\text{alloc}}(A \oplus B) = E_{\text{alloc}}(A) \cup E_{\text{alloc}}(B) \). Of course it is assumed that \( A \cap B = \emptyset \) for this composition to be valid.

### D.4.3 Mode descriptions

The last semantic domain we use are applied mode descriptions, illustrated in figure D.11. Modes are high-level descriptions of the behavior of systems, similar to states but usually on a higher level. We map all different types of mode descriptions to the same semantic domain \( S_m \), consisting of sequences of active mode sets. The set of all applicable modes for the system is denoted as \( m \), and the semantic domain consists of ordered sequences \( \langle m_0, m_1, \ldots \rangle \), where \( m_0, m_1, \ldots \subseteq m \).

We describe four different variants of mode descriptions: mode lists \( (V_{ml}) \), mode groups \( (V_{mg}) \), hierarchical mode groups \( (V_{hmg}) \) and hierarchical mode machines \( (V_{mm}) \). The semantics are then given by the respective semantic mapping \( V_{ml} \rightarrow S_m, V_{mg} \rightarrow S_{mg}, V_{hmg} \rightarrow S_{hmg}, \text{ or } V_{mm} \rightarrow S_{mm} \), where \( S_m \supseteq S_{mg} \supseteq S_{hmg} \supseteq S_{mm} \). Through this, we define in total 4 different formalisms, as illustrated in part of figure D.9:

- \( V_{ml} \) – Mode lists. Consists of a list of modes, i.e. the set \( m \). Any combination of modes may be active at any time, forming a sequence of lists of currently active modes.

  Composition: The composition of components \( A \) and \( B \) consists of the union of the mode lists: \( m_{A \oplus B} \equiv m_A \cup m_B \).  

  Semantics: The semantic mapping is simple, as any sequence of sets of active modes is allowed: \( V_{ml} \rightarrow S_m \).  

---

**Figure D.11**: The complete meta-model \( M_{mm} \). Parts introduced in \( M_{ml} \) are given in black, those introduced in \( M_{mg} \) in blue, those from \( M_{hmg} \) in red and those from \( M_{mm} \) in green. Constraints are not shown: In \( M_{hmg} \), loops are not allowed, and in \( M_{mm} \) additional constraints on which transitions are allowed to be used apply.
• \( \mathcal{V}_{mg} \) – Mode groups. Modes are assigned one mode group each, i.e. additionally we introduce the groups \( g = \{g_1, g_2, \ldots \} \), and mode group links \( E_{m \rightarrow g} = \{m_i, g_j\}, \ldots \) of type part. Semantics now differ: within each mode group, at most one of the modes may be active at each time (and potentially none).

Composition: As in the previous case, mode groups are also composed under set union, i.e. \( g_A \otimes g_B \equiv g_A \cup g_B \). Mode group links are also merged in the same manner, i.e. \( E_{m_A \rightarrow g_A} \otimes E_{m_B \rightarrow g_B} \equiv E_{m_A \rightarrow g_A} \cup E_{m_B \rightarrow g_B} \), removing any duplicate link copies (i.e. where \( g(m_A) = g(m_B) \), only one of them is preserved).

Semantics: The semantic mapping is now more complex: \( \mathcal{V}_{mg} \rightarrow S_{mg} \), where \( S_{mg} = \{mathbf{m} : \forall m_i, g_j : |\{m_i : g(m_i) = g_j\}| \leq 1\} \), i.e. there may at any point in time be at most one (and possibly none at all) active mode in each mode group.

• \( \mathcal{V}_{hmg} \) – Hierarchical mode groups. Modes may additionally have a master mode, i.e. an additional constraint saying that when master mode is inactive, all of the submodes also have to be inactive. There may or may not be a requirement that one submode is active whenever the supermode is active.

Composition: As in the previous case, mode groups are also composed under set union, i.e. \( g_A \otimes g_B \equiv g_A \cup g_B \). Mode group links are also merged in the same manner, i.e. \( E_{m_A \rightarrow g_A} \otimes E_{m_B \rightarrow g_B} \equiv E_{m_A \rightarrow g_A} \cup E_{m_B \rightarrow g_B} \), removing any duplicate link copies (i.e. where \( g(m_A) = g(m_B) \), only one of them is preserved).

