Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report

Malte Heithoff
heithoff@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Nico Jansen
jansen@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Jörg Christian Kirchhof
kirchhof@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Judith Michael
michael@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Florian Rademacher
rademacher@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Bernhard Rumpe
rumpe@se-rwth.de
Software Engineering, RWTH Aachen University
Germany

Abstract

In modern systems engineering, domain experts increasingly utilize models to define domain-specific viewpoints in a highly interdisciplinary context. Despite considerable advances in developing model composition techniques, their integration in a largely heterogeneous language landscape still poses a challenge. Until now, composition in practice mainly focuses on developing foundational language components or applying language composition in smaller scenarios, while the application to extensive, heterogeneous languages is still missing. In this paper, we report on our experiences of composing sophisticated modeling languages using different techniques simultaneously in the context of heterogeneous application areas such as assistive systems and cyber-physical systems in the Internet of Things. We apply state-of-the-art practices, show their realization, and discuss which techniques are suitable for particular modeling scenarios. Pushing model composition to the next level by integrating complex, heterogeneous languages is essential for establishing modeling languages for highly interdisciplinary development teams.

CCS Concepts: • **Software and its engineering → Model-driven software engineering**; **Domain specific languages**.

SLE '23, October 23–24, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0396-6/23/10…$15.00
https://doi.org/10.1145/3623476.3623527

Keywords: Software Language Engineering, Language Composition, Domain-Specific Languages, Language Families, Reuse, Internet of Things, Assistive Systems

ACM Reference Format:
Malte Heithoff, Nico Jansen, Jörg Christian Kirchhof, Judith Michael, Florian Rademacher, and Bernhard Rumpe. 2023. Deriving Integrated Multi-Viewpoint Modeling Languages from Heterogeneous Modeling Languages: An Experience Report. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE '23), October 23–24, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3623476.3623527

1 Introduction

Software and systems engineering faces an increasing level of complexity as we have to handle the increasing complexity of the world. Using modeling approaches has proven to be a suitable approach to handle this complexity [86]. To create models of reality for domains such as production [10, 32], automotive [87], and medicine [77], to be used in, e.g., digital twins [36], for explainable cyber-physical systems [9], or complex systems-of-systems, it is necessary to consider a range of perspectives and viewpoints. This requirement is commonly known as multi-viewpoint modeling, which entails addressing different properties of systems for the diverse disciplines involved in an accessible fashion.

One approach to meeting the specific needs of particular disciplines in their engineering efforts is to use Domain-Specific Languages (DSLs). Although such DSLs can be employed simultaneously for different use cases, in practice, they often cover only a single viewpoint if not further supported by tooling, such as projective approaches. As a result, also considering that a single DSL often cannot suit every use case alone, this requires combining several languages to achieve a more holistic view of a system. To address this issue, researchers have proposed various techniques, such

as using multiple DSLs, developing a unified language that covers different viewpoints such as UML or SysML, or using language workbenches, which provide an integrated environment for designing, implementing, and using DSLs.

It can be argued that the number of modeling languages in different domains is steadily increasing [15, 20, 59, 66] which, next to maturity and evolution, raises the question on how to integrate these languages—not only for coping with the complexity of contemporary software systems that consist of many heterogeneous parts, but also for the sake of *reuse* [80]. Reuse in software engineering benefits, among others, (i) quality by gradually accumulating error fixes; (ii) productivity by decreasing the demand for new software; and (iii) reliability by increasing the chance to find errors through higher usage rates [78]. However, a key ingredient for software reuse is the establishment of interoperability between heterogeneous components, e.g., by means of mutually agreed interfaces. These reuse considerations also apply to the DSLs and their related tooling. When integrating heterogeneous modeling languages, we aim to reuse both, the languages or parts of the languages themselves as well as the already developed or generated tools such as parsers, pretty printers, or full generators. We study the integration of heterogeneous modeling languages by leveraging different mechanisms for language composition. This allows us to make independently developed languages reusable, achieving different viewpoints at the model level for the distinct application domains.

This experience report tackles the research question of *how to integrate different modeling languages via established language composition techniques achieving multi-viewpoint modeling languages.* In this paper, we elaborate on our experiences from two case studies of complex, real-world, software-intensive systems and show which language composition methods are applied there. One is a language family for model-driven development of IoT applications. The other language family is used to support the model-driven engineering of assistive systems and to use models at runtime of the system. Additionally, we discuss the different mechanisms for language composition used and detail our experiences on which techniques were suitable for which cases and whether they contribute to establishing multi-viewpoint modeling. For our studies, we use the MontiCore language workbench [45] as it comes with various composition techniques.

Structure. Sec. 2 provides background information for our approach. Sec. 3 discusses related work for the composition of modeling languages, language workbenches with composition support, and modeling languages. In Sec. 4, we introduce two use cases from complex, real-world, software-intensive systems, namely to develop IoT systems and assistive systems. Sec. 5 discusses the application of the different language composition approaches in our modeling scenarios and their contribution to achieving a multi-viewpoint modeling environment. The last section concludes.

2 Background

We provide relevant background on model-driven engineering as well as language composition mechanisms.

2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) [18] is a software engineering paradigm that promotes the use of models as first-class citizens in all or selected phases of the software engineering process. In the sense of MDE, a model is a software artifact that abstracts from certain details of a software system, and can replace specific parts of the system for certain purposes such as implementation, testing, or simulation.

Next to model application, MDE also systematizes model construction, evolution, and maintenance. All of these activities require an unambiguous notion of model validity that is commonly defined by the language in which a model is expressed [18].

To this end, a modeling language consists of (i) an abstract syntax that specifies the essential information of models independent of their representation; (ii) a concrete syntax that specifies the user-facing representation of model elements; and (iii) a semantic associating each model element with a meaning [18, 44].

Language workbenches [29] denote IDEs that bundle tools for modeling language construction, e.g., meta-grammars, parser generators, and language composition facilities. Examples of contemporary language workbenches include MPS [70], Xtext [30], and MontiCore [45]. Due to its mature support for a variety of language composition mechanisms, we henceforth leverage MontiCore to study the derivation of multi-viewpoint modeling languages by language composition.

MontiCore is a language workbench [45] whose EBNF-like [88] meta-grammar allows the specification of grammars for modeling languages with textual concrete syntaxes. From a language grammar expressed in its meta-grammar, MontiCore is able to generate (i) the implementation of the corresponding abstract syntax in the form of a metamodel; (ii) the parser infrastructure to instantiate the metamodel from input files adhering to the grammar; and (iii) additional infrastructure for common concerns in modeling language implementation such as context condition checking, symbol table management, and template-based code generation. With its generative approach, MontiCore effectively reduces the effort in modeling language implementation. In addition, and by contrast to other language workbenches, MontiCore's meta-grammar also provides constructs for modeling language composition [13], thereby facilitating the integrated

evolution of a modeling language from a single source artifact, i.e., the language's grammar.

Given its versatility, MontiCore is actively used to create and maintain DSLs targeting heterogeneous domains such as automotive [24], cloud services [27], Internet of Things [53], robotics [1], and systems engineering [19].

2.2 Language Composition Mechanisms

For the efficient engineering of modeling languages, MontiCore especially focuses on compositional language design. To this end, MontiCore supports multiple language composition techniques and corresponding design patterns, enabling combining multiple DSLs [26]. Furthermore, it provides an extensive library of language components [12] serving as a common foundation for building more sophisticated modeling languages. Overall, MontiCore supports four different types of language composition realized either directly via the language specification, i.e., the grammar, or indirectly via the symbol table infrastructure [14] of a language.

To explain the various composition techniques, we consider a number of languages developed in MontiCore's ecosystem. The following language definitions are simplified versions for clarity reasons and space limitations. The original sources are referenced accordingly.

First, we introduce a simple automata language. Such an automaton gradually processes letters of an input alphabet. A sequence of letters (i.e., a word) is accepted if there exists a path to a final state. Otherwise, the word is rejected. Overall, the automaton represents the set of words it accepts. Figure 1 contains the grammar of the automata language[1]. An `Automaton` (l. 02) starts with the respective keyword, has a name, and consists of multiple states and transitions (l. 03). A `State` (also indicated via a corresponding keyword) has a name (l. 05) and can be marked as «`initial`» or «`final`» (l. 06). Finally, a `Transition` (l. 08) describes the change from the source (`src`) to a target (`tgt`) state via an `input` letter enclosed in an arrow-like syntactical structure.

```
01 grammar Automata extends MCBasics {                    MG
02   symbol scope Automaton =
03     "automaton" Name "{" (State | Transition)* "}" ;
04
05   symbol State = "state" Name
06     (("<<" ["initial"] ">>") | ("<<" ["final"] ">>"))* ;
07
08   Transition = src:Name "-" input:Name ">" tgt:Name ";";
09 }
```

Figure 1. Simplified version of MontiCore's automaton language[1] for modeling non-hierarchical automata with states and transitions.

The production rules of the automaton language employ predefined constructs such as the `Name` token (cf. l. 03). This usage already indicates the first application of language extension as this respective token comes from the base grammar `MCBasics`, which is extended by the automaton language (l. 01), importing its productions.

The next DSL under consideration is the class diagram language `CD4Analysis` used to describe data structures consisting of classes and their attributes. Its context-free grammar is depicted in Figure 2. Please note that the original specification[2] is designed in a highly compositional fashion, further modularizing the different constituents. Thus, the Class Diagram (CD) languages in this paper are simplified versions that roughly sketch the structure and are tailored to explain the distinct composition techniques.

The root node of a class diagram model is the `CDCompilationUnit` (l. 02). It consists of a package declaration and a set of import statements, two inherited properties. It entails a `CDDefinition`, representing the actual diagram (l. 03). The `CDDefinition` (l. 05) depicts the start of the diagram via a corresponding keyword. It has a name and comprises multiple elements contained in curly brackets. These elements are specified by the interface nonterminal `CDElement` (l. 07). This interface can be implemented by other nonterminals, thus serving as an explicit extension point. In this grammar, the only element implementing it is the `CDClass` (ll. 09-12) that has a name and comprises multiple `CDAttributes`. In turn, these attributes (l. 14) consist of a type and a corresponding name.

```
01 grammar CD4Analysis extends MCBasics, MCBasicTypes     MG
02   CDCompilationUnit = MCPackageDeclaration
03     MCImportStatement* CDDefinition;
04
05   CDDefinition = "classdiagram" Name "{" CDElement* "}";
06
07   interface CDElement;
08
09   symbol scope CDClass implements CDElement =
10     "class" Name "{"
11       CDAttribute*
12     "}";
13
14   symbol CDAttribute = MCType Name;
```

Figure 2. Simplified version of MontiCore's class diagram language[2] for modeling the data structure of a system via classes and their attributes.

2.2.1 Language Inheritance.

The first composition technique of MontiCore is language inheritance. Here, the constructs of an original language are adopted and extended or modified for a new use case. While the original language remains unchanged, the new DSL incorporates concrete and abstract syntax, as well as the generated tooling and its handwritten extensions.

[1]Automata language definition available at: https://github.com/MontiCore/automaton

[2]Compositional class diagram language definition available at: https://github.com/MontiCore/cd4analysis

Figure 3 shows an example of language inheritance by an extended class diagram language. The inheritance relation is indicated by the `extends` keyword (l. 01) followed by the corresponding host language name that should be adopted. As the overall structure of a class diagram remains unchanged, we keep the starting nonterminal `CDCompilationUnit` (l. 02). In addition to the adopted constructs, we expand the language by modifying or adding production rules for modeling elements. Thus, the nonterminal `CDClass` is overwritten (ll. 04-08) to enable basic method signatures in the class body in addition to the already existing attributes. For these signatures, we introduce the corresponding nonterminal `CDMethod`, which describes the syntax of the element (ll. 10-11) and can be referenced in other production rules. Besides modifying the class contents, the extended language also introduces interface definitions (ll. 13-16) and enumerations (ll. 18-23) as new diagram elements.

```
01 grammar CD4Code extends CD4Analysis {                    MG
02   start CDCompilationUnit;
03
04   @Override
05   symbol scope CDClass implements CDElement =
06     "class" Name "{"
07       (CDAttribute | CDMethod)*
08     "}" ;
09
10   symbol CDMethod =
11     MCType Name "(" (argT:MCType argN:Name)* ")" ";";
12
13   symbol scope CDInterface implements CDElement =
14     "interface" Name "{"
15       CDMethod*
16     "}" ;
17
18   symbol scope CDEnum implements CDElement =
19     "enumeration" Name "{"
20       (EnumLiteral || ",")*
21     "}" ;
22
23   symbol EnumLiteral = Name;
24 }
```

Figure 3. Extended class diagram language[2] including methods, interfaces, and enumerations. Application of language inheritance with conservative extension.

2.2.2 Language Extension.

Language extension is a specific, more restrictive form of language inheritance. This technique also takes over all constituents of a host language. The difference is that changes are only allowed in the form of conservative extension. This means that only new elements may be added to a language or existing elements may only be modified in an extending but non-restricting way. Thus, valid models of the original language still remain valid in the context of the extended variant.

In fact, the inheritance example in Figure 3 features only conservative extensions. Adding further elements such as `CDInterface` or `CDEnum` only extends the set of valid sentences. Also, despite being overwritten, the altering of `CDClass` remains conservative as it further introduces methods inside a class without impacting the use of their attributes (l. 07). Figure 3 also illustrates the benefit of languages being intentionally tailored for their extension (or the drawbacks if not). Thus, while adding methods to classes via overwriting the production is possible, it includes lots of duplication in the production rules. This is a considerable overhead for adding a single reference inside a production rule. In contrast, adding `CDInterface` and `CDEnum` to the overall diagram yields no overhead. The difference is that the newly introduced nonterminals implement the already existing interface nonterminal `CDElement` (cf. Figure 2, l. 07) of the original language. This element is an explicit extension point that supports inheriting languages to weave new constituents into existing production rules. In this case, the original `CDDefinition` (l. 05) references the interface, allowing all incarnations as valid CD elements. Thus, designing a language with extension in mind can significantly improve the engineering of further variants.

2.2.3 Language Embedding.

Language embedding integrates multiple DSL definitions, combining their production rules in a single grammar, enabling integrated modeling via their combined constituents. Therefore, this technique not only collects the entirety of nonterminals of all included languages but automatically combines their usages concerning shared interface definitions and usages. This is especially useful for integrating default language components, such as expressions, into an existing DSL. A language component is a (possibly incomplete) definition comprising a grammar, corresponding generated and handwritten artifacts, as well as an integration interface established via predefined extension points. They constitute a decoupled set of reusable standard productions explicitly tailored for embedding. Technically in MontiCore, language embedding employs multiple inheritance. Thus, embedding a language into another is as simple as extending both in the grammar signature.

```
01 grammar MealyAutomata extends Automata,                  MG
02         CommonExpressions, AssignmentExpressions {
03
04   MealyAutomaton = MCImportStatement* Automaton;
05
06   @Override
07   Transition =
08     from:Name "-" input:Name "/"
09     output:Expression ">" to:Name ";" ;
10 }
```

Figure 4. Language for modeling mealy automata, non-conservatively inheriting from the automaton language and embedding expressions.

Figure 4 contains an example of embedding expressions into the already established automata language by extending both definitions (ll. 01-02). Simultaneously, the new language

adds the possibility to import other artifacts (l. 04) and advances the automaton to mealy machines (i.e., processing an input and producing a corresponding response action). The latter is achieved via overwriting the production rule of the `Transition` (ll. 06-09) and further separating the `input` from a newly established `output` expression, separated by a slash. As this addition is not optional, the language is extended in a non-conservative way, i.e., original automaton models are not valid anymore in this variant. Realizing the output as an expression enables arbitrary terms of all embedded languages, such as `s1 - a / (x == 4+3) > s2`, indicating a state change from `s1` to `s2` on the input `a` and triggering the action evaluation for the boolean expression `x == 4+3`.

2.2.4 Language Aggregation. Language aggregation enables integrating models of multiple DSLs while simultaneously keeping them as separate artifacts. In contrast to embedding, the technique of aggregating languages only loosely couples DSL definitions and makes them operable in a common context. This inter-operationality is achieved via MontiCore's symbol table infrastructure, allowing cross-referencing, even over multiple artifacts.

```
01 import games.Tennis.*;                                   Aut
02 automaton PingPong {
03   state NoGame <<initial>> <<final>>;
04   state Ping;
05   state Pong;
06
07   NoGame - start / strokes=0 > Ping;
08   Ping - returnBall / strokes++ > Pong;
09   Pong - returnBall / strokes++ > Ping;
10   Ping - missBall / p2_points+=strokes > NoGame;
11   Pong - missBall / p1_points+=strokes > NoGame;
12 }
```

```
01 classdiagram games {                                     CD
02   class Tennis {
03     int strokes;
04     int p1_points;
05     int p2_points;
06   }
07 }
```

variable symbols of CD in the context of a PingPong game automaton

Figure 5. Models of the extended automaton and CD DSLs used in a shared context via language aggregation.

Considering again the extended automata language in Figure 4 introduced for language embedding, we can further extend its usage aggregating class diagrams. The main idea here is to combine a structural and a behavioral language. Figure 5 presents the composition of two exemplary models, preserving them as separate artifacts. At the top, we have a simplified automaton model of a customized `PingPong` game featuring three states (top, ll. 03-05) and five transitions for depicting the gameplay (top, ll. 07-11). The second model is a class diagram defining a set of variables for counting the `strokes` and the points of two respective players (bottom, ll. 03-05). Furthermore, the automaton imports the class diagram (top, l. 01), making all types and variables accessible. Thus, the embedded expressions at the transitions can be employed to reference the externally defined variables[3], e.g., for incrementing the number of strokes when the ball is returned (top, ll. 08-09) or assigning points to a player when the other misses the ball (top, ll. 10-12). That way, the embedded expressions are additionally employed to reference symbols of other models, enabling a seamless composition over multiple artifacts.

3 Related Work

This section presents work related to the composition of modeling languages (Sect. 3.1), language workbenches with composition support (Sect. 3.2), and families of modeling languages derived by language composition (Sect. 3.3). For a comprehensive overview of model view approaches, we refer the readers to [11].

3.1 Composition Approaches

A straightforward approach to modeling language composition is the exploitation of inter-model references [89]. To realize this approach, the referring modeling language must integrate modeling concepts by which it can establish links to instances of another modeling language's concepts. On the model-level, such links appear as references from elements in the referring model to elements in the referenced model. While this approach is versatile, e.g., it can be retrofitted into the referring language without impacting the referred language, it requires the user to comprehend models in different languages, scatters information across different models, does not enable directed alteration of elements in referred models, and may result in invalid models when referred elements are changed independently of the referring model.

A more sophisticated approach to reference-based language composition is the conversion of technology-specific language metamodels into more abstract representations. This enables the specification of correspondence relationships between concepts from heterogeneous modeling languages [48]. These correspondence relationships may anticipate conversion rules between modeling concepts, thereby making links between model elements actionable, e.g., to base the validity of a correspondence relationship on the concrete peculiarity of a referred model element. However, the same drawbacks as for inter-model references apply.

Modeling language variability [41] constitutes an approach to language composition when considering the base modeling language as one language and the delta sets of language concepts derived from activated base language features as their own languages. While modeling language variability can anticipate all possible language compositions and provide exhaustive tooling from the

[3]We do not distinguish between type and instance level in this example for simplicity reasons.

beginning, composition is constrained to the supported features of the base language.

A base language may also explicitly delegate the realization of certain modeling concepts to independently developed languages that provide concrete support for such concepts [35, 55]. This approach requires a priori reasoning about compositionality by base language designers as well as meta-languages or meta-operators that systematize the delegation of modeling concept realization. Furthermore, it may be necessary to add additional glue code for concept realization in order to align the semantics of concept-realizing languages with that of the corresponding base languages.

3.2 Language Workbenches with Composition Support

Melange [21] is a language workbench that supports model-first composition on the level of metamodel concepts including their operational semantics. However, composition of language artifacts besides metamodel implementations is out of Melange's scope. Similarly, MetaEdit+ [83] allows language integration via references between metamodel concepts. MPS [85] inherently exhibits support for model-first composition because language definition is also model-first. Concepts from the abstract syntax of a composed language can thus embed, extend, or adapt concepts from the abstract syntaxes of other languages, including related tooling like code generators.

Xtext [8] is a language workbench with grammar-first composition support, i.e., its meta language for grammar specification and subsequent metamodel derivation integrates keywords for language composition. Specifically, Xtext supports composition by importing the rules of an independent language grammar into a composed grammar and the derivation of new languages by leveraging the rules of a base language as an initial, yet extensible, rule set. Similarly to Xtext, Neverlang [16] is a grammar-first language workbench, which provides a more fine-grained, but rather complex, support for language composition based on language modules, roles, and role slicing for language feature specification.

3.3 Composed Modeling Language Families

Modeling language composition fosters the systematic creation of *modeling language families*, which constitute sets of two or more integrated languages, to enable the application of MDE to coherent parts of a problem domain.

Inter-model references (Sect. 3.1) represent a flexible means of generic language creation because they can be used to non-intrusively relate modeling languages, thus giving rise to language families by enriching or constraining modeling syntaxes and semantics. For example, reference-based composition allows for (i) constraining modeling syntaxes or model element peculiarities by linking metamodel concepts with invariants expressed in another language [72, 82]; or (ii) making relationships between diverse parts of a software architecture explicit, thereby fostering architecture comprehension and reasoning [34, 74].

When being based on a more abstract representation acting as an intermediate language to bridge between heterogeneous language concepts, inter-model references also foster independent evolution of composed languages and extensibility of language families [79]. Similarly, they facilitate the creation of modeling languages whose concepts are tailored to domain expert concerns but map to other languages' concepts of a different domain, e.g., to generate executable code [76].

Modeling language families derived from variability-based language composition (Sect. 3.1) often consist of *sibling languages*, or sub-families of such siblings, that are immediate descendants of the base language. They can be automatically derived by modelers and afterwards applied to related, yet slightly different problem sets, in the target domain [54, 57, 90].

4 Case Studies from Complex, Real-World, Software-Intensive Systems

For a better understanding of the possible uses of the different language composition techniques, we describe two specific case studies. Their size and complexity show, why different composition techniques are needed in practice.

4.1 IoT Systems

The Internet of Things connects objects with each other and with the Internet. Applications of the IoT include both industrial and consumer sectors and range from connected vehicles (and fleet tracking), to Asset Tracking, to Smart Homes. To do so, these objects are equipped with sensors and actuators. As inherently distributed applications, the development of IoT systems requires different skills than the development of classic software systems such as smartphone apps [81]. One way to manage the heterogeneity and complexity of IoT solutions is to use model-driven techniques [28, 62, 67], as they raise the level of abstraction.

MontiThings [53] is a language family for model-driven development of IoT applications. MontiThings covers the design, deployment [50], and analysis [51, 52] of the applications generated from the models. MontiThings aims to simplify the complicated development of IoT applications and abstract from the heterogeneity of IoT devices. To separate concerns and not mix, *e.g.,* technical details with high-level business logic, MontiThings consists of a family of multiple languages. The core of MontiThings is a component-and-connector (CnC) architecture description language that is used to describe the business logic of the applications. From the models of this language, MontiThings generates the C++ code for distributed applications and the scripts to package

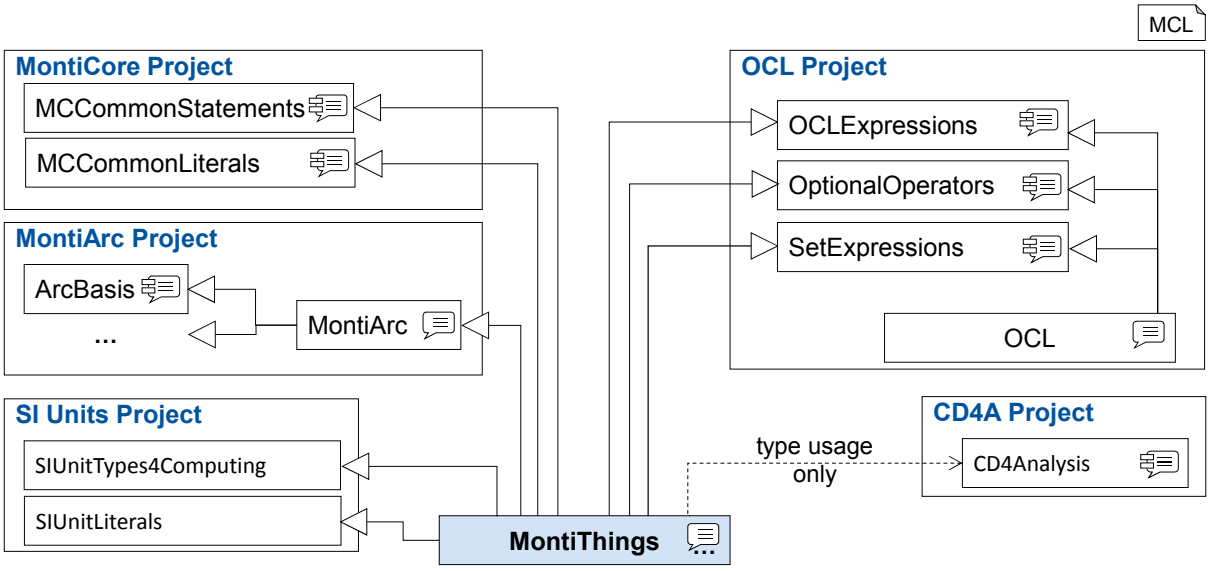

Figure 6. An overview of languages integrated by the MontiThings language (adapted from [49]).

them in the containers. Shared infrastructure such as interfaces to message brokers or the serialization of messages is provided by MontiThings. If the generated code does not meet the user's requirements, it is possible to supplement or overwrite the code with handwritten code using MontiCore's TOP mechanism. In addition to this main language, other languages exist, *e.g.,* for configuring models for specific hardware/software platforms or for defining tests.

The MontiThings language is composed of 46 grammars of the MontiCore project reusing 4371 lines of grammar [49]. Particularly noteworthy is the fact that the combined languages are not just smaller subgrammars, but modeling languages in their own right. Figure 6 shows a reduced overview of MontiThings' grammar reuse. In doing so, MontiThings uses several types of language composition: Language extension (and, thus, language inheritance), language embedding, and language aggregation (*cf.* [42]). Most importantly, MontiThings extends MontiArc [43]. MontiArc is a CnC architecture description language for simulating distributed systems. Consequently, MontiArc provides large parts of the abstract and concrete syntax of MontiThings' CnC language. However, MontiThings' generator is very different from MontiArc's generator, because MontiThings is focused on generating applications intended to be executed on real IoT devices while MontiArc is focused on simulations. In addition to the inherited language elements, MontiThings extends MontiArc with numerous elements, *e.g.,* for error handling [53] to deal with often unreliable sensor input.

MontiThings' type system is mainly based on MontiCore's Java-like type system. For primitive types, MontiThings reuses MontiCore's primitive types through language embedding. Furthermore, MontiThings uses MontiCore's SI unit language via language embedding. This enables modelers to use SI units like primitive types, *e.g.,* define a variable of type `km/h` or `°C`. If two convertible but different units are calculated together (*e.g.,* `km/h` and `m/s`), MontiThings automatically ensures that the units are converted into the same unit. More complex data structures can be defined in class diagrams. The class diagrams are specified in their own files. MontiThings can import the symbols defined in the class diagrams via language aggregation. Using aggregation, loose coupling and separation of concerns can be achieved, resulting in isolated yet synchronized views between the architectural models and their referenced types. Thus, the CD4A language could simply be replaced by another data type language as long as it conforms to MontiCore's type system. For example, if a class diagram defines a type `Photo`, variables, ports, and other elements in MontiThings models can use the type `Photo` if the artifact imports the corresponding class diagram. Besides the type system for variables, MontiThings also reuses MontiArc's type system for specifying component types. Most parts of the type check could be reused 1:1. Only in cases where the combination of languages results in new cases that the type checks of the individual languages cannot handle or languages deviate from MontiCore's defaults, individual manual adjustments have to be made.

The behavior of MontiThings components can be defined using four techniques:

1. Subcomponents
2. A Java-like language
3. Statecharts
4. Handwritten code (C++ or Python)

The ability to instantiate and connect subcomponents is a capability that the MontiThings language inherits from MontiArc. The Java-like behavior language is included in MontiThings through language embedding and is based on MontiCore's MCCommonStatements. Statecharts are another way of defining behavior. Similar to the MCCommonStatements, they are included in MontiThings via language embedding. The embedded languages are augmented with other MontiCore languages such as the OCL for writing boolean expressions. In this regard, MontiCore's common foundation of types and symbols reduces the effort of integrating languages. By providing a common denominator for common symbol types (functions, variables, etc.), languages can be reused largely unchanged, reducing the need to write adapters. Integrating OCL expressions only required us to include the statements once for the whole language, while the concepts applied to multiple locations within the language (*e.g.,* both Pre- and Postconditions, as well as within the statement language). An alternative way of defining behavior is through handwritten code. As always in MontiCore, generated code can be overridden using MontiCore's TOP mechanism. Besides overriding generated classes using C++ code via the TOP mechanism, MontiThings also has the ability to integrate Python code for behavior. In this case, MontiThings serializes data sent via ports using Google Protobuf[4] and exchanges this information with a generated Python wrapper that forwards the data to the handwritten code.

Besides its main language, MontiThings also includes a separate language for configuring components for specific targets. This language acts as a tagging language (*cf.* [40]), adding extra properties to components and ports. For example, the configuration language can be used to define that the code generator should treat a component as a single deployment unit that includes all its subcomponents instead of creating its subcomponents as independent services. This can be used, *e.g.,* to reduce communication overhead if a component is expected to be deployed on the same device. Furthermore, different variants of a component for different target platforms can be defined. For example, if the generated code is expected to be deployed on an Arduino it might require other handwritten code for accessing sensors than code intended to be deployed on a Raspberry Pi.

Moreover, the MontiThings project landscape includes a testing language inherited from MontiCore's sequence diagrams. The testing language uses MontiCore's resolving delegate mechanism to refer to symbols from the MontiThings models under test. It enables users to define white box test cases using sequence diagrams that model the interaction between a component's subcomponents. It is of course also possible to only define in- and outputs of the test case to define a black box test. A code generator independent of MontiThings' main code generator uses the sequence diagrams for C++ code transformations written against the GoogleTest framework.

The configuration and testing languages inherit from MontiThings and MontiArc, respectively. While language aggregation over symbols would have led to a better separation of languages, and external, exchangeable views, inheritance avoided the development effort of importing the symbol table. The disadvantages of this approach are the bad reusability (because of high coupling) and the long compile time of the tagging languages.

4.2 Assistive Systems

Assistive systems play an important role in ensuring safety and supporting individuals in a variety of settings, including work [61, 77, 91], driving [87], and daily life activities [5, 60, 64]. To be able to provide human behavior support, an assistive system needs context information [63] as well as behavior data, e.g., via activity recognition systems [58], both previously stored and real-time monitored [46]. After analyzing and reasoning about this information [3, 56], support information is provided either in a situation a person needs it or when she asks for it.

We have investigated which modeling languages are needed to apply a model-driven approach for the engineering of assistive systems and which languages are needed to use models at runtime [7]. We have used the assistive system language family to develop assistive systems to support processes in a smart kitchen as well as processes for manual assembly in production.

For the model-driven approach, we use the MontiGem [2, 38] generator framework. MontiGem was developed to support the model-based engineering of web-based information systems. It uses models in the CD4A language as input to define domain information in the data structures, models in the GUI language [37] to define user interfaces and OCL to define constraints for user input. Out of these languages, it generates the backend, frontend, and database of a web application, in this case, the core of an assistive system. Additionally, we have added hand-written components to handle relevant information during runtime, e.g., to transform data into runtime models, to reason about information, or to create support information. The support information for end users includes full sentences (in the first version in German) as well as additional pictures and acoustic information for each defined task.

To use models at runtime of the assistive system, we have defined a language family for model-based assistive systems (see Figure 7). This includes a `ContextLanguage` to define concrete objects to be used in supported processes and a `TaskLanguage` to describe the processes in a textual way.

As the set-up of assistive systems for a concrete location and tasks is time-consuming, we have developed the `ContextLanguage`. It allows us to define what tools and

[4]Protobuf Documentation: https://protobuf.dev/, Last accessed: 11.04.2023

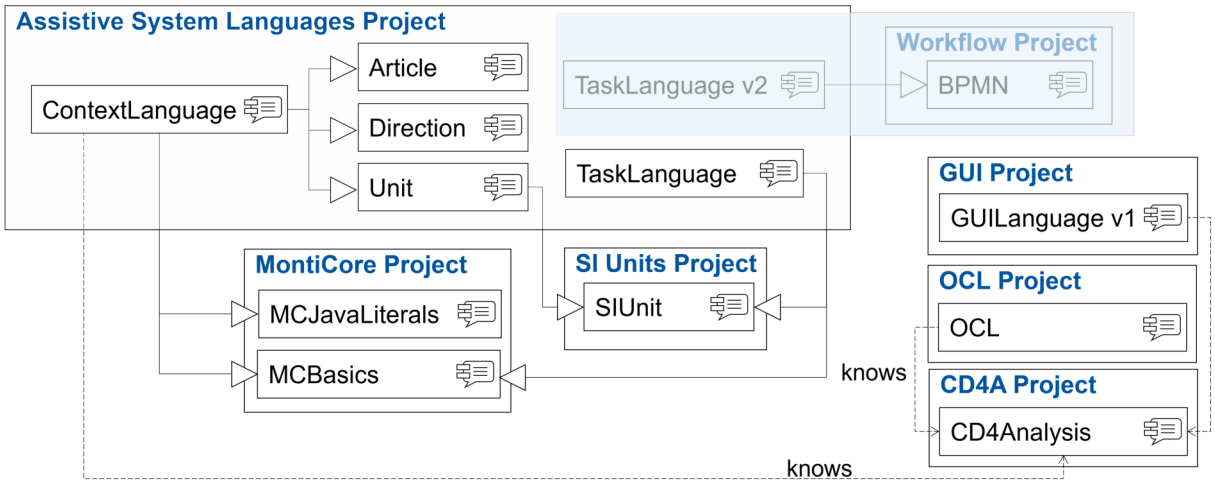

Figure 7. Overview of the Language Family for Assistive Systems

objects to use in supported tasks and where they can be found. More specifically, it allows defining objects in four object groups: Machines, Storages, Utensils, and Items. These groups describe the spatio-temporal and environmental context of a task. As we aim to use this language during runtime, these concepts also have to appear in the CD4A models as part of the concepts describing the domain. Thus, the use of the Context Language during the runtime of the assistive system relies on the existence of certain classes in the CD4A model. Figure 8 shows this relationship with an example. The concepts Machine (ContextLanguage Grammar l.9) and MachinePart (l.13) are defined in the CD4A model. As the CD4A model is used as input for generating the assistive system, we can use ContextLanguage models during runtime and store their information via data access objects in the backend into the database.

The `ContextLanguage` reuses the Java-like comments and names (`MCBasics`) and Java-like number specifications (`MCJavaLiterals`) provided by MontiCore through language embedding. This not only allows the reuse of terms like `2.2d` but also the reuse of MontiCore's type system, *e.g.,* to check whether the range in a `StepWiseComponent` is type safe (see Figure 9).

Further reusable parts relevant to the `ContextLanguage` were moved to new language components: The first version of the `Article` language component included a set of German definite articles. The `Direction` language component includes a set of phrases used to define directions relative to a certain place, e.g., left, in front of, or in the middle. To help write a more intuitive model, our languages use numbers in combination with units such as `3 l` for three liters, *e.g.,* to appropriately indicate quantities in a cooking recipe. The MontiCore `SIUnit` language fits most of our needs for the most common SI and SI-derived units [68]. Additionally, the `ContextLanguage`

```
01 grammar ContextDefinition extends                                    MG
02   de.monticore.MCBasics,
03   de.monticore.literals.MCJavaLiterals,
04   Article , Unit, Direction {
05   …
06
07   scope Machines = "Machines" "{"    Machine*  "}";
08
09   symbol scope Machine = identification:Name?
10     ":"? Article name:Name RelativePosition?
11     ("{" MachinePart+ "}")?;
12
13   symbol MachinePart = name:Name ":"? Article
14     partName:Name ","? ReferencePosition? ","?
15     (FunctionalComponent || ",")*
16     ("{" MachinePart+ "}")?;
```

```
01 class Machine {                                                      CD4A
02   String identification;
03   String name;
04 }
05
06 class MachinePart extends Resource {
07   boolean occupied;
08 }
09
10 association [1] Machine -> MachinePart [*];
```

Figure 8. Relationship `ContextLanguage` grammar concepts and CD4A model concepts

```
01 StepwiseComponent implements FunctionalComponent                     MG
02   = "modifies" "stepwise" "("
03     min:NumericLiteral "," max:NumericLiteral ","
04     stepSize:NumericLiteral ")" Controls?;
```

Figure 9. Extent of the `ContextLanguage` grammar embedding a MontiCore component grammar

needed to define its own set of `Units`, *e.g.,* `°Celsius` instead of `°C` or `EL` (common German word describing a spoon full of an ingredient). Here, the `Unit` component language inherits from MontiCore's `SIUnit` language

adding more informal units and removing others, *e.g.,* m^3. The `ContextLanguage` embeds these three language components. The concepts in these language components are needed later on to assemble correct German sentences based on model information.

In the generated assistive system, we want to inform the user by providing crucial information needed for the next task. In MontiGem, the web interface is modeled using the `GUILanguage v1` which we will use to model user interfaces. Models of this language refer to the data types defined in the corresponding domain model (`CD4Analysis`) to specify data access in the running system. `OCL` expressions constrain the domain model. The MontiGem framework is then responsible for generating infrastructure for data transfer and validators for each constraint. In our assistive system, we use one generic `GUILanguage v1` model which presents all necessary information to the user for one step of behavior. This model is accompanied by others for domain-specific presentation tasks, e.g., providing overviews, or allowing to change settings.

The `TaskLanguage` can be used to model human behavior tasks in sequential order. It is especially designed to be as user-friendly as possible with not only a task order but also a natural description of how to perform those. The core elements of such a sequential order are `Find` tasks to instantiate a findable item, `Placing`, `Filling` and `Setting` tasks to alter items, as well as `Waiting` and `Moving` tasks to give an order directly to the user. Here, we again embed `MCBasics` to use common names and comments, `MCJavaLiterals` for Java-like number usage. The `SIUnit` language is again used on multiple occasions, for which we restrict the usage in one place to only allow time units like `s` or `h`.

As, *e.g.,* recipes or manuals, describe tasks in sequential order, we have assumed that it is sufficient if the `TaskLanguage` supports behavior sequences. But more complex processes might require a lot of waiting where other tasks can be done in parallel or might require a (valid) reordering based on personal preferences. Modeling languages like UML activity diagrams or the BPMN standard [69] also allow specifying the parallel ordering of tasks/activities. Since the BPMN also specifically targets human interaction, we will inherit from the textual version of the BPMN standard [25] in the next version of the language, `TaskLanguage v2`. This allows us to specify more real-world suitable task ordering. For this, we will only allow user tasks (removing, *e.g.,* sending or service tasks) and extend them by the task types identified in the `TaskLanguage`. We will also need to additionally embed `MCJavaLiterals` and `SIUnit` to fit our user-friendly notation. With these steps, we achieve a new language suitable for our needs. Models of the `TaskLanguage v2` could then stand for themselves or can be used to generate multiple valid `TaskLanguage` models. In the generated assistive system, we could reuse a standardized workflow engine to manage our task definitions. To ensure backward compatibility, modeling in `TaskLanguage (v1)` could then be a starting point for a transformation in a sequential `TaskLanguage v2` model.

5 Discussion

For our investigation, we have considered the notion of *viewpoint* in its broadest sense, i.e., as a conceptual means for the model-based description and reasoning about different concerns pertaining to a software system. Our consideration of the notion is thus consistent with other publications in the MDE area [6, 17, 33, 39, 65, 75], and specifically with those at the intersection of MDE and software architecture [22, 23, 31, 47, 71, 73]. Recently, Multi-Paradigm Modeling (MPM), which has its roots in simulation [84], has been discovered to greatly benefit the MDE-based development of cyber-physical systems as it enables to model and subsequently process heterogeneous parts of the system with the most appropriate MDE formalisms and workflows [4]. Hence, we perceive multi-viewpoint modeling and MPM to constitute two sides of the same coin, both aiming to tackle complex system design, development, and operation by the integration of heterogeneous modeling languages. These languages' application eventually results in models that can be analyzed and processed leveraging well-understood MDE techniques such as quality analysis, model transformation, code generation, and simulation [18].

Constructing sophisticated modeling languages and language families in heterogeneous domain use cases delivered a comprehensive set of observations, which composition technique is applicable in particular scenarios. Overall, we summarize our experiences from the two presented case studies into seven potential language engineering scenarios, depicted in Table 1. Please note that the table only reflects the conceptual composition technique and not its implementation. That is, while in MontiCore, extension and inheritance are methodically different executions of the same mechanism, or embedding always incorporates inheritance as well, they are distinguished concerning their assessment.

Adapting a single, already existing language to a use case (S1) requires only inheriting from that language. Applying modifications to the original can either be achieved by conservative extension or via inheritance. The latter also allows for overriding or restricting productions, which, in some cases, might be necessary. However, this leads to the original models not being valid in the context of the modified language anymore. If original models must be retained (S2), only conservative extension is applicable. Overall, these composition techniques are the easiest to apply since constructs are directly adopted from an existing language. In the case of extension, engineers must further methodically care to keep all modifications genuinely conservative. To facilitate this,

Table 1. Suitability assessment of the investigated language composition techniques concerning different modeling scenarios (● = suitable, ⊙ = partially suitable, ○ = not suitable)

Scenario / Use Case	Inheritance	Extension	Embedding	Aggregation
(S1) Modifying a language, tailoring it to a specific use case	●	⊙	○	○
(S2) Extending a language to a use case while maintaining the integrity of the original models	⊙	●	○	○
(S3) Combining multiple language components into a modeling language	○	○	●	○
(S4) Combining modeling languages into a language family	○	○	●	●
(S5) Constructing huge languages with different constituents	○	○	●	●
(S6) Constructing a language or language family with heterogeneous parts for interdisciplinary use	○	○	⊙	●
(S7) Modularization of model artifacts	○	○	○	●

MontiCore provides a warning when this is not the case. As these scenarios only relate to a single DSL, embedding or aggregation are unsuitable.

For scenarios that involve employing multiple languages or language components, single inheritance (and extension, respectively) is not applicable. Evolving DSLs and tailoring them towards more sophisticated applications usually implies including various modeling techniques. Language embedding combines the constituents of multiple languages into a single one, connecting the different constructs. This is especially effective when the integrated DSLs share common interfaces, enabling the automatic embedding with nearly no glue code necessary. Thus, although requiring intricate knowledge of the involved components, embedding can still be facile when prepared well. For integrating (potentially incomplete) language components (S3), e.g., MontiThings comprising various literals, statements, and expressions, language embedding is the only applicable technique. As aggregation establishes a loose coupling only, this technique does not complete the components into a fully functional DSL. On the other hand, when integrating already functional languages into a family (S4), both embedding and aggregation might be applicable. The choice mainly depends on the respective modeling goal. An integrated view can be supportive in scenarios where the same domain experts create all aspects of models (e.g., in the context language of assistive systems). On the other hand, splitting dependent constructs of a large language (S5) into separate artifacts (which is automatically achieved by language aggregation) supports the organization and structuring of larger modeling projects.

While opting for language aggregation over embedding can positively impact structuring, this effect becomes even more apparent for interdisciplinary modeling teams working together on a product (S6). Here, the composed yet separated artifacts represent different domain-specific views of the system under development. This way, domain experts can contribute without getting distracted by the information of other modeling views. This observation results from both case studies as they employ class diagrams as separated artifacts for delivering type information. Finally, language aggregation for modularizing modeling artifacts (S7) is, even while not always necessary, a technique that language engineers should consider to foster a suitable modeling project structure and avoid model cluttering.

While all composition techniques are essential, the general impression is that the more sophisticated a language becomes, the more likely it is to apply a more elaborate approach. Therefore, minor DSL modifications usually employ inheritance or extension and keep the scope within a single domain or use case. Furthermore, reusing multiple concepts requires embedding constituents of different languages appropriately. Reasons such as structuring logical units of big models into artifacts or engineering whole language families incorporating multiple viewpoints of heterogeneous domains, both following the notion of separation of concerns, require language aggregation.

With this in mind, language aggregation is also the only composition technique natively supporting multi-viewpoint modeling. One of the main challenges in multi-viewpoint modeling is maintaining consistency between the individual views [11]. Aggregation automatically fulfills this requirement as models are organized in different artifacts. Thus, each artifact represents a separate, integrated view of the overall system automatically synchronized with other domain models' elements.

As usual for experience reports, our observations are subject to threats of validity, especially concerning generalizability. We have conducted our case studies in the technological space of MontiCore and are therefore tied to its capabilities and restrictions. However, we intentionally have chosen this ecosystem as it is specifically tailored for language composition, and the proposed composition techniques are state-of-the-art. Additionally, the presented approaches can conceptually, at least partially, be found in other frameworks as well, such as MPS or Xtext. This mitigates the threat to generalizability.

6 Conclusion

In this paper, we studied the composition of heterogeneous modeling languages which were originally developed independently but yet address different concerns in the same domain. While our investigation shows that the composition of such languages is both sensible and feasible, we also found that their composition requires different techniques whose application depends on a language's use case in the envisioned composition.

In total, we considered four composition techniques, namely inheritance, extension, embedding, and aggregation, and employed them to derive integrated, non-trivial language families for two distinct case studies concerning the engineering of cyber-physical systems for IoT and assistive systems. As a result, these language families are not only practically applicable for the integrated modeling of different viewpoints on systems of the mentioned kinds but also enabled us to assess the suitability of the aforementioned composition techniques. In this context, a major finding is that embedding and aggregation are indispensable for the composition of modeling language families, and even complement each other in a natural fashion. Finally, language aggregation automatically supports establishing different viewpoints on the model level for interdisciplinary modeling tasks, making it a considerable technique for realizing multi-viewpoint modeling scenarios.

Further evolvement of the language families, e.g., to include DSLs for describing requirements or goals, and the development of language families for other domains will provide additional examples to evaluate the composition techniques.

Acknowledgements

Funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2023 Internet of Production - 390621612. Website: https://www.iop.rwth-aachen.de.

References

[1] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2017. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics* 8, 1 (2017).

[2] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future (EMISA'19) (LNI, Vol. P-304)*. GI, 59–66.

[3] Fadi Al Machot, Heinrich C. Mayr, and Judith Michael. 2014. Behavior Modeling and Reasoning for Ambient Support: HCM-L Modeler. In *Int. Conf. on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA-AIE 2014) (LNAI)*.

[4] Moussa Amrani, Dominique Blouin, Robert Heinrich, Arend Rensink, Hans Vangheluwe, and Andreas Wortmann. 2021. Multi-paradigm modelling for cyber-physical systems: a descriptive framework. *Software and Systems Modeling* 20, 3 (2021), 611–639.

[5] Prashanti Angara, Miguel Jiménez, Kirti Agarwal, Harshit Jain, Roshni Jain, Ulrike Stege, Sudhakar Ganti, Hausi A. Müller, and Joanna W. Ng. 2017. Foodie Fooderson a Conversational Agent for the Smart Kitchen. In *27th Annual Int. Conf. on Computer Science and Software Engineering (CASCON '17)*. IBM, 247–253.

[6] Adil Anwar, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, Abdelaziz Kriouile, et al. 2010. A Rule-Driven Approach for composing Viewpoint-oriented Models. *Journal of Object Technology* 9, 2 (2010).

[7] Nelly Bencomo, Sebastian Götz, and Hui Song. 2019. Models@run.time: a guided tour of the state of the art and research challenges. *Software and Systems Modeling* 18, 5 (2019), 3049–3082.

[8] Lorenzo Bettini. 2016. *Implementing Domain-Specific Languages with Xtext and Xtend* (second ed.). Packt Publishing.

[9] Mathias Blumreiter, Joel Greenyer, Francisco Javier Chiyah Garcia, Verena Klös, Maike Schwammberger, Christoph Sommer, Andreas Vogelsang, and Andreas Wortmann. 2021. Towards Self-Explainable Cyber-Physical Systems. In *22nd Int. Conf. on Model Driven Engineering Languages and Systems (MODELS '19)*. IEEE Press, 543–548.

[10] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. 2022. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things* 3 (2022).

[11] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2019. A feature-based survey of model view approaches. *Software & Systems Modeling* 18 (2019), 1931–1952.

[12] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. 2020. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology* 19, 3 (October 2020), 3:1–16.

[13] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2021. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In *Composing Model-Based Analysis Tools*. Springer, 217–234.

[14] Arvid Butting, Judith Michael, and Bernhard Rumpe. 2022. Language Composition via Kind-Typed Symbol Tables. *Journal of Object Technology* 21 (October 2022), 4:1–13.

[15] Giuseppina Lucia Casalaro, Giulio Cattivera, Federico Ciccozzi, Ivano Malavolta, Andreas Wortmann, and Patrizio Pelliccione. 2022. Model-driven engineering for mobile robotic systems: a systematic mapping study. *Software and Systems Modeling* 21, 1 (2022), 19–49.

[16] Walter Cazzola and Edoardo Vacchi. 2013. Neverlang 2 – Componentised Language Development for the JVM. In *Software Composition*. Springer, 17–32.

[17] Federico Ciccozzi and Romina Spalazzese. 2017. MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In *Intelligent Distributed Computing X*. Springer, 67–76.

[18] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. 2016. *Engineering Modeling Languages*. Chapman & Hall.

[19] Manuela Dalibor, Nico Jansen, Bernhard Rumpe, Louis Wachtmeister, and Andreas Wortmann. 2019. Model-Driven Systems Engineering for Virtual Product Design. In *Proc. of MODELS 2019. WS MPM4CPS*. IEEE, 430–435.

[20] Istvan David, Kousar Aslam, Sogol Faridmoayer, Ivano Malavolta, Eugene Syriani, and Patricia Lago. 2021. Collaborative Model-Driven Software Engineering: A Systematic Update. In *ACM/IEEE 24th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*.

[21] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2015. Melange: A Meta-Language for Modular and Reusable Development of DSLs. In *Proc. of the 2015 Int. Conf. on Software Language Engineering (SLE 2015)*. ACM, 25–36.

[22] Elif Demirli and Bedir Tekinerdogan. 2011. Software Language Engineering of Architectural Viewpoints. In *Software Architecture*. Springer.

[23] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. 2010. Developing next Generation ADLs through MDE Techniques. In *32nd Int. Conf. on Software Engineering (ICSE '10)*. ACM, 85–94.

[24] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. 2019. SMArDT modeling for automotive software testing. *Journal on Software: Practice and Experience* 49, 2 (February 2019), 301–328.

[25] Imke Drave, Judith Michael, Erik Müller, Bernhard Rumpe, and Simon Varga. 2022. Model-Driven Engineering of Process-Aware Information Systems. *Springer Nature Computer Science Journal* 3 (November 2022).

[26] Florian Drux, Nico Jansen, and Bernhard Rumpe. 2022. A Catalog of Design Patterns for Compositional Language Engineering. *Journal of Object Technology* 21, 4 (October 2022), 4:1–13.

[27] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. 2017. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In *Software Architecture for Big Data and the Cloud*. Elsevier Science & Technology, Chapter 12.

[28] Johan Eker, Jorn W Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Sonia Sachs, Yuhong Xiong, and Stephen Neuendorffer. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. of the IEEE* 91, 1 (2003), 127–144.

[29] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, et al. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. SI on the 6th and 7th Int. Conf. on Software Language Engineering (SLE 2013, SLE 2014).

[30] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Int. Conf. Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10)*. ACM.

[31] J.-M. Favre. 2004. CaCOphoNy: metamodel-driven software architecture reconstruction. In *11th Working Conf. on Reverse Engineering*.

[32] Kevin Feichtinger, Kristof Meixner, Felix Rinker, István Koren, Holger Eichelberger, Tonja Heinemann, Jörg Holtmann, Marco Konersmann, Judith Michael, Eva-Maria Neumann, Jérôme Pfeiffer, Rick Rabiser, Matthias Riebisch, and Klaus Schmid. 2022. Industry Voices on Software Engineering Challenges in Cyber-Physical Production Systems Engineering. In *IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE.

[33] Robert France and Bernhard Rumpe. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering (FOSE '07)* (May 2007), 37–54.

[34] Ulrich Frank. 2002. Multi-perspective enterprise modeling (MEMO) conceptual framework and modeling languages. In *Proc. of the 35th Annual Hawaii Int. Conf. on System Sciences*. 1258–1267.

[35] Damian Frölich and L. Thomas van Binsbergen. 2022. ICoLa: A Compositional Meta-Language with Support for Incremental Language Development. In *Proc. of the 15th Int. Conf. on Software Language Engineering (SLE 2022)*. ACM, 202–215.

[36] Shan Fur, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2023. *Sustainable Digital Twin Engineering for the Internet of Production*. Springer Nature Singapore, 101–121.

[37] Arkadii Gerasimov, Judith Michael, Lukas Netz, and Bernhard Rumpe. 2021. Agile Generator-Based GUI Modeling for Information Systems. In *Modelling to Program (M2P)*. Springer, 113–126.

[38] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2020. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In *25th Americas Conf. on Information Systems (AMCIS) (AISeL)*. AIS.

[39] Javier González-Huerta, Emilio Insfran, and Silvia Abrahão. 2012. A Multimodel for Integrating Quality Assessment in Model-Driven Engineering. In *2012 8th Int. Conf. on the Quality of Information and Communications Technology*. 251–254.

[40] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. 2015. Engineering Tagging Languages for DSLs. In *Conf. on Model Driven Engineering Languages and Systems (MODELS'15)*. ACM/IEEE.

[41] Hans Grönniger and Bernhard Rumpe. 2011. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems (LNCS 6662)*. Springer, 17–32.

[42] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. 2015. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Vol. 580. Springer.

[43] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. 2012. *MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems*. Technical Report AIB-2012-03. RWTH Aachen University.

[44] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal* 37, 10 (2004).

[45] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. 2021. *MontiCore Language Workbench and Library Handbook: Edition 2021*. Shaker Verlag.

[46] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. 2019. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology* 18, 1 (2019).

[47] Eric Jouenne and Véronique Normand. 2005. Tailoring IEEE 1471 for MDE support. In *UML Modeling Languages and Applications: Satellite Activities*. Springer, 163–174.

[48] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. 2006. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In *Model Driven Engineering Languages and Systems*. Springer, 528–542.

[49] Jörg Christian Kirchhof, Anno Kleiss, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. 2022. Efficiently Engineering IoT Architecture Languages - An Experience Report (Poster). In *STAF 2022 Workshop Proc.: 2nd Int. Workshop on MDE for Smart IoT Systems (MeSS 2022)*, Vol. 3250. CEUR-WS.

[50] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. 2022. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Transactions on Internet of Things* 3 (2022), 1–30.

[51] Jörg Chrisitian Kirchhof, Lukas Malcher, Judith Michael, Bernhard Rumpe, and Andreas Wortmann. 2022. Web-Based Tracing for Model-Driven Applications. In *48th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*. IEEE, 374–381.

[52] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. 2021. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In *Proc. of the 20th Int. Conf. on Generative Programming (GPCE 21)*. ACM, 197–209.

[53] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. 2022. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software (JSS)* 183 (2022), 1–21.

[54] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*. Springer.

[55] Manuel Leduc, Thomas Degueule, and Benoit Combemale. 2018. Modular Language Composition for the Masses. In *Proc. of the 11th Int. Conf. on Software Language Engineering (SLE 2018)*. ACM, 47–59.

[56] Po-Sheng Li, Alan Liu, and Pei-Chuan Zhou. 2014. Context reasoning for smart homes using case-based reasoning. In *18th Int. Symp. on Consumer Electronics (ISCE'14)*. IEEE.

[57] Jörg Liebig, Rolf Daniel, and Sven Apel. 2013. Feature-Oriented Language Families: A Case Study. In *7th Int. WS on Variability Modelling of Software-Intensive Systems (VaMoS '13)*. ACM.

[58] Fadi Al Machot, Heinrich C. Mayr, and Suneth Ranasinghe. 2016. A windowing approach for activity recognition in sensor data streams. In *8th Int. Conf. on Ubiquitous and Future Networks, (ICUFN 2016)*. IEEE.

[59] Atif Mashkoor, Alexander Egyed, Robert Wille, and Sebastian Stock. 2022. Model-driven engineering of safety and security software systems: A systematic mapping study and future research directions. *Journal of Software: Evolution and Process* (2022), e2457.

[60] Apostolos Meliones and Stavros Maidonis. 2020. DALÍ: A Digital Assistant for the Elderly and Visually Impaired Using Alexa Speech Interaction and TV Display. In *13th ACM Int. Conf. on PErvasive Technologies Related to Assistive Env. (PETRA)*. ACM, Article 37, 9 pages.

[61] Judith Michael. 2022. A Vision Towards Generated Assistive Systems for Supporting Human Interactions in Production. In *Modellierung 2022 Satellite Events*. GI, 150–153.

[62] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2019. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In *Proceedings of MODELS 2019. Workshop MDE4IoT* (Munich), Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann (Eds.). CEUR Workshop Proceedings, 595–614.

[63] Judith Michael and Claudia Steinberger. 2017. Context Modeling for Active Assistance. In *ER Forum 2017 and ER 2017 Demo Track co-located with 36th Int. Conf. on Conceptual Modelling (ER 2017)*.

[64] Judith Michael, Claudia Steinberger, Vladimir A. Shekhovtsov, Fadi Al Machot, Suneth Ranasinghe, and Gert Morak. 2018. The HBMS Story - Past and Future of an Active Assistance Approach. *Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling* 13 (2018), 345–370. Issue Special Issue on Conceptual Modelling in Honour of Heinrich C. Mayr.

[65] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A. Fernandez, Bjørn Nordmoen, and Mathias Fritzsche. 2013. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling* 12, 3 (2013), 619–639.

[66] Mustafa Abshir Mohamed, Moharram Challenger, and Geylani Kardas. 2020. Applications of model-driven engineering in cyber-physical systems: A systematic mapping study. *Journal of Computer Languages* 59 (2020), 100972.

[67] Brice Morin, Nicolas Harrand, and Franck Fleurey. 2017. Model-Based Software Engineering to Tame the IoT Jungle. *IEEE Software* 34, 1 (January 2017), 30–36.

[68] David B Newell, Eite Tiesinga, et al. 2019. The international system of units (SI). *NIST Special Publication* 330 (2019), 1–138.

[69] OMG. 2013. *Business Process Model and Notation (BPMN), Version 2.0.2*. Technical Report. Object Management Group.

[70] Václav Pech. 2021. *JetBrains MPS: Why Modern Language Workbenches Matter*. Springer, 1–22.

[71] Carlos Peña and Jorge Villalobos. 2010. An MDE Approach to Design Enterprise Architecture Viewpoints. In *2010 12th Conf. on Commerce and Enterprise Computing*. IEEE, 80–87.

[72] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. 2018. Towards a UML Profile for Domain-Driven Design of Microservice Architectures. In *Software Engineering and Formal Methods*. Springer.

[73] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh, and Albert Zündorf. 2019. Viewpoint-Specific Model-Driven Microservice Development with Interlinked Modeling Languages. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 57–5709.

[74] Florian Rademacher, Jonas Sorgalla, Philip Wizenty, Sabine Sachweh, and Albert Zündorf. 2020. Graphical and Textual Model-Driven Microservice Development. In *Microservices: Science and Engineering*. Springer, 147–179.

[75] Alberto Rodrigues da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* 43 (2015), 139–155.

[76] Davide Di Ruscio, Ivano Malavolta, and Patrizio Pelliccione. 2014. A Family of Domain-Specific Languages for Specifying Civilian Missions of Multi-Robot Systems. In *Proc. of the First Int. Workshop on Model-Driven Robot Software Engineering (MORSE) Co-Located with STAF 2014*. CEUR-WS, 22–31.

[77] Stefan Rüther, Thomas Hermann, Maik Mracek, Stefan Kopp, and Jochen Steil. 2013. An Assistance System for Guiding Workers in Central Sterilization Supply Departments. In *6th Int. Conf. on PErvasive Technologies Related to Assistive Env. (PETRA '13)*. ACM.

[78] Johannes Sametinger. 1997. *Software engineering with reusable components*. Springer Science & Business Media.

[79] Jesús Sánchez Cuadrado. 2012. Towards a Family of Model Transformation Languages. In *Theory and Practice of Model Transformations*. Springer, 176–191.

[80] Thomas A. Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 494–497.

[81] Antero Taivalsaari and Tommi Mikkonen. 2017. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software* 34, 1 (Jan 2017), 72–80.

[82] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. 2010. A family of languages for architecture constraint specification. *Journal of Systems and Software* 83, 5 (2010), 815–831.

[83] Juha-Pekka Tolvanen and Steven Kelly. 2009. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proc. of the 24th Conf. Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, 819–820.

[84] Hans L Vangheluwe and G Vansteenkiste. 1996. A multi-paradigm modeling and simulation methodology. In *Simulation in Industry*.

[85] Markus Voelter. 2013. *Language and IDE Modularization and Composition with MPS*. Springer, 383–430.

[86] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen, and Krzysztof Czarnecki. 2013. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

[87] Marco Wagner, Dieter Zöbel, and Ansgar Meroth. 2013. Model-driven development of SOA-based driver assistance systems. *ACM SIGBED Review* 10, 1 (2013), 37–42.

[88] Niklaus Wirth. 1996. Extended Backus-Naur Form (EBNF). *ISO/IEC* 14977, 2996 (1996), 2–21.

[89] Enes Yigitbas, Ivan Jovanovikj, Kai Biermeier, Stefan Sauer, and Gregor Engels. 2020. Integrated model-driven development of self-adaptive user interfaces. *Software and Systems Modeling* 19, 5 (2020), 1057–1081.

[90] Steffen Zschaler, Pablo Sánchez, João Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. 2010. VML* – A Family of Languages for Variability Management in Software Product Lines. In *Software Language Engineering*. Springer.

[91] Doruk Şahinel, Cem Akpolat, O. Can Görür, Fikret Sivrikaya, and Sahin Albayrak. 2021. Human modeling and interaction in cyber-physical systems: A reference framework. *Journal of Manufacturing Systems* 59 (2021), 367–385.