Domain Globalization: Using Languages to Support Technical and Social Coordination

Julien Deantoni[1], Cédric Brun[2], Benoit Caillaud[3], Robert B. France[4], Gabor Karsai[5], Oscar Nierstrasz[6], and Eugene Syriani[7]

[1] University of Nice-Sophia-Antipolis, France
[2] Obeo, France
[3] INRIA, Rennes, France
[4] University of Colorado, CO, U.S.A.
[5] Vanderbilt University, TN, U.S.A.
[6] University of Bern, Switzerland
[7] University of Montreal, Canada

Abstract. When a project is realized in a globalized environment, multiple stakeholders from different organizations work on the same system. Depending on the stakeholders and their organizations, various (possibly overlapping) concerns are raised in the development of the system. In this context a Domain Specific Language (DSL) supports the work of a group of stakeholders who are responsible for addressing a specific set of concerns. This chapter identifies the open challenges arising from the coordination of globalized domain-specific languages. We identify two types of coordination: technical coordination and social coordination. After presenting an overview of the current state of the art, we discuss first the open challenges arising from the composition of multiple DSLs, and then the open challenges associated to the collaboration in a globalized environment.

Keywords: Composition, Coordination, DSL, Globalization

1 Context

In this chapter we describe the issues associated with the coordination aspects of the globalizing languages challenge. Specifically, the focus is on how globalized domain-specific languages (DSLs) can be used on projects with multiple stakeholder groups, each focusing on a different engineering/development concern, to support analysis of system properties and coordination of work across the groups. The groups may span multiple organizations that temporarily need to collaborate on a project, thus one needs to accommodate different collaboration styles and engineering cultures, and manage differing trust and security procedures when it comes to sharing information. The language-based coordination mechanisms should take these into consideration.

In this context, a DSL is a software or system language that is specifically built to support the work of a group of stakeholders that are responsible for addressing a specific set of concerns. It must therefore be supported by technologies that serve the particular purposes of the stakeholders. For example, a DSL's purpose may be to support static analysis of properties, provide a description of a system component, or it may be used to support simulation of some behavioral aspect. Based on purpose, a DSL may be declarative, executable, prescriptive or descriptive.

A distinguishing characteristic of a globalized framework of DSLs is its openness, that is, there is no restriction on the form of languages and supporting tools that can be added to the framework. Realizing the globalized DSL vision thus requires consideration of how new DSLs and their toolsets are incorporated into the language framework.

Two types of language-based coordination can be broadly identified: Technical and Social. Technical coordination is concerned with the mechanisms used to compose heterogeneous languages to support analysis of properties that require information captured in models expressed in different languages. Such analysis typically requires coordinated analysis of the models expressed in the different languages. Examples of two forms of analysis are consistency checking and compatibility checking. Checking consistency means to determine whether information spread across models expressed in different languages is not contradictory. This is easily understood when considering DSLs expressing logical or numerical constraints where checking consistency amounts to checking the satisfiability of the conjunction of the constraints. In system engineering, consistency usually applies to models belonging to the various viewpoints of the same component. Compatibility checking is concerned with determining whether two models can be composed in a particular environment, that is, two models are compatible if there is an environment in which two models can work together. The simplest instance of this concept is the type compatibility of components with input and output ports, where the output of one component should be a subtype of the input of the other components. Compatibility usually applies to models of several interacting components that form a system architecture.

To support technical coordination, correspondences between language elements should be defined at the syntactic level. Elements for identifying and describing such correspondences needs to be supported.

Language-based social coordination is concerned with how globalized DSLs can be used to support effective collaboration across stakeholder groups. Coordination of work through globalized software languages leads to social translucence, where relatively autonomous groups of stakeholders are made aware of activities performed by other groups. Groups can thus react accordingly to communicated information and in turn notify other groups of their reactions. These interactions can take place through the sending of notifications and feedback. Social coordination can also include support for managing resources across groups. To support social coordination the DSLs may have, for example, to be augmented with metadata about activities and resources associated with the DSL language elements.

2 State of Art

Multiple DSLs can be composed into a host environment at a variety of different levels. The coarsest level is that of *tool composition*, where tools supporting a domain modeling approach may be composed, but there is no language composition per se. With *model composition*, the underlying models can interact, but again there is no language composition. Finally, with *language composition*, individual DSLs may be integrated or coordinated, either at a syntactic level or at a deeper semantic level.

2.1 Tool composition frameworks

Tool composition frameworks provide a means for individual tools to interact with one another. "Tool" here can mean a model editing or model execution environment, but in practice they may include simpler tools like spreadsheets. The tools may be domain-specific or general-purpose, and tool composition frameworks facilitate tool interoperability. Several tool integration patterns have been developed and used in complex toolchains [10, 29, 31, 5]. Tool chains may support collaborative work, either directly (when a multitude of developers is assisted by the framework, in real-time, synchronously), or indirectly (when the collaboration is more asynchronous).

