Challenges for Model-Integrating Components

Mahdi Derakhshanmanesh
University of Koblenz-Landau
Institute for Software Technology
Koblenz, Germany
Email: manesh@uni-koblenz.de

Jürgen Ebert
University of Koblenz-Landau
Institute for Software Technology
Koblenz, Germany
Email: ebert@uni-koblenz.de

Marvin Grieger
University of Paderborn
Department of Computer Science
Paderborn, Germany
Email: marvin.grieger@uni-paderborn.de

***Abstract*—Model-Integrating Software Components (MoCos) use models at runtime as first class entities within components to build flexible and adaptive software systems. Building such systems requires to design and implement the required domain-specific modeling languages. Insufficient design and realization of modeling languages raises the risk that they may not be optimized for their later use. Although various works on the use of models at runtime exist, they do not address the engineering of modeling languages to be used in software components at runtime. In this paper, we introduce the idea of Comprehensive Language Models (CLMs) which explicitly considers modeling language engineering as a part of the development of component based software systems. This is achieved by extending the modeling language specification, e.g., by a set of interfaces for models which are used for accessing models at runtime. We illustrate an initial solution concept along an insurance sales app case study on Android based on which we derive a set of key challenges for the community.**

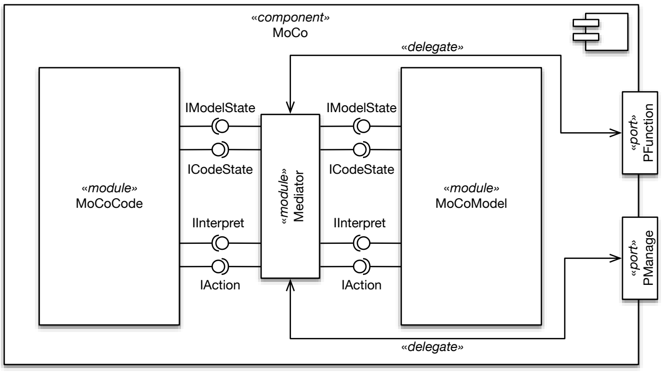


Fig. 1. Internal View on the MoCo Template (see [1])

I. Introduction

Models are no longer just used to design software but can become an integrated part of it, i.e., (executable) models and code coexist at runtime with equal rights. In previous works [1], [2], we proposed *Model-Integrating Software Components* (MoCos) as a concept for the design and development of such *model-integrating software systems*. Software engineers can choose to realize some parts of a system programmatically in code, while other parts are kept as models. No code is generated from the models but they are used at runtime. This concept yields flexible well-performing software that can be easily and systematically monitored, analyzed and modified.

The MoCo-approach combines models described using arbitrary *Domain-Specific Modeling Languages* (DSMLs) and embeds them within software components following a tailorable component design pattern that guides software engineers: the *MoCo Template*. It is depicted in Figure 1 and briefly introduced, next.

A. Model-Integrating Software Components

Each MoCo can have ports (`PFunction`, `PManage`) that are wired to the internal implementation, which is either encoded by a programming language (`MoCoCode`) or by a modeling language (`MoCoModel`). Conceptually, there may be sets of languages used on either side. In practice, a base technology such as the Java Virtual Machine (JVM) and its byte code format acts as a unifier. Programming languages are used on the code side, e.g., Java, and different – potentially integrated – DSMLs are used on the model side. Both constituents of a MoCo are encapsulated using *interfaces*. Optionally, a smart `Mediator` can manage any redirection from and to the MoCo's ports as well as communication between the model and code parts.

The *expected advantages* of the MoCo approach are: (i) *enhanced flexibility* because the system and its individual components can be observed using model queries, can be modified by adapting models using an editor or model transformations and can be executed using *model interpreters* [3], (ii) *support of separation of concerns* because each model targets a concern, (iii) *understandability and maintainability* because models are assumed to be easier to understand and easier to handle than code, (iv) *self-documentation* because a well designed modeling language is assumed to be a documentation and (v) *no synchronization problem* because there is no redundancy between model and code unless it is introduced willfully, e.g., to realize reflection.