Semantics: The semantic mapping is now more complex: \( \mathcal{V}_{mg} \rightarrow S_{mg} \), where \( S_{mg} = \{mathbf{m} : \forall m_i, g_j : |\{m_i : g(m_i) = g_j\}| \leq 1\} \), i.e. there may at any point in time be at most one (and possibly none at all) active mode in each mode group.

Composition: As previously, with the additional set union of the relevant mode groupings. The constraint of no loops applies.

Semantics: Just as previously for \( \mathcal{V}_{mg} \), but with a further constraint: \( \exists m(g(m)) \) and \( m_b \in \mathbf{m} \), \( \implies m(g(m)) \in \mathbf{m} \), i.e. if a submode \( m_b \) is active in \( \mathbf{m} \), so is its supermode \( m(g(m)) \).

• \( \mathcal{V}_{mm} \) – Hierarchical mode machines. Mode groups are extended with internal transitions between the different modes. Within each mode machine, one mode is designated as starting mode. The sequence of modes at execution time must now follow the sequence of switching as described by the transitions and starting modes (there are different ways to handle what happens for a submode machine when the supermode is inactivated – to retain status, or to reinitialize). Transitions are annotated with either input or output events, which need to be synchronized with the exposed event ports as described in one of the viewpoints for the physical structure, in our case through their naming. \( \mathcal{V}_{mm} \) is dependent on the software structure, specifically \( \mathcal{V}_{SW_{ep}} \), since the events from event ports are reused to direct the mode machines. In this case, we further add to the semantics transitions \( \tau = \{\tau_1, \tau_2, \ldots\} = \{(\tau_{s1}, \tau_{i1}), (\tau_{s2}, \tau_{i2}), \ldots\} \) between modes (for now, we choose not to introduce transition guards or any other very commonly applied mode machine extensions).

Composition: Composition is given by synchronous composition as normally defined for automata. We choose not to give that definition here.

---

The exact choice of formalization lattice is important in this case. A slight variation of the mode group concept, requiring exactly one mode to always be active in each mode group, would have implied the need for introducing additional “off” modes for the hierarchical mode subgroups, in order to handle the case of the supermode not being active, and handling the constraint setting this mode as active when the supermode is inactive. Although workable, such an approach would be more complicated.
Figure D.12: The viewpoints introduced in the example with their direct refinement and composition relations to each other, forming the semantic lattice of the example. There are four main technical perspectives – hardware, allocation, software and modes. Dashed arrows denote implicit relations.

Semantics: Semantics are as above, with the additional constraint that transitions need to be obeyed, i.e. \( \forall m_i : m_i = m_{i+1} \) or \( \exists \tau_i = (\tau_s, \tau_t) \in \tau \) and \( m_{i+1} = (m_i \setminus m(\tau_s)) \cup m(\tau_t) \).

D.4.4 Combining the viewpoints

The four groups of viewpoints have a minimal but existing overlap, in that the only shared constraint between them is the event names that are both part of the ports and the basic structure of the system and there only the name is affected, per the usual assumption that identical names imply direct mapping. This leads to an implicit viewpoint \( V_{SWep@mm-check} \) that can be used to handle inconsistent definition of events, e.g. that events that are not available on the ports are used or that some events from the ports are not handled.

Further, the mode composition structure needs to be mappable to a subset of the
system structure (some subcomponents in the structure may not have more than a single mode), such that there are no mode descriptions built for inexistent components. Different viewpoints also interact due to the allocation viewpoint referring to both the software and hardware viewpoint, introducing the additional design constraint that software connectivity implies hardware connectivity as well.

Further constraints could be added, e.g. that mode names should be consistent with the system structure. We have however not introduced them into the example, in order to improve understandability.

D.4.5 Summary

The example gives a group of related and interconnected viewpoints. This is still a relatively simple set of viewpoints. All of them are either unrelated or can be mapped to each other as syntactical refinements. Hence, it is also very much simplified: real-world sets of views will have more complicated relationships between the viewpoints, including those that are entirely on the semantic level.

D.5 Discussion

The introduced formalization of multi-view components given in this document can potentially be used by framework developers in several ways as discussed below.

D.5.1 Implementation of a Multi-View Component Model

The first way of applying the insights and formalization described in this report would be to apply it as the formal basis for an implementation of a generic heterogeneous multi-view component model. Building such an implementation would imply not only choosing a certain modeling environment, but also a development environment and other tools such as a suitable tool integration framework and languages for the instantiations of the component models that are instantiated.