Some tool integration frameworks are distributed (with tools running on different platforms), some are desktop-based, where the framework provides a unified visual interface to a suite of the integrated tools. For the latter **Eclipse**[8] is the most prevalent example. Commercial products are also available.[9].

There also exist tool coordination frameworks whose goal is to orchestrate the execution/simulation driven by different tools so that data can be exchanged between them during the simulation. One of the best-known coordination framework is the the functional mock-up interface [7].

2.2 Model composition frameworks

The purpose of model composition is to provide a consistent view of various models possibly expressed in different modeling languages for the purposes of analysis and synthesis. The challenge is that the domain, the syntax, and the semantics of modeling languages can be widely different, yet the composed model has to have a semantics on its own that is relevant for the task at hand. Broadly speaking, models can be composed either via a hierarchy or via side-by-side composition. In hierarchical composition models coming from one language are embedded in models expressed in another language, while in side-by-side composition models at the same level of abstraction are related, usually via their interfaces. Models can be static artifacts (*i.e.,* passive documents) or dynamic entities (*e.g.,* models embedded within a simulation engine). Hence, model integration can be static or

[8] http://en.wikipedia.org/wiki/Eclipse_Modeling_Framework
[9] http://www.phoenix-int.com/software/phx-modelcenter.php

dynamic, yielding either a composed (static) document or an active, integrated dynamic model executing on some platform. In all model integration frameworks, there is some common foundation to support integration. This could be syntactic, semantic, operational, or some mixture. By syntactic foundation we mean a concrete textual or visual notation that allows model integration; by semantic foundation we mean a common semantic domain, and by operational foundation we mean some software infrastructure that allows the inter-operation of models.

UML Profiles provide a mechanism for defining and composing domain-specific modeling languages in the overall UML framework. These are not new languages, but already defined UML constructs that are specialized through stereotyping. Stereotypes are special markers applied to specific UML model elements, which gain a specific semantics through this process. Profiles often specify model patterns (built from stereotyped model elements) that have domain-specific semantics. In this case the model integration platform is UML, and the integration is supported by the model integration operators of UML.

Coordination languages encompass both the formalisms and the mechanisms needed to achieve multiple parallel, possibly distributed computation. Their purpose is to coordinate a number of possibly heterogeneous executable models together, by interfacing with each of them in such a way that they can take advantage of parallel and distributed systems [25]. Examples of such languages are Linda [12] for data-driven coordination, or Esper[10] for event-driven coordination. These languages emphasize the benefits of having an explicit coordination model to reason about the coordination of multiple executable models.

The **CyPhy**[11] [47] modeling language was developed to facilitate model coordination for the design of complex cyber-physical systems, for instance vehicles. This model coordination language is built around a hierarchical component model where components have four categories of interfaces: parametric and properties (for setting and getting parametric values), signal interfaces (for causal interactions among electronic and software components), power interfaces (for acausal interactions among physical components with dynamics), and structural interfaces (for geometric 3D composition). CyPhy components contain references for high-fidelity domain-specific models stored in external modeling tools and model databases. CyPhy models are composed by connecting the strongly-typed component interfaces so that the composite allows a coordinated analysis of the entire system. Note that the analysis can cut across many different domain-specific models.

The **High-Level Architecture**[12] (HLA) (IEEE-1516) [34] is a run-time framework for coordinating heterogeneous distributed simulation engines, called federates. It provides a standard for facilitating interaction among simulations, including message formats and the synchronization of the logical clocks of the simulators. Each simulation preserves its own semantics for the model, but as simulations advance in time, their clocks remain synchronized. The semantics of

[10] http://www.espertech.com/esper/
[11] http://cps-vo.org/group/avm/meta-overview
[12] http://en.wikipedia.org/wiki/High-level_architecture_(simulation)

the federation (composed from the federates) is that of a large-scale dynamic system where each component has its own dynamics, yet the temporal progression of the individual engines is carefully regulated.

2.3 Language composition frameworks

Language integration frameworks enable the embedding of multiple DSLs into a host language. This integration is commonly at a *syntactic* level. The composition may also be done at a deeper *semantic* level either by integration or by coordination.

Syntactic integration Syntactic integration of domain-specific languages is commonly supported by so-called *language workbenches* [22], environments that define (1) a schema for an abstract syntax for a language (*i.e.,* a grammar), (2) one or more rich editing environments for the language, and (3) language semantics, typically either by direct interpretation or code generation. Language workbenches can be based on a variety of parsing technologies, such as generalized LR (GLR) parsing [48], generalized LL (GLL) parsing [43], term rewriting [23], parser combinators [24], or parsing expression grammars [9].

AToMPM[13] [46] is a framework for generating syntax-directed domain-specific modeling editors, performing in-place model transformation, and managing DSMs in a cloud-based web environment. Each DSL has one meta-model. However, a model can be built that links instances from different meta-models, therefore a model can conform to multiple meta-models. A DSL can be assigned multiple graphical concrete syntaxes.