B. Research Problem

An essential difference between component-based and model-integrating software is the use of various *models at runtime* (not just reflective *models@run.time* [4]). Therefore, developing model-integrating software systems includes choosing existing modeling languages or designing and implementing adequate DSMLs. In fact, it must be possible to introduce new DSMLs easily and quickly to realize certain parts of a system as a model. In turn, the use of models of a given DSML requires support for the following core activities: (i) *building models*, e.g., using an application programming interface or an editor, (ii) *binding models into components* as building blocks, e.g., following the MoCo Template and (iii) *using models*, e.g., querying, transforming and interpreting them. These activities can only be supported if there is a powerful *technological space* [5] for modeling languages and models, which provides all relevant capabilities needed. In the context of MoCos, such a technological space needs to work together with the respective *component execution platform* [6] or even be part of it.

There are examples for such technological spaces like the Eclipse Modeling Framework (EMF) [7] or JGraLab [8] that deliver acceptable support for language design, especially for syntax and constraints. Additionally, many research works use models at runtime in various ways and this specific topic is still a very relevant research area [9].

While different approaches and solutions for modeling and models at runtime already exist in isolation, we observe that they do not comprehensively address the required capabilities for designing new modeling languages that shall be an integrated part of a software system. Moreover, the *challenges* associated with designing DSMLs that support the *symbiosis* of models at runtime and code within software components have to be inspected.

This paper tackles the following *research problems* and their associated challenges:

(Q1) How to specify modeling languages comprehensively for generating adequate runtime support for them?

(Q2) What are challenges for modeling languages in the context of MoCos?

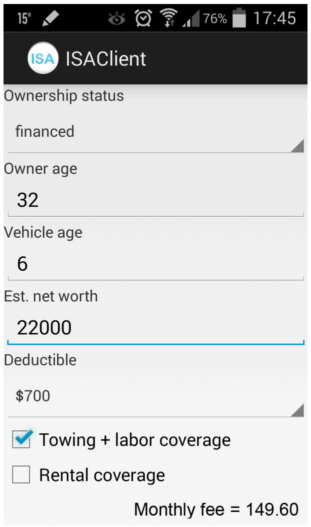
C. Contributions

In answering Q1, we propose that the introduction of a new DSML requires a *Comprehensive Language Model* (CLM), i.e., a specification of a modeling language to such an extent that all required activities on models are supported. We claim that each CLM should at least specify the following parts of DSML: (i) *syntax* (metamodel and constraints), (ii) *semantics* (dynamic state, constraints, state transitions) and (iii) *pragmatics* (at least facades [10]). We assume that a realization of a modeling language L_i will be derived from a CLM_i.

In answering Q2, we provide a description of selected *challenges* related to the seamless integration of models and code in software components, based on the *Insurance Sales App* (ISA) case study [1]. All in all, we aim to *raise awareness* for this topic and to initiate a fruitful discussion.

II. Running Example: Insurance Sales App

The *Insurance Sales App* (ISA) is a prototypical application for Google's Android mobile operating system. It has been developed to evaluate the feasibility of MoCos [1], [2]. It also serves as a *running example* throughout the rest of this paper. From the user's perspective, ISA's primary purpose is to support field staff in the insurance domain with their daily sales tasks. The system is built with MoCos, thus it is dynamically extensible at the component level and single components' internals can be also monitored and modified at runtime. For example, insurance fee formulas are adapted, based on the current physical location of a customer. A screenshot of ISA is shown in Figure 2 (left), illustrating one of the views of the Graphical User Interface (GUI) specific to the *car insurance product*.



15° 76% 17:45

ISAClient

Ownership status

financed

Owner age

32

Vehicle age

6

Est. net worth

22000

Deductible

$700

☑ Towing + labor coverage

☐ Rental coverage

Monthly fee = 149.60

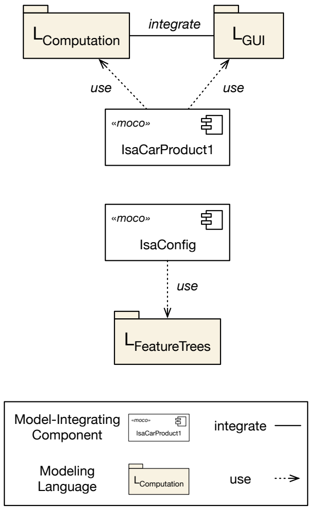


Fig. 2. MoCo-based ISA Client App and an Architecture Excerpt

ISA's architecture consists of a mix of pure Java libraries and MoCos, i.e., a certain part of the running software is encoded in models that are used at runtime, e.g., by querying and transforming them. An excerpt is depicted in Figure 2 (right). The specific modeling languages used represent (i) *feature trees* (L_f) for architectural reconfiguration, (ii) *computation* (L_c) for insurance fee formulas and (iii) *graphical user interfaces* (L_g) for data presentation and user input capturing.