It also implies choosing suitable meta-languages and semantic mapping languages. Languages used for semantic mapping may either rely on formal definitions or on mappings to formal definitions, i.e. relying either on a denotational or an operational description of semantics. These mappings would then be used in order to ensure

In contrast to traditional model-based approaches for multi-view modeling, it would also be necessary to explicitly describe the result of component composition explicitly, just as in component-based design. It would further also be explicitly necessary to talk about the semantic composition of separate but semantically overlapping viewpoints. By doing so explicitly, it becomes possible to before-hand understand the possible semantic collisions and challenges, and handle them.

D.5.2 Usage as Design Principles for Viewpoint Sets

Since views are modeling the same architecture from different viewpoints, it is vital that relevant parts of the descriptions are consistent between views. The degree of formalization of this synchronization will depend on the development process. The problem of multi-view consistency is a substantial challenge in itself, and there are multiple different
approaches to ensure consistency, and there are also many different approaches to the concept of “component”.

By applying the analysis introduced in section D.3, it is possible to evaluate information overlap in existing view frameworks, to see the extent of semantic overlap between different viewpoints. As such, it can also be used for evaluation during the design of new view frameworks, as a design guideline or design requirement. Looking back to the example (section D.4), it has been built in a generally modularized way emphasizing clear viewpoint decomposition.

Hierarchical composition of viewpoints will only, in general, be composable if and only if a few key principles on viewpoint choice are used:

- **Hierarchical refinement through the semantic lattice.** The more specialized a view is, the more specific the model of the system/component should be. This means that removing a view from a view hierarchy (and possibly all its dependent views) would result in a more general model. Views that abstract and refine each other hence create a semantic lattice (see subsection D.3.4 for further details).

- **Explicit view referencing.** When a viewpoint is dependent (i.e. a refinement) on data in another view, this is explicitly modeled, either as semantic subsets, through transformations, or as having a targeted consistency check (which may be applicable only for certain views of the viewpoint).

- **Proper viewpoint decomposition.** In order not to inflict unnecessary conflicts between viewpoints, it is necessary to decompose them in a suitable manner. For example, almost any viewpoint will relate to the physical structure of the system to some extent if one is used. Hence, such data should not be reproduced in every single viewpoint but coordinated in a special physical structure viewpoint which the other viewpoints extend. It is possible to do a comparison to table normalization, as used in relational databases [38], which is similar in its purpose: To avoid redundant duplication of data, especially if it is only implicit.

- **Focus on semantic consistency, but only up to a chosen level.** The main principle is that to ensure consistency, syntactic consistency is not sufficient. To ensure full semantic consistency is however not a scalable or even feasible solution, since different approaches and users may put different levels of semantic meaning into the same diagram depending on context. Relying on the idea of different levels of semantic consistency, we make it feasible to make semantic consistency well-defined but partial.

- **Explicit handling of conflicts.** Despite all efforts to coordinate viewpoints as best as possible, there will always be a few remaining conflicts between them, typically based on fundamentally different modeling concepts that are not easily mappable to each other. Such a conflict can usually only be handled through consistency checking – either manually or through mapping of both views to a third formalism. Yet, just knowing about the data representation conflict may be a sufficient input in order to pick a more suitable (more homogeneous) set of viewpoints.

These principles may in practice be applied for viewpoint design in several different manners: (1) as a guideline/recommendation when constructing new viewpoints, (2) as an evaluation means when evaluating view frameworks including the viewpoints that they specify, and (3) as a design requirement which is an absolute requirement for viewpoint choices.
In order to fully make such mappings as mentioned above feasible, formalization of the semantics of the viewpoints is needed. In particular, only through full formalization of the relationships between the different formalisms, it will be possible to fully ensure viewpoint orthogonality without a very effort-needing manual process reliant on understanding of the actual semantics, as used in “the wild”.

If sufficient formalization of two separate viewpoints is available, especially in the case of a complete mapping to a common third formalism, and the relation between the concepts in the different viewpoints are not too dissimilar, it may even mean that it may be possible to automatically generate a semantically consistent transformation between them. This may also be the case if the viewpoints are modularized and a mapping exists for the common subset, and it can be verified that the remaining parts are non-overlapping.