AToMPM follows a view-based modeling approach. A user only interacts with a view of a model, specified in a dedicated concrete syntax, showing a sub-set of the underlying model. Changes in one view are automatically propagated to the model and to other overlapping views. Multiple users can collaboratively work on the same view. Concurrent conflicting changes are handled by manual inspection through a chatting system.

Helvetia[14] [40] is a PEG-based language workbench for adding DSLs to Smalltalk by source code transformation. The transformations are available to the entire language toolchain, so tools like the editor and the debugger can exploit them to accurately display the original embedded DSL source rather than just the generated host language code.

MetaEdit+[15] is a mature language workbench that supports graphical diagram, matrix and table representations for DSLs. Languages can be composed by integrating individual metamodels or by creating a metamodel that includes several integrated languages. A language definition is integrated combining abstract syntax, static semantics, concrete syntax and transformations. MetaEdit+ supports collaborative language engineering allowing several persons to create

[13] http://www-ens.iro.umontreal.ca/ syriani/atompm/atompm.htm
[14] http://scg.unibe.ch/research/helvetia
[15] http://www.metacase.com/

DSLs at the same time as well as it supports collaborative modeling when using the DSLs. It is a commercial and supported environment that is used to create hundreds of DSLs.

TXL[16] [15] is a source code transformation language based on term rewriting. TXL can be used to transform embedded DSLs to a host language, much like Helvetia. **Spoofax**[17] [37] is a language workbench based on term rewriting. Spoofax offers a fine degree of control over the term rewriting traversal strategy.

MPS[18] is a platform enabling the definition and integration of DSLs through the use of language extensions and projectional editing. Rather than manipulating a program as a text, MPS stores a program as an abstract syntax tree (AST) and edits it directly. MPS enables embedding of a language into another while avoiding the problem of textual grammar ambiguity by not storing language code as text at all but storing the AST and reifying the notion of *Language Extension*: a set of concepts that refine those present in the base model with their own attributes and references.

The **Generic Modeling Environment** (**GME**) [36][19] is a metaprogrammable graphical modeling environment that enables the definition of graphical modeling languages through metamodels. Once defined, the metamodel can immediately be used in a domain-specific graphical model editor that enforces the use of concepts, integration operators, and well-formedness rules (*i.e.,* the structural semantics) of the domain-specific modeling language defined by the metamodel. The tool has its own meta-metamodel, and provides model access API-s, both on the meta- (*i.e.,* language-) level and the domain (*i.e.,* model-) level. Metamodels of languages are composable within the tool, allowing the integration of domain-specific modeling languages. The most recent development in GME (called WebGME[20]) provides a web-based collaborative graphical modeling environment with version control and support for distributed model editing.

Dictō [11] follows a lightweight approach to integrating architectural constraint checkers. Rather than integrating multiple DSLs, it offers a single, syntactical framework for expressing different kinds of architectural constraints, and allows tools for checking those constraints to be plugged into the Dictō framework.

Semantic composition. When the system consists of different executable languages, it is of primary importance to understand what are the emerging behaviors (whether expected or not) of the global system. In such cases it is necessary to understand how the semantics of each language can be composed. The goal of the semantic composition is to enable simulation and/or verification activities on the global system according to (1) the semantics of each language and (2) the behavior scattered in/specified by the heterogeneous models (*i.e.,* the

[16] http://www.txl.ca/
[17] http://strategoxt.org/Spoofax/WebHome
[18] http://www.jetbrains.com/mps/
[19] http://www.isis.vanderbilt.edu/Projects/gme/
[20] http://webgme.org/

models conforming to different languages). In consequence actual approaches for semantic composition are usually ensure that the coordination of the models conforming to these languages can be automatically obtained. There exist very different approaches to deal with this problem.

A first kind of approach, typified by **Formula**[21] [21] or **Modelyze**[22] [8], provides a formal environment where you can define, based on a specific form of the grammar, the domain-specific semantics of your language. This involves a translation of the domain-specific language into a representation suitable for formal anchoring in the targeted tool. In these approaches, the underlying semantics of these environment acts as a common semantic domain from which reasoning is possible. Based on a common representation of the semantics, it is then possible to specify how they are related. Such an approach provides very interesting formal verification capabilities, but a first drawback is the need for an arbitrarily complex transformation, which can make the semantics of the original model difficult to handle. This drawback is often pointed out in more classical approaches using translational semantics (*i.e.,* the translation of a semantic free language into a common formal representation). The other drawback relies on the existence of a common semantic background that must be expressive enough to be suitable for a wide variety of language while staying well founded for verification and validation activities.

A second kind of approach makes explicit the notion of a Model of Computation (MoC) [18, 26, 42]. In these approaches, a MoC specifies the causalities, timing and synchronization aspects of a language. First, making a MoC explicit enables fine tuning of the computational semantics, and usually offers simulation facilities. Second, it enables a clear specification of some semantic adaptations between different MoCs so that the semantics of heterogeneous models can be consistently coordinated. These approaches provide either the capacity to adapt to a domain-specific language (**Ptolemy** [18]) or **Modhel'X** [26]) or the capacity to drive formal refinement and reasoning on the system (**Forsyde** [42]), hence forcing the designer to choose between domain adequacy and analysis power.