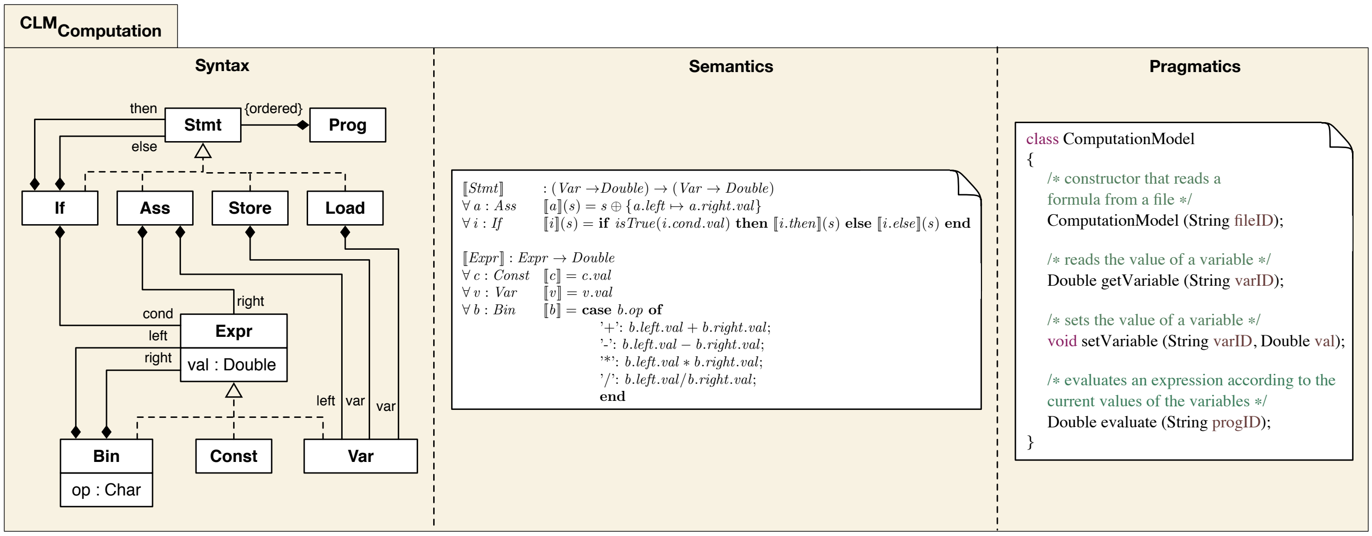
Regarding implementation, all MoCos conform to the structure proposed by the MoCo Template. For components, we used *OSGi's* [11] dynamic component technology, code was written in Java and models were developed in *JGraLab* [8]. The base execution platform is the Java Virtual Machine.

A. Comprehensive Language Model for L_c

As a clarification for what exactly a *Comprehensive Language Model* (CLM) is, we give an example in the context of the ISA case study. More concretely, we describe the CLM for L_c in the following and sketch how it relates to L_g. This background knowledge is required to understand some of the challenges described later in this paper.

CLM_c, i.e., the CLM that fully specifies the modeling language L_c, is depicted in Figure 3. It consists of three major parts: syntax, semantics and pragmatics.

1) Syntax Specification: L_c's *syntax* is specified using a UML-style metamodel and mostly represents the *static structure*. The language represents programs (`Prog`) consisting of statements (`Stmt`). There are special statements such as conditional (`If`), an assignment (`Ass`) and further specific