If the the view decomposition concept is applied to components and suitable composition models are available, both views and components will compose in a hierarchical manner. Each subcomponent will consist of a number of views, as will each supercomponent. This set of views may or may not be identical or relevant subsets to each other.

D.5.3 For Analysis of Existing Viewpoint Sets

Just as the principles can be used when designing entirely new sets of viewpoints, they can also be applied on already existing viewpoints or development environment. The degree to which these principles are adhered give an indication of the complexity of using them in actual development, whether view consistency is handled manually or has automated tool support.

D.5.4 Analysis of Semantics through Viewpoint Decomposition

If viewpoints are explicitly and cleanly decomposed according to the guidelines presented in the previous section, and modeling formalisms are chosen in a way that makes them less

For example, a transformation chain could apply transformation from meta-model $M_A$ to meta-model $M_C$ over meta-model $M_B$ as follows, using the largest possible semantic pivots for the steps in between each step:

$$
M_A \xrightarrow{M_{A\cap B}} M_B \xrightarrow{M_{B\cap C}} M_C
$$

which simplifies to (if the middle stage is neglected)

$$
M_A \xrightarrow{M_{(A\cup C)\setminus B^C}} M_C
$$

where $B^C$ is the complement of $B$. This way it is easy to analyze the implication of doing the transformation in two steps: all information which is not representable in $B$ is lost. This could be alleviated through consistency checking or other means. In this case there is probable information loss (which may or may not be relevant), unless $(S_A \cup S_C) \setminus S_B = \emptyset$.

Of course, with specific models, it is potentially so that the actual information loss is less. That can be the case if for example only a certain part of the meta-model is actually used, and the lossy part of the meta-model combinations are not actually used by the modelers.

By coupling well-structured decomposable viewpoints with tools that describe the toolchain (e.g. TIL [18], FTG+PM [64], or the process used for development, e.g. SPEM, BPMN,
BPEL [10], we can thus semantically analyze the semantic loss of meaning in a tool chain, even though the above discussion only sketches the approach.

The captured semantic relations can also be used as the first step towards change impact analysis between views. A change in one view is only possibly an implication for changes in other views if these other views are semantically dependent on each other, such that the changed information would have impact over viewpoint borders.

D.6 Related Work

Multi-view integration is not nearly a new subject, not even in the context of component-based design. We here refer to two slightly different groups of reports: firstly overview articles surveying several relevant approaches within a certain approach or with a certain angle or groups of articles with similar approaches, and secondly articles describing individual, especially interesting, approaches more in detail.

The individual approaches relate to specific approaches tackling the same problem as this report, or at least a bigger subset of it. The approaches have been roughly categorized into subsections based on their approach to the problem. Common to all of the related work listed here, is that they do not take all the aspects included in this report into account, as outlined by subsection D.1.1. Specifically, at least one of the following key points of this paper is being excluded in each of them:

- not providing composability in composition (between components and/or viewpoints) or even has no definition of the composition operation at all
- not allowing heterogeneous formalisms (or even custom ones)
- only syntactic consistency reasoning or none at all – disregarding the semantic relations between different viewpoints, needed especially in the case of heterogeneous formalisms
- too theoretic for actual implementation and understandability
- not providing a clear definition of viewpoints, or only a very informal one
- the approach sets a predescribed set of viewpoints and does not allow definition of custom ones.

D.6.1 Overview Articles

Model integration has been surveyed by among others [25], where a number of different model integration mechanisms are compared, some of them having component concepts, however defined in a simple operational manner, not ensuring semantic consistency.

Boronat et al [20] distuingish between three main approaches to multi-modeling languages, i.e. using views and viewpoints in the software engineering community: a “system model approach” where all individual viewpoints are translated into a common system model, a “model-driven architecture approach” where model transformations handle consistency on a (typically) syntactic level, and a “heterogeneous semantics and development approach” where each viewpoint is given a mathematical semantics, and the consistency is analyzed. They further formally define the concept of a generic multi-modeling language, i.e. a group of coupled modeling languages that have a partially shared semantics. The semantics is given through mappings to the category-theoretic concept of
institutions and through semantic connections between different modeling languages. The paper however does not consider component-based concepts such as composability (neither of components nor viewpoints).