A third kind of approach is based on the notion of meta-languages. It provides meta-languages for the specification of the domain-specific syntax (abstract and concrete) but also meta-languages for the specification of the semantics and its mapping to the syntax [14]. When using a meta-language like **Ecore** [45], you benefit from the associated generative techniques like the generation of a simple editor, its API, etc.In the same way, when using the meta-language for behavioral semantics specification, you can draw benefits from the automated generation of an interpreter and a explorer, making the models executable. The same advantage is obtained when using a meta-language for behavioral semantic composition, like BCOoL [49]. The main drawback of this approach comes from the meta-language for behavioral semantics specification, which is not suitable for an adequate specification of acausal models.

[21] http://research.microsoft.com/en-us/projects/formula/
[22] https://github.com/david-broman/modelyze

3 Open Challenges

The key challenge in the globalization of domain-specific languages is naturally how to compose multiple languages, both syntactically and semantically within a single software system. But there is another challenge related to collaboration in a globalized environment. It includes the management of the individual artifacts over time, environmental support for the viewpoints of multiple stakeholders, and scalable, persistent management of diverse models in a global environment.

We will survey each of these challenges and their associated research questions in turn.

3.1 Composition of multiple DSLs

The composition of multiple DSLs for the construction of a single software system entails a number of fundamental questions. How are such languages composed syntactically and semantically? Can we view DSLs as components, and, if so, what are their interfaces? How can we determine if languages are semantically compatible, and how can we check if the models expressed in them are consistent? Finally, how can we integrate legacy tools tied to individual DSLs within a common integrated system?

How do we compose languages? State of the art approaches have mainly focused on the syntactic integration of languages. They specify operators for *merging* the abstract syntaxes of different DSLs. A first challenge would be to identify and to classify the integration operators and their impact on the properties of the composed language [13, 19]. Existing approaches seldom deal with the behavioral semantics of the integration. Beyond the syntactic composition, another challenge is therefore to extend the classification to cover the semantic aspects of languages.

In many language composition approaches, the composition operators are specified on languages but the integration (*i.e.,* the merging) itself is applied on models. They use the knowledge of the meta-language to specify the composition. Yet another challenge is to understand if it makes sense to adapt such an idea to the behavioral semantics of language. In this case a directly associated challenge would be to understand what kind of meta-languages can describe the behavioral semantics of one language. Then, during the behavioral composition of languages, does the composition integrate the behavioral semantics of the languages or is it used to coordinate the behaviors of models that conform the languages?

Can we view languages as components? If so, what are their interfaces? Component-Based Software Engineering was quite successful in abstracting pieces of code or binary behind interfaces. Interfaces can be used to coordinate multiple components without requiring any knowledge of the components' internal implementation. This idea has penetrated many domain-specific languages so that models can be seen as components equipped with interfaces to enable their coordination. The challenge now is to see the languages themselves as components, meaning that they can be equipped with purpose-specific interfaces. Beyond the agreement that an interface is an abstraction of the language, the exact nature of the interface is far from clear. For instance, if one sees a language as a specification of a set of models describable in this language, then an interface could offer a way to specify the subset of models supported by the purpose the interface relates to. However, it can also be the set of operators, together with a characterization of what it accepts from the language. It is not clear also if the interface of a language can be used for language integration or only for language coordination.

Many sub-questions arises from this challenge *e.g.,* does it make sense to provide some family of language interfaces according to some purpose at the model level [1, 47]? Another research question is: in what language should a language's interface be specified? Should such a specification be reflexive at some point?

Does a composed system have a unified semantics? Semantic composition means that one can analyze the properties of the composition of a set of models expressed in several DSLs. This analysis can be manual, based on experts' knowledge, and possibly on a precise mathematical semantics of the composition, or it can be automated. If the analysis is to be automated, then the semantics of the composition of DSLs has to be implemented in some way. This can range from simple type-checking rules to the composition of heterogeneous behavior paradigms: for instance asynchronous processes coupled with time-triggered processes, or the coupling of discrete-time and continuous-time dynamical systems.

How can this be achieved in practice? At a first glance, a common semantic domain could be defined and implemented. Analysis would be performed using the methods and tools of this unified semantic domain. Unfortunately, unified semantic domains would become inconceivably complex when composing more than a few DSLs. Unified semantic domains would be very expressive, and not surprisingly, even the simplest analyses might turn out to be undecidable. The unified semantic domain approach certainly has practical value whenever the semantic domains to be unified are not too dissimilar.

Another approach is to avoid implementing a unified semantic domain, but rather to provide means to coordinate the different models (*e.g.,* by constructing on-the-fly combined heterogeneous behavior). This is best understood when the DSLs describe event-based discrete-time behavior, and each DSL comes with a transition system based operational semantics. In Gemoc Studio[23], behavior synchronization is achieved on-the-fly, without recourse to an explicit, unified semantic domain. In Ptolemy II[24], programs called directors are used not only

[23] http://gemoc.org/studio

[24] http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm

to define models of computation, but also to define how low-level synchronization between heterogeneous components is achieved.

Composing models with dissimilar semantic domains is largely an open problem. There are even semantic domains in which composability is difficult to achieve. A striking example is that of stochastic systems: they are difficult to compose, unless drastic stochastic independence assumptions are made.

Challenges in DSL semantic composition. Semantic composition can be achieved using several *ad-hoc* techniques, depending on the semantics to be composed. Event-based operational semantics, often used for state machines or dataflow programs can easily be composed by synchronizing the step functions of their concrete semantics. The same principle applies to timed extensions of these formalisms, namely timed automata, and the network calculus. The subject is however still in its infancy and no clear methodology has been proposed to address the composition of arbitrary semantic domains. We review below several hard cases of semantic composition.

Composition of discrete-time and continuous-time models. How can one co-simulate a system combining models with radically different semantics: discrete time dynamical systems on one hand, and continuous time dynamical systems on the other hand? The discrete time dynamics is often expressed in a data flow or automata based language, while the continuous dynamics results from a system of ordinary or algebraic differential equations (resp. ODEs and DAEs). Several techniques can be used to address this problem, depending on the overall system architecture and the assumptions that can be made on the overall system behaviour. These techniques range from simple asymmetric co-simulation methods, where time is handled by a unique numerical solver, to more involved techniques combining several numerical solvers. With the Functional Mock-up Interface[25], several models mixing continuous and discrete time dynamics can be co-simulated, with the restriction that the whole continuous-time dynamics is handled by a unique numerical solver. Proposals have been put forward to extend FMI, with for instance roll-back and step-size prediction mechanisms, to support a deterministic co-simulation with several variable step-size numerical solvers [17]. These techniques suffer from poor parallelism, and are difficult to implement on a distributed parallel architecture, making them unusable on large system models. Radically different techniques with good parallelism have been explored, but for limited classes of models. For example, Waveform Relaxation is a distributed simulation method for continuous-time dynamical systems, with superlinear convergence properties under mild Lipschitz smoothness assumptions [51]. An interesting challenge would be to extend Waveform Relaxation techniques to the hybrid systems case.

Composition of acausal models. A key challenge is the compositionality of acausal continuous time models. They are often expressed using algebraic differential equations (DAEs), where the data flow orientation of an incomplete

[25] https://www.fmi-standard.org

model may depend on its environment. This makes the generation of simulation code from a component model a difficult problem. The reason is that the environment of a component is not known before this component instantiated in a closed model. This problem becomes even harder when considering hybrid systems with DAEs, found for example in the Modelica language[26]. The main reason is that both the dataflow orientation and the differentiation index may change dynamically, depending on the discrete state of the model. This has severe consequences on the separate compilation, and the export of Modelica components encapsulated in a FMU[27]. Currently, this can be done only under the stringent assumption that the input/output orientation of variables appearing at the interface of each compilation unit is fixed, and that the differentiation index is invariant.

Composition of stochastic models. Composing stochastic models is a true challenge. Composition operators can be easily defined, under the assumption that the probability laws of the two models are independent. Unfortunately, this assumption most often makes no sense when considering models representing viewpoints of the same component. It turns out that these probability distributions are marginal probabilities of hidden probability distributions defining the stochastic behavior of the component. Marginal probabilities are in general not independent. Several stochastic system theories with good composability properties have been proposed. Unfortunately, their composition operators are involved [27, 33, 3, 41] and none of them have been implemented in a DSL, using the techniques developed in this book.

What is the difference between compatibility and consistency? A component model is said to be consistent if it admits at least one correct realization. Since components are often described according to several viewpoints, two models related to the same component are consistent if and only if there exists a common correct realization of both models. Hence, consistency is a logical property that applies to sets of models related to the same component.

Compatibility applies to models related to distinct interacting components. Two models are compatible if and only if there exists an environment of both components such that every realization of the two components can work together. This may have many different meanings. Type compatibility is the simplest form of compatibility. For example, the compatibility of two Interface Automata [39] means that there exists an environment that will prevent the occurrence of an output event whenever the peer component may not be ready to perform the corresponding input. This notion of compatibility appears also in several other specification formalisms, for instance Modal Interfaces [28], Sociable Interfaces [38] and Session Types [30]. Another fine example of compatibility is the Eiffel programming language [4], where preconditions and postconditions attached to methods are evaluated at runtime and can raise an exception whenever a component is incorrectly used.

[26] https://modelica.org
[27] functional mock-up unit, https://www.fmi-standard.org

How do we check compatibility and consistency of a composition? Consistency and compatibility are semantic properties of the composition of several components that can be checked statically, or by using model-checking techniques, or at runtime, depending on the semantic domains of the involved DSLs. Type-checking and timing constraints are classical examples.