$$
\begin{array}{ll}
\llbracket Stmt \rrbracket & : (Var \rightarrow Double) \rightarrow (Var \rightarrow Double) \\
\forall\, a : Ass & \llbracket a \rrbracket(s) = s \oplus \{a.left \mapsto a.right.val\} \\
\forall\, i : If & \llbracket i \rrbracket(s) = \textbf{if } isTrue(i.cond.val) \textbf{ then } \llbracket i.then \rrbracket(s) \textbf{ else } \llbracket i.else \rrbracket(s) \textbf{ end}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\llbracket Expr \rrbracket : Expr \rightarrow Double} \\
\forall\, c : Const & \llbracket c \rrbracket = c.val \\
\forall\, v : Var & \llbracket v \rrbracket = v.val \\
\forall\, b : Bin & \llbracket b \rrbracket = \textbf{case } b.op \textbf{ of} \\
 & \quad \text{'+'}: b.left.val + b.right.val; \\
 & \quad \text{'-'}: b.left.val - b.right.val; \\
 & \quad \text{'*'}: b.left.val * b.right.val; \\
 & \quad \text{'/'}: b.left.val / b.right.val; \\
 & \quad \textbf{end}
\end{array}
$$

```
class ComputationModel
{
    /* constructor that reads a
    formula from a file */
    ComputationModel (String fileID);

    /* reads the value of a variable */
    Double getVariable (String varID);

    /* sets the value of a variable */
    void setVariable (String varID, Double val);

    /* evaluates an expression according to the
    current values of the variables */
    Double evaluate (String progID);
}
```

Fig. 3. CLM_c Comprising the Specification of Syntax (Dynamic Metamodel), Semantics (Dynamic Metamodel + State Transitions) and Pragmatics (Facade)

statements for loading (`Load`) and saving (`Store`) variables (`Var`). Variables and constants (`Const`) are expressions (`Expr`). Expressions have a value (`val`) as well as a left and right side of a specified binary operator (`Bin`).

2) Semantics Specification: L_c's *semantics* is specified (i) by extending the metamodel with information about the *dynamic state* and (ii) by adding a description of *state transitions* by following Plotkin's *Structured Operational Semantics* (SOS) approach [12]. This was an ad-hoc, pragmatic choice.

Like in *Dynamic Metamodeling* [13], L_c's metamodel elements depicted in Figure 3 also cover the *dynamic state* of its set of conforming models. The dynamic state is part of the semantics specification of L_c which is an essential part of any CLM. The attribute `val` belonging to the dynamic state can be changed during model execution.

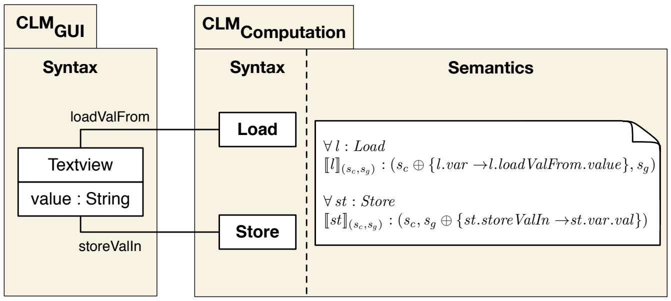
In terms of encoding the allowed *state transitions* in the dynamic state, we chose Plotkin's approach as a technology-independent precise and comprehensible formalism. For example, as shown in Figure 3, the semantics of a `Stmt` in L_c is that a given variable's value is replaced with another (potentially the same) value.

3) Pragmatics Specification: L_c's *pragmatics* is specified using a facade, e.g., using a notation similar to Java classes as illustrated in Figure 3. This approach facilitates the use of models similar to code objects. Moreover, the specification of available *services on models* of a given language, here L_c, supports communication between modeling language designers, software architects and software engineers. We use the term *services on models* to denote capabilities and functionalities specific to a modeling language that facilitate the use of models. These services are realized as *facades*.

For example, the `ComputationModel` facade allows to load a model from a file, to get and set a value for a variable and, importantly, to evaluate a model (e.g., representing an insurance fee formula in ISA) by starting model execution at a certain `Prog` element.

B. Integration of L_c and L_g

Besides the use of single modeling languages, it is particularly interesting when multiple languages are used together.[1] In the context of ISA, each insurance product MoCo carries the insurance fee formula (an instance of L_c) and its corresponding representation of the graphical user interface (an instance of L_g). The two modeling languages had to be *integrated*. For this purpose, additional associations (`loadValFrom` and `storeValIn` roles) were introduced and the specification of state transitions in CLM_c was extended to encode the semantics of loading values from the GUI (`Load`) and storing values from a formula in the GUI's `TextView` (`Load`). In Figure 4, the corresponding integration via additive extension of two CLMs is given.



$$
\begin{array}{l}
\forall\, l : Load \\
\llbracket l \rrbracket_{(s_c, s_g)} : (s_c \oplus \{l.var \rightarrow l.loadValFrom.value\}, s_g) \\
\\
\forall\, st : Store \\
\llbracket st \rrbracket_{(s_c, s_g)} : (s_c, s_g \oplus \{st.storeValIn \rightarrow st.var.val\})
\end{array}
$$

Fig. 4. Integration of two Comprehensive Language Models

CLMs support the specification and design of modeling languages to be used within model-integrating software components as described. However, there are still many open challenges that need to be tackled.

[1] The *GEMOC initiative* (http://gemoc.org/) provides related work on the coordinated use of heterogeneous modeling languages.

TABLE I
INITIAL LIST OF CHALLENGES FOR MODELING LANGUAGES IN MOCOS

ID	Challenge Description
Syntax	
C1	How to modularize metamodels?
C2	How to integrate two metamodels?
C3	How to establish links between different models?
C4	How to provide a context-dependent view on sets of models?
Semantics	
C5	How to specify model semantics?
C6	How to realize model execution?
C6.1	How to manage the dynamic state of a model?
C6.2	How to reuse model interpreters?
C6.3	How to support the interplay of different model interpreters?
C7	How to support variants of semantics for the same modeling language?
Pragmatics	
C8	How to establish control and data flow between models and code?
C9	How to design language-specific and usage-specific services on models?
C10	How to control access to models?

III. CHALLENGES

Building on the use of CLMs, we elaborate on an initial list of *challenges* for modeling languages in the context of MoCos using the ISA running example. For readability, we formulate each challenge as a question and cluster them according to (i) *syntax*, (ii) *semantics* and (iii) *pragmatics* as summarized in Table I. A detailed description follows subsequently.

A. Syntax Challenges

1) Modularization of Metamodels: Software architects follow a divide-and-conquer approach and split larger systems in smaller pieces, e.g., into software components and connectors. A standardized approach for the modularization of modeling languages is missing, though. A main part of any DSML's definition is the specification of its syntax with a metamodel. While package-structures and import mechanisms are available, depending on the concrete modeling technology, the modeling language designer cannot orchestrate modeling languages and their metamodels in a black-box fashion (C1). In ISA for example, feature models (L_f) are managed and used by a MoCo called `IsaConfig` and insurance fees models (L_c) as well as GUI models (L_g) are managed and used by insurance products like the `IsaCarProduct1` MoCo.

2) Integration of Metamodels: Integration means that at least two existing modeling languages shall be merged. In contrast to distributed, potentially not connected models conforming to different metamodels, this approach conveniently enables full access, e.g., via model queries, to the conforming models of this integrated modeling language (C2). In ISA for example, insurance fee models and GUI models are used tightly together within the `IsaCarProduct1` MoCo, e.g., values from computed fees are directly associated with elements of the user interface.

3) Links between Different Models: In a MoCo-based software system, the architectural decomposition based on functionality dictates a clean separation of concerns between the various MoCos. As in any other component-based software systems, MoCos are connected with each other via provided and required interfaces. While separation of concerns has many well-known advantages, it has one major disadvantage in the context of MoCos: the flexibility that comes with the ability to query and transform a single (possibly large) interconnected model can no longer be leveraged if single models are distributed and encapsulated across individual MoCos (C3). There are no associations between them on the model-level. The ability to establish links and to access these distributed models is especially helpful to debug MoCos and their interdependencies. In ISA for example, it is interesting to know the available insurance fee formulas (contained in `IsaCarProduct1`) for a specific feature configuration (contained in `IsaConfig`).

4) Context-Dependent Views on Models: Modeling languages and their metamodels are only partly used, i.e., the available expressiveness is not required and a restricted metamodel will be sufficient. Moreover, only some data may be relevant for a certain use case and MoCo. The ability to define an adequate and context-specific *view* on sets of models (C4) helps to reduce complexity and supports ease of use of DSMLs. In fact, a view can be seen as a specification for the model parts that can be used by another MoCo. In ISA for example, an adaptation manager MoCo may need to control certain location-dependent variables of the insurance fees and their associated GUI elements. These parts could be encoded by a special adaptation view.

B. Semantics Challenges

1) Specification of Model Semantics: The models in MoCos are not only used as pure data (similar to databases) but some are also executed. Therefore, the semantics of modeling languages needs to be precisely specified (C5). There is a multitude of options available and one can choose between a spectrum of rather informal and very formal approaches [14]. It is important to choose a formalism that is both sufficiently formal but also adequately practical for software engineers and modeling language designers. In ISA for example, all three modeling languages are executable: L_f is interpreted for architectural reconfiguration, L_c is interpreted to compute an insurance fee and L_g is interpreted to create and synchronize an Android-specific graphical user interface for each insurance product represented by a single MoCo.

2) Realization of Model Execution: Given a specification of semantics for a modeling language, this specification needs to be implemented (C6) and related decisions need to be taken carefully. In general, there has been no commonly accepted proposal for the realization of model semantics/model interpreters, yet. Besides the development of stand-alone interpreters and model interpreters embedded into the metamodel, code generation is another option. Each approach has its advantages and disadvantages, e.g., with regards to performance, complexity and reusability. In ISA for example, there is at least one model interpreter for each modeling language.

One sub-challenge that is critical in the case of interpretation is the way to deal with the parts of a model that may change during interpretation (C6.1). We refer to them as the model's *dynamic state*, in contrast to the rest of the model, the *static structure*. While these parts can be regarded as the execution context of a stand-alone model interpreter, it can be also seen as a part of the actual model. In ISA for example, models of the kind of L_f, i.e., feature configurations, are used for runtime reconfiguration. This implies that the state of a feature (selected or not selected) varies. This information can be stored separately by the model interpreter (e.g., technically as a hashmap) or it can be a Boolean attribute in the L_f metamodel.

A second sub-challenge is related to the reuse of model interpreters (C6.2). In this specific case, one needs to distinguish between (i) reuse of model interpreter implementations and (ii) their instances at runtime. Depending on the chosen type of implementation, the reuse potential varies. In ISA for example, the same feature model can be interpreted by different model interpreters concurrently if the dynamic state, i.e., the feature selection flag, is stored by the model interpreters themselves. On the contrary, if the dynamic state is part of each model but needs to be different per semantics, then the model needs to be duplicated or an embedded model interpreter needs to instantiate the dynamic state multiple times.

A third sub-challenge is related to the interdependencies of semantics and, hence, the interplay of model interpreters (C6.3). Ideally, each modeling language comes with its own set of model interpreters. In case that two modeling languages need to be used together, it is required that not only their metamodels are integrated (see C2) but it is also necessary that their semantics fit. In the simplest form, one model interpreter invokes another model interpreter, which asks for a more sophisticated management of dependencies – especially if these shall be dynamic. In ISA for example, each insurance product MoCo encapsulates an insurance fee formula model and a GUI model. Given that their metamodels were previously integrated, the model interpreter of L_c (i) may access metaclasses of L_g and operate on them (e.g., store a computed insurance fee in a text field), or (ii) may invoke the model interpreter of L_g to perform the task.

3) Variants of Semantics: We experienced that while for some modeling language (especially general-purpose modeling languages) a single semantics specification is sufficient, in the case of DSMLs, multiple semantics for the same modeling language need to be supported (C7). Therefore, in this context, a modeling language becomes a *software product line*. Reuse is critical to deal with complexity and to avoid redundancy and duplication. In ISA for example, there may be behavioral semantics (dynamic reconfiguration), constraint checking semantics and visualization semantics for L_f.

C. Pragmatics Challenges

1) Data and Control Flow between Model and Code: In MoCos, models and code coexist and realize the functionality of the component together. It is important to be able to invoke models from code and vice versa (C8). The MoCo Template already defines a pattern with its `Mediator` and sketched interfaces. We deem it important to further standardize these interfaces and to provide realization guidelines, e.g., in the scope of our *reference implementation* (MoCo API) [1]. The design and development of *language-specific facades* encapsulating required services as a part of a CLM needs to be researched. In ISA for example, there is code for sending an email report in the `MoCoCode` module of the `IsaCarProduct1` MoCo that receives data from the `MoCoModel` module (insurance fee model, GUI model).

2) Services on Models: When talking about services on models, two categories need to be distinguished: (i) *foreseen services* that are provided by the modeling language designer and (ii) *unforeseen services* that are specific to a certain user of a modeling language (e.g., a system or a component). Moreover, modeling languages – especially DSMLs as primarily used in MoCos – need to be compact and adequately expressive. A strategy and a set of mechanisms is required to specify *context-specific services*, realize them and to manage the resulting variability (C9). In ISA for example, different insurance products, i.e., different MoCos, require similar special services, like email reporting. This particular service was not initially foreseen when developing L_c and L_g. Therefore, it was first developed as a part of the respective `MoCoCode` module, resulting in clones across the different insurance product MoCos. To solve such issues, often required and DSML-specific services need to be offered by a facade as a part of the CLM.

3) Access Control for Models: Given the power of models in the MoCo concept, any kind of analysis or manipulation needs to be controlled (C10). Models need to be accessible only in predefined authorized, i.e., safe and secure, ways. It needs to be decided whether models can be accessed using the MoCos' interfaces only or if there is a more powerful role that can inspect everything, i.e., all models within all MoCos across the system architecture. Indeed, there is a tradeoff between flexibility and encapsulation. In ISA for example, obviously insurance fee formulas should not be editable by anyone, even though this is technically possible at any time using model transformations. An adaptation manager MoCo that adjusts the fee formulas according to the geolocation of the sales person and potential customer requires exactly this ability, though.

IV. Related Work

While there are works that deal with individual challenges described in this paper, we observe that a *comprehensive solution approach* is missing. The work on Model-Integrated Computing, initiated by *Sztipanovits and Karsai* [15], targets similar issues but lacks the runtime aspect. Due to limited space, we can only hint at an excerpt of related work here.

Regarding the topic of *syntax*, *Heidenreich et al.* [16] present a generic approach for the composition of models that is based on invasive software composition and the Reuseware Composition Framework. *Krahn et al.* [17] use an extended grammar format that supports language inheritance and embedding for the modular development of textual domain-specific languages. *Bae et al.* [18] propose to modularize a large metamodel into a set of small metamodels and present their idea of model slicing along the UML. In contrast to modularization approaches, *Atkinson et al.* propose a Single-Underlying-Model (SUM) [19] that serves all users.

Regarding the topic of *semantics*, *Plotkin* [12] proposes a structural approach to operational semantics. *Engels et al.* [13] describe dynamic metamodeling as a graph-based approach to the specification of semantics for (behavioral) modeling languages. The *Object Management Group* (OMG) [20] provides a specification of the semantics of a foundational subset for executable UML models (fUML) using activity diagrams and a dedicated action language. *Mayerhofer* [14] comprehensively describes the state of the art in model execution.

Regarding the topic of *pragmatics*, *Balz et al.* [21] discuss the embedding of behavioral models (state machines) into object-oriented source code. *Ecore Facade* [22] is a textual domain-specific language for annotating existing Ecore metamodels. This mechanism can be used to define multiple views for a single metamodel via Ecore facade models. The survey by *Szvetit and Zdun* [23] covers existing research on models at runtime and software architecture in detail.

V. Concluding Remarks

In this paper, we introduced *comprehensive language models* as a way to specify modeling languages in the context of *model-integrating software components*. Moreover, we gave concrete examples along the insurance sales app study and elaborated on a first set of *challenges*.

We conclude that an infrastructure is needed that provides all model-specific services in a light-weight, homogeneous, formally founded, easily understandable, and efficient manner using a comprehensive *technological modeling space* supplying full modeling and metamodeling support, and coherent interoperable services based on a powerful data structure.

Regarding future work, we plan to carry out additional case studies to identify further challenges for the infrastructure needed to support model-integrating software components. In the long run, we aim to manifest our lessons learned in a systematically derived engineering method [24].

Acknowledgment

This work is supported by the Deutsche Forschungsgemeinschaft (DFG) under grants EB 119/11-1 and EN 184/6-1. The authors would like to thank Gregor Engels for his valuable feedback and Thomas Iguchi for his implementation support.

References

[1] M. Derakhshanmanesh, J. Ebert, T. Iguchi, and G. Engels, “Model-Integrating Software Components,” in *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, ser. Lecture Notes in Computer Science, J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfrán, Eds., vol. 8767. Springer, 2014, pp. 386–402.

[2] M. Derakhshanmanesh, *Model-Integrating Software Components - Engineering Flexible Software Systems*. Springer, 2015.

[3] M. Derakhshanmanesh, M. Amoui, G. O’Grady, J. Ebert, and L. Tahvildari, “GRAF: Graph-based Runtime Adaptation Framework,” in *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS ’11*. New York, NY, USA: ACM Press, May 2011, pp. 128–137.

[4] G. Blair, N. Bencomo, and R. B. France, “Models@run.time,” *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[5] I. Kurtev, J. Bézivin, and M. Aksit, “Technological Spaces: An Initial Appraisal,” in *International Symposium on Distributed Objects and Applications, DOA 2002*, 2002.

[6] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, “A Classification Framework for Software Component Models,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011.

[7] “Eclipse Modeling Framework Hompage,” https://eclipse.org/modeling/emf/ (accessed July 16th, 2015).

[8] “JGraLab Hompage,” http://jgralab.uni-koblenz.de (accessed July 15th, 2015).

[9] S. Götz, N. Bencomo, and R. France, “Devising the Future of the Models@Run.Time Workshop,” *SIGSOFT Softw. Eng. Notes*, vol. 40, no. 1, pp. 26–29, Feb. 2015.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[11] The OSGi Alliance, “OSGi Core Release 5,” The OSGi Alliance, Tech. Rep. March, 2012, http://www.osgi.org/Download/File?url=/download/r5/osgi.core-5.0.0.pdf (accessed July 15th, 2015).

[12] G. D. Plotkin, “A Structural Approach to Operational Semantics,” 1981. [Online]. Available: http://homepages.inf.ed.ac.uk/gdp/publications/

[13] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer, “Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML,” in *Proceedings of the 3rd international conference on the Unified Modeling Language (UML 2000), York (UK)*, ser. LNCS, B. S. A. Evans, S. Kent, Ed., vol. 1939. Berlin/Heidelberg: Springer, 2000, pp. 323–337, third International Conference.

[14] T. Mayerhofer, “Defining Executable Modeling Languages with fUML,” Ph.D. dissertation, Institute of Software Technology and Interactive Systems, 2014.

[15] J. Sztipanovits and G. Karsai, “Model-Integrated Computing,” *Computer*, vol. 30, no. 4, pp. 110–111, Apr. 1997.

[16] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler, “On Language-Independent Model Modularisation,” in *Transactions on Aspect-Oriented Software Development VI*, ser. Lecture Notes in Computer Science, S. Katz, H. Ossher, R. France, and J.-M. Jézéquel, Eds. Springer Berlin Heidelberg, 2009, vol. 5560, pp. 39–82.

[17] H. Krahn, B. Rumpe, and S. Völkel, “MontiCore: Modular Development of Textual Domain Specific Languages,” in *Objects, Components, Models and Patterns*, ser. Lecture Notes in Business Information Processing, R. Paige and B. Meyer, Eds. Springer Berlin Heidelberg, 2008, vol. 11, pp. 297–315.

[18] J. H. Bae, K. Lee, and H. S. Chae, “Modularization of the UML Metamodel Using Model Slicing,” in *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, April 2008, pp. 1253–1254.

[19] C. Atkinson, R. Gerbig, and C. Tunjic, “A Multi-level Modeling Environment for SUM-based Software Engineering,” in *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, ser. VAO ’13. New York, NY, USA: ACM, 2013, pp. 2:1—-2:9.

[20] The Object Management Group, “Semantics of a Foundational Subset for Executable UML Models (fUML),” p. 441, 2012. [Online]. Available: http://www.omg.org/spec/FUML/

[21] M. Balz, M. Striewe, and M. Goedicke, “Embedding Behavioral Models into Object-Oriented Source Code,” *Proceedings of ”Software Engineering 2009”*, 2009.

[22] “Ecore Facade,” 2015. [Online]. Available: http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Ecore_Facade

[23] M. Szvetits and U. Zdun, “Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime,” *Software & Systems Modeling*, pp. 1–39, 2013.

[24] G. Engels and S. Sauer, “A Meta-Method for Defining Software Engineering Methods,” in *Graph Transformations and Model-Driven Engineering*, ser. Lecture Notes in Computer Science, G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, Eds. Springer Berlin Heidelberg, 2010, vol. 5765, pp. 411–440.