D.6.2 Category Theory

Category theory is a mathematical approach with the main emphasis on modeling relations between different types of object groups (called categories), in order to describe the structure of relations between the objects rather than the objects themselves. The area is highly abstract and is even being called “general abstract nonsense” by many. As reasoned by [34], there is a clear link between the mathematical area of category theory and model-based engineering. It is further illustrated by a report [33] on generic formal semantics of heterogeneous multi-models, although not explicitly considering componentization and composition.

Still, some papers in the area have approached the problem of modularizing components into viewpoints while still providing even semantic composability, e.g. [35]. Unfortunately, most of these papers keep the reasoning at the very abstract level. This unfortunately make these papers very inaccessible to implementers of modeling environments or component models. The best exception to that rule is [36] which explicitly includes also models and meta-models for a concrete example. Unfortunately, that approach takes an entirely syntactic approach to consistency, as do many of the other category-theoretical papers: [26, 44, 54] which deals of conflicts between model transformations. This worldview is insufficient in the heterogeneous development environments that are relevant for this report.

The foundations of the Rosetta language and semantics [88] are based on a category-theoretic relation between different discrete formalisms in a domain lattice (similar to the semantic lattice suggested in this report). To accomplish this, facets are used which can extend each other. However, it does not explicitly take the same understanding of viewpoint composition as this report, in that it assumes separately defined parts of the facet specifications to be orthogonal.

In summary, most category-theoretical papers on multi-viewed components provide much interesting theory but little practical use for implementers - they are written at an abstract level and hence hard to understand.

D.6.3 Model-Based Design and Architecture Design

The concepts of view and viewpoint has been standardized in the IEEE 1471 [52] and ISO 42010 [53] standards. These standards also discuss the concept of view frameworks: i.e. a set of predefined viewpoints to be applied to systems under design. By using a predefined set of viewpoints, the relations between viewpoints can be well understood, but is usually not formalized. Examples of view frameworks include 4+1, MoDAF, DODAF and more.

Already early in the field of model-based design [40] was the need for view integration and composition identified. Most papers in the area however assume identical abstract syntax (or at least structure of the abstract syntax) in order to compose views. Even fewer take composition of components into account explicitly.

Early efforts include e.g. [41] which implements view consistency checking through heuristic matching of the graphs in the different views. Later work, e.g. by Bhave [14, 15]
have extended that principle through the modeling environment AcmeStudio. In this tool, views are seen as partial projections of a base architecture. All views use a component-connector structure, and graph matching is used in order to find the correspondences between different views. Another principle is used in [77, 78], where the meta-models of the different viewpoints are complemented by explicit links are used to model relations between model elements that are related. Through a mapping to a traditional database, traditional database normalization approaches can be used on the data. Another variant is to explicitly define the consistency checks needed, as in [59].

The SPES project [74], similarly to CESAR [9, 79] and iFest [99], defines a development project for based on a multi-viewpoint formalism. However, although both of these are aligned with methodology, they are less concerned about core values of component-based development, and do not provide e.g. explicit declarations of composition. They also presume that the viewpoints to be used, at least the core ones, are industry-wide predefined ones.

There are also older papers in the model-based discipline using similar approaches. [49] provides an approach where structures (i.e. groups of model elements) in different models are given explicitly modeled relations.

## D.6.4 Component-Based Development

In work on component-based development and different component models, there are several interesting example. For example, Alfaro and Henzinger [3] introduce a number of component models relating to each other, with similar related interface theories. They however do not explicitly model/characterize these relations between the different component models, nor do they explicitly consider these different formalisms as specific views of the same system, usable in different contexts. Henzinger build upon that work by discussing the relations between interfaces in separate viewpoints [51].

HRC is an component-based approach with contracts [12, 30] over hybrid automata as interface formalization. The approach has four predefined viewpoints, and additional ones can be added. However, the different viewpoints are not orthogonal to each other; they all share the same meta-model and semantic domain and the viewpoints are rather an informal categorization. Since the composition is based on synchronous composition (an operation which in general can not be shown to be composable since the different viewpoints may interact with each other in unexpected ways).

De Lara et al [47, 48, 60] describe a multi-viewed component model where the consistency between views is ensured by gluing them through relating them to each other with triple graph grammars (TGGs), a type of model transformations. Thereafter the glued model may be stored in a common repository which can ensure consistency. A concrete example is based on structure and behavior (timing diagrams). The approach does not explicitly handle composition.

Co-simulation is a special case of multi-view modeling, where the semantics of the views are given as the joint simulation behavior as they execute. [95] defines a suggestion for a modular formal semantics for the heterogeneous modeling tool Ptolemy, including description of the models of computation SR (synchronous), DE (discrete event), CT (continuous), PN (Petri net) and MM (state machines). Formally, such an approach implies building a group of modeling languages with mappings to the same, joint semantic domain, typically for simulators different variants of timed sequences or traces of output valuations
The problem of mapping different simulation semantics to each other is further discussed in [82], where also relations between different kinds of semantics of some models of computation are described. An explicit definition made is the distinction between abstract and concrete semantics. The paper further relates these concepts to implementations in the Ptolemy [95] and Metro II [31] tools, which both combine certain predefined groups of formalisms.

A further group of papers introduce specific multi-viewed component models of different kinds. Most of these work with a predefined set of viewpoints, as for example exemplified by [45], giving a component model based on SysML using two viewpoints: structure and power usage, through modeling of state changes and used clock signals. Lévêque et al [90] describe ProCom, a component model based on C code with support for annotation of extra-functional properties in the form of WCETs, relying on flattening operations for composition. [39] describe DiaSuite, an environment focusing on software structure, exception handling and timing. [63] describes an approach based on aspect weaving and the Hulotte component model, joining different aspects on the code-level interfaces. An approach annotating components with resource usage, mainly memory usage and timing, is described in [97].

Finally, Bureš et al [23] suggest an approach where component models form families, similar to the concept of the semantic lattice discussed in this paper. A similar approach, but more generic, is described also by [46] which discusses the challenges of component composition, suggesting an abstract layered model of components with three main perspectives: behavior, interaction and execution.

D.6.5 Tool and Process Integration

Within tool integration, many approaches have had a more hands-on approach. Similar to the idea sketched in [80], they provide an environment consisting of different model transformations, possibly linked by automation support.

UML [70] provides a concept of packages, and also concrete relations to use for modeling relations between different packages. There is however no explicit semantics connected with these relations, making it an approach based on manual update of the modeled relations.

One further example of approaches in this area is [75], which uses dependency links between different properties in separate views. It differs between synthetic and analytic properties, i.e. design input and vs. analysis output. By doing so, it establishes relations between the different views. The paper also defines different levels of making the modeling of these dependencies explicit.

Other approaches include different process modeling efforts. A number of different formalisms exist to describe different process modeling languages. [10] Most of these focus very closely on the development process and hence only reference e.g. applicable model transformations. There are however a few also at least partially reference the semantics of the content of the views being worked on, including TIL [18] and FTG+PM [64], which both reference the meta-models/formalisms applied at different stages of the described process.
D.6.6 Semantic mapping

There are a number of papers focusing on more or less formally explicitly modeling the semantics of modeling languages, either in the context of views and viewpoints or not. [20] formally defines requirements on multi-modeling languages in general, describing the needs for environments where several parallel modeling languages have (partly) overlapping semantic domains. They do this by formalizing the semantic mappings. They

Other approaches take a more hands-on approach where the main goal is to implement a tool. Baresi and Pezzè [7, 8] describe an approach where a specific shared semantic domain is chosen, and all relevant domain-specific modeling languages are mapped to that using graph transformation systems. In their case, that shared semantic domain is Petri nets. They apply their approach to LEMMA, a domain-specific language for modeling medical diagnostic processes. [42] describe a tool based on a weaving approach with a mapping to state machines of the OMG meta-modeling framework.

Further examples of this type include [85], which informally introduce the concept of a semantic lattice of modeling formalisms from an implementer's perspective. By building semantic mappings between differently expressive formalisms, a semantic lattice is created.

The approach of semantic mediation and ontologies also take an explicit route of building semantic mappings between different modeling languages. Traditionally, creating an ontology has implied creating a library of shared (identical) concepts, which not necessarily covers both domains completely.

Similarly, [6] uses explicit modeling of semantic relations (≡, ⊆ and ⊥, all with identical meaning as in this report) as a way to relating different architecture description languages (ADLs) to each other. By doing ontology mappings between the different languages, they can be compared. Even though the paper gives an example using the COSA component model, its composition is not given.

D.7 Conclusions

This report has presented an approach for viewpoint decomposition inside multi-viewed components. It has given principles and guidelines on how to define multi-view components in a coherent manner and the necessary formalization to also keep them consistent while remaining composable.

The first important principle is that viewpoints have semantic relations to each other. When choosing viewpoints to describe a system, the choice of viewpoints should not only depend on the stakeholders’ interest, but also on these semantic relations, such that the viewpoint decomposition of the system creates as little data dependencies as possible, and the ones that are created should be as simple as possible. The implication of this is that composability is a property not only relevant to reasoning about relations between components, but also for relationships between viewpoints.

We further make a first step towards unifying the model-based and component-based design paradigms, in viewing components as collections of views (i.e. models) of the component. From the component-based paradigm we take the practice of dividing the system design by system structure, and from model-based design and architecture design, we take the principle of dividing the system design by stakeholder concerns and technical aspects. In comparison to pure model-based design, further also composition operators need to be defined in order to ensure the requirements on composability of views, as
inspired by component-based design.

Based on this, a few important principles have been seen as vital for achieving efficient implementation of multi-viewed component models:

- hierarchical viewpoint refinement,
- explicit viewpoint referencing,
- proper viewpoint decomposition, and
- explicit conflict handling between viewpoints.

In comparison to many previous approaches to multi-view modeling in components, we have decided to take a middle track between having a predecided and fixed set of viewpoints, and having a very open set without any further guidance. Having a fixed set is normally not realistic in industrial settings; indeed, it is our opinion that it is probably not feasible to find an exhaustive list of applicable viewpoints even in specific settings. On the other hand, having a very open situation and no guidance at all is neither as desirable: viewpoints with an ad-hoc definition tend to have messy dependencies and relations to each other. This is especially true if the viewpoints are defined in a heterogeneous tooling landscape, where the component abstractions are emerging effects of the underlying domain tools used in the development process.

We have in this report not exclusively used a traditional meta-modeling approach, for a few different reasons: to begin with, traditional meta-modeling does not supply semantic mappings. Hence, regardless, semantics would have to be given in a separate notation. Another reason was to bridge the gap between model-based and component-based approaches by not choosing a specific approach from either of them.

The approach has a number of advantages compared to previous work on viewpoints in component models:

- Viewpoints are only explicitly synchronized when there is information shared between them. This means that where applicable, viewpoint consistency is ensured, either through viewpoint orthogonality, through an explicitly defined semantic lattice, or through an explicit (manual or automated) consistency check.
- The explicit formalization of relations between viewpoints also means that the fact that data is lost in transformations between viewpoints is made more explicit. In the case where there is no explicit synchronization between all viewpoints (for example in a tool chain), reasoning can be made about the information loss between different views of the same component.
- The approach assumes no specific approach to semantics but can handle both formal (denotational or operational) and informal semantics.
- The approach hence finally makes any need for explicit, manual, synchronization of the content of two views explicit.

D.7.1 Future Work

As the current report only has given the main principles of the approach, there is still plenty of future work to be done based on the work presented here:

- Extending the framework and formally modeling (instantiating) more realistic and expressive component models.
• Using the framework created to closely compare related existing component models and currently used tools, and try to find relevant common shared viewpoints between them.

• Performing a existing real-world industrial case study – i.e. building viewpoints suitable for design of an industrial embedded system already in production.

• Using the formalization supplied in this report as the formal basis for a domain-specific language describing the semantic relations between different artifacts in tool integration chains.

• Utilizing the formalization of semantics into a semantic lattice in the evaluation of tool integration quality, possibly using the above mentioned domain-specific languages for process description [17, 18, 64], by integrating with formalisms that describe development processes and information flow between different tools in a tool-chain.

Acknowledgment

This work was partially supported by the European Commission within the project CESAR which is funded by the ARTEMIS Joint Undertaking under the Grant Agreement No. 100016 and by national programs/funding authorities.

This work was also partially supported by the national project ESPRESSO funded by Vinnova.

References


arnumber=1517771.


Multi-Viewed Components

(INCOSE IL 2013), 2013.


[95] Stavros Tripakis, Christos Stergiou, Chris Shaver, and Edward A Lee, A Modular Formal Semantics for Ptolemy, in Mathematical Structures in Computer Science, accepted for publication.