How can we exploit legacy tools in an integrated system? Specific domains, such as mechanical engineering or control engineering, each have well-established tool sets that allows users to perform their tasks efficiently. Globalizing DSLs implies that different tools must work together. Therefore the integrated system output from the composition of DSLs must be able to exploit existing tools without modifying them. Although tool integration is not a new problem[28], it is important to re-use and not re-invent tools for specific DSLs. When composing DSLs, their affiliated tools must remain part of the globalization and therefore appropriate interfacing between the different tools must be investigated.

3.2 Collaboration in a globalized environment

Overall consistency along the lifecycle of DSL instances. Models are used to specify and capture different aspects of a system. Although each DSL has a specific purpose, their instance models might need to refer to elements from other models conforming to other DSLs. As such, a model can end up being coupled and inter-related to other models, where it is likely the referenced model is owned by a different stakeholder.

Coordination is required when a model is updated but a given stakeholder might not be aware how a provided model is being used by other parts of the design. There may be few means available to assess the impact that changes may lead to. Furthermore the client stakeholder who is consuming and referencing elements from the other model may have no other choice than to inspect the changes made by the provider to assess their impact on the client model. This leads to a contradiction: both stakeholders, each having different concerns and a dedicated language, have to understand the language of the other stakeholder to collaborate.

In this context coordination requires many interactions among stakeholders who are not even from the same domain. If DSLs can be processed in a unified way, and the language engineer is the most competent person to express requirements between language, shouldn't there be the potential of reducing and managing this coordination at the tool level? Coupling has to be considered as a first-class citizen by the language engineer: in a globalized environment in which language reuse and integration are no longer the exception but the general rule, the language engineer needs to express which concepts are suitable for being referenced or extended (probably among other possible relationships) and these might have an impact at the model level regarding instance evolution.

[28] Pheonix: http://www.phoenix-int.com/software/phx-modelcenter.php

The challenge is to provide a conceptual framework to the language engineer to extend DSL definitions so that their instances can be coordinated without requiring the different stakeholders to understand all the domains their work is coupled to.

How to characterize a change for stakeholders. Enabling the collaboration of different stakeholders through specific languages at a minimum requires changes to be expressed using a language any stakeholder will understand. Changes either need to have an obvious semantic for each stakeholder, or to be unique so that they can be understood by all stakeholders. Changes are the backbone of collaboration hence have to focus on three properties of socially translucent systems: visibility, awareness, and accountability [20].

How and when does the language engineer characterize model compatibility. In the context of multiple DSLs each having their own semantics we have to ask what is a "compatible" change. Is it a change that preserves the semantics of the model? Is it a change that preserves the fact that all model instances that were visible are still visible? Just going through several examples of DSLs it appears that there might be different aspects exposed by a DSL, each of them probably having its own notion of "compatibility".

In the SERS use case, autonomous vehicles are represented in the system in the form of a Computer Aided Design (CAD) model, parts of which are referred to by other models: the SmartIntersection model refers to a vehicle's position, while Mission Command & Control use the longevity and payload characterizations of the CAD model to reach control decisions.

When a CAD model evolves in reaction to a change in the vehicle's design or characteristics, it is very likely that decisions captured in other models need to be revised. On the other hand many evolutions of the CAD model will have no effect whatsoever on the other models. Since the CAD model provides different aspects, changes made in the CAD models might impact those aspects in a compatible or incompatible way, and consumers will need to assess this impact.

Furthermore in the context of a global collaborative process one has to ask when compatibility should be checked, and when consumers of models should be notified of incompatibilities. This can have a dramatic effect on the collaborative process: too late and much work will need to be done by the other stakeholders, too frequent and the stakeholders will use most of their time to align their work with the other changes. There is a need to be isolated yet informed; in this balance resides a key factor of collaborative process efficiency.

How should the concrete syntax be impacted by the use of an external language? The concrete DSL syntax is the primary means available to stakeholders to adapt and change the models. When parts of models are in semantic relationships with other elements from an external language, new aspects are mandatory to achieve a seamless use of multiple DSLs: concrete syntaxes have to be integrated (See the Syntactic Integration Chapter 2.3), and navigation between the syntaxes has to be considered a first class citizen.

How do we define a concrete syntax so that parts of it can be reused or merged with others, especially when the types referred to in the languages differ? We will might need to embed parts of a textual syntax into another textual language, or possible into a graphical language. What is the common ground to achieve these syntactic integrations?

Besides these questions, in an open world DSLs and their concrete syntaxes are not known beforehand and as such these issues should be adressed without any specific operation by the end-user. The challenge is that the role of a tool integrator doesn't exist in such context and the DSL and concrete syntax definitions themselves will have to be adapted so that the environment can provide such services at runtime.

Multi-view modeling shared by multiple stakeholders. The collaboration between stakeholders requires an appropriate tool support for sharing and jointly working on common models. There are three possible multi-view modeling scenarios [16]: (1) Stakeholders are working on exactly the same artifact: both of them share the same screen. All changes made by one stakeholder are directly reflected and perceived by the other stakeholders, such as in Google Docs. This situation is useful when, for example, two stakeholders are manually inspecting a model together, if one stakeholder is training the other, or in a development process favoring pair development. In this case, the collaboration is performed at the granularity of individual model elements and conflicting operations are resolved per element (or group of elements) as done in AToMPM [46]. (2) Stakeholders are working on different viewpoints of the same model. This situation is useful when artifacts are designed incrementally. This is possible when the language in which the artifact is described, offers a modularity mechanism that allows one to split its instances into different parts, such as partial classes in C# and aliases in UML diagrams. In this case, each viewpoint evolves separately, and changes are made locally to each viewpoint. At specific moments (on save or commit), changes from different viewpoints are merged into the underlying model. Conflicts that arise must then be resolved one by one by an expert as in WebGME. (3) Stakeholders with differing expertise are working on distinct models that, together, compose the overall system. Each artifact represents a concern of the overall system, e.g., the electrical, software, and the security concerns of an automotive. This is useful when a system is designed by separating its concerns, such as in aspect-oriented programming. This case requires traceability across DSLs. The traces need to be modeled explicitly in order to specify, at the language level, how conflicts are resolved automatically at the model level as in eMoflon [35].

Large scale model management. A collaborative modeling environment typically requires more storage space and more efficient model manipulation techniques than in a single-user modeling environment. Models grow in size more rapidly because multiple stakeholders are contributing and evolving them. Furthermore, traceability links between viewpoints, models, and DSLs must be stored. It is therefore of paramount importance to seek a suitable data model for persistent storage. Typically, all modeling artifacts are stored centrally on a distributed cloud server. Graph databases are of particular interest because they are optimized for graph representations of models as opposed to relational SQL database that have been shown to not perform as well [50]. Example candidates are: Neo4j which supports transaction processing [2], Trinity which virtualizes random-access memory of a cluster of computer nodes [6], and Apache Giraph which relies on the Hadoop paradigm [32]. A starting point for comparison is Shah *et al.*'s tool for benchmarking NoSQL databases to store models [44].

4 Conclusion

After presenting an overview of current work related to the composition of tools, models and languages, this chapter compiled a list of key open challenges related to both technical coordination of domain-specific languages and to social coordination of stakeholders in a globalized environment. While many challenges have to be addressed before achieving the globalization of modeling languages, the number of recent works that are currently paving the road toward this globalization makes these challenges very exciting.

References

1. Omar Alam, Jörg Kienzle, and Gunter Mussbacher. Concern-Oriented Software Design. In Ana Moreira, Bernhard Schtz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 604–621. Springer Berlin Heidelberg, 2013.
2. Amine Benelallam, Abel Gmez, Gerson Suny, Massimo Tisi, and David Launay. Neo4EMF, A Scalable Persistence Layer for EMF Models. In Jordi Cabot and Julia Rubin, editors, *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 230–241. Springer International Publishing, 2014.
3. Benoît Caillaud, Benoît Delahaye, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, and Andrzej Wasowski. Constraint Markov Chains. *Theor. Comput. Sci.*, 412(34):4373–4404, 2011.
4. Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1991.
5. Jean Bézivin, Hugo Brunelière, Jordi Cabot, Guillaume Doux, Frédéric Jouault, Jean-Sébastien Sottet, et al. Model driven tool interoperability in practice. In *Proceedings of the 3rd Workshop on Model-Driven Tool & Process Integration (co-located with ECMFA 2010)*, pages 62–72, 2010.
6. Bin Shao, Haixun Wang, and Yatao Li. The Trinity Graph Engine. Technical Report MSR-TR-2012-30, March 2012.
7. Torsten Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *8th International Modelica Conference, Dresden*, pages 20–22, 2011.

8. David Broman and Jeremy G. Siek. Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages. Technical Report UCB/EECS-2012-173, EECS Department, University of California, Berkeley, Jun 2012.
9. Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM.
10. Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International journal on software tools for technology transfer*, 6(3):203–218, 2004.
11. Andrea Caracciolo, Mircea Lungu, and Oscar Nierstrasz. A unified approach to architecture conformance checking. In *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. ACM Press, 2015.
12. N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.
13. Mickael Clavreul. *Model and Metamodel Composition: Separation of Mapping and Interpretation for Unifying Existing Model Composition Techniques*. PhD thesis, Université Rennes 1, 2011.
14. Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *SLE - 6th International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 365–384, Indianapolis, IN, États-Unis, 2013. Springer. CNRS PICS Project MBSAR (http://gemoc.org/mbsar).
15. James R. Cordy. The TXL Source Transformation Language. *Sci. Comput. Program.*, 61(3):190–210, August 2006.
16. Jonathan Corley, Huseyin Ergin, Simon Van Mierlo, and Eugene Syriani. *Modern Software Engineering Methodologies for Mobile and Cloud Environments*, chapter Cloud-based Multi-View Modeling Environments. IGI Global, 2015.
17. David Broman, Christopher X. Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Determinate composition of FMUs for co-simulation. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*, pages 1–12. IEEE, 2013.
18. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – The Ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, 2003.
19. Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8, New York, NY, USA, 2012. ACM.
20. Thomas Erickson and Wendy A. Kellogg. Social Translucence: An Approach to Designing Systems That Support Social Processes. *ACM Trans. Comput.-Hum. Interact.*, 7(1):59–83, March 2000.
21. Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: towards generic automation for MDA. In *EMSOFT*, pages 39–48. ACM, 2010.
22. Martin Fowler. Language Workbenches: The Killer-App for Domain-Specific Languages, June 2005.

23. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
24. Richard Frost and John Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, 1989.
25. George A. Papadopoulos and Farhad Arbab. Coordination Models and Languages. volume 46 of *Advances in Computers*, pages 329 – 400. Elsevier, 1998.
26. Cécile Hardebolle and Frédéric Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*, pages 247–258. Springer, 2008.
27. Igor Kozine and Lev V. Utkin. Interval-Valued Finite Markov Chains. *Reliable Computing*, 8(2):97–113, 2002.
28. Jean-Baptiste Raclet, Eric Badouel, Albert Benveniste, Benoît Caillaud, Axel Legay, and Roberto Passerone. A Modal Interface Theory for Component-based Design. *Fundamenta Informaticae*, 108(1-2):119–149, 2011.
29. Gabor Karsai, Andras Lang, and Sandeep Neema. Design patterns for open tool integration. *Software & Systems Modeling*, 4(2):157–170, 2005.
30. Kohei Honda. Session Types and Distributed Computing. In Artur Czumaj, Kurt Mehlhorn, Andrew M. Pitts, and Roger Wattenhofer, editors, *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, volume 7392 of *Lecture Notes in Computer Science*, page 23. Springer, 2012.
31. Gerhard Kramler, Gerti Kappel, Thomas Reiter, Elisabeth Kapsammer, Werner Retschitzegger, and Wieland Schwinger. Towards a semantic infrastructure supporting model-based tool integration. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 43–46. ACM, 2006.
32. Christian Krause, Matthias Tichy, and Holger Giese. Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In *Fundamental Approaches to Software Engineering*, pages 325–339. Springer, 2014.
33. Krishnendu Chatterjee, Koushik Sen, and Thomas A. Henzinger. Model-Checking omega-Regular Properties of Interval Markov Chains. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 302–317. Springer, 2008.
34. Frederick Kuhl, Judith Dahmann, and Richard Weatherly. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR Upper Saddle River, 2000.
35. Erhan Leblebici, Anthony Anjorin, and Andy Schrr. Developing eMoflon with eMoflon. In Davide Di Ruscio and Dniel Varr, editors, *Theory and Practice of Model Transformations*, volume 8568 of *Lecture Notes in Computer Science*, pages 138–145. Springer International Publishing, 2014.
36. Akos Ledeczi, Peter Volgyesi, and Gabor Karsai. Metamodel composition in the generic modeling environment. In *Comm. at workshop on Adaptive Object-Models and Metamodeling Techniques, Ecoop*, volume 1, 2001.
37. Lennart C. L. Kats and Eelco Visser. The Spoofax Language Workbench. Rules for Declarative Specification of Languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.

38. Luca de Alfaro, Leandro Dias da Silva, Marco Faella, Axel Legay, Pritam Roy, and Maria Sorea. Sociable Interfaces. In *Proc. of the 5th International Workshop on Frontiers of Combining Systems (FroCos'05)*, volume 3717 of *Lecture Notes in Computer Science*, pages 81–105. Springer, 2005.
39. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proc. of the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'01)*, pages 109–120. ACM Press, 2001.
40. Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding Languages Without Breaking Tools. In Theo D'Hondt, editor, *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *LNCS*, pages 380–404, Maribor, Slovenia, 2010. Springer-Verlag.
41. Samy Abbes and Albert Benveniste. True-concurrency probabilistic models: Markov nets and a law of large numbers. *Theor. Comput. Sci.*, 390(2-3):129–170, 2008.
42. Ingo Sander and Axel Jantsch. System modeling and transformational design refinement in ForSyDe [formal system design]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(1):17–32, 2004.
43. Elizabeth Scott and Adrian Johnstone. GLL Parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, September 2010.
44. Seyyed M. Shah et al. A Framework to Benchmark NoSQL Data Stores for Large-Scale Model Persistence. In *Proceedings of MODELS'14*, volume 8767 of *LNCS*, pages 586–601. Springer, 2014.
45. David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2009.
46. Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A Web-based Modeling Environment. In *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, volume 1115, Miami FL, USA, 2013. CEUR-WS.org.
47. J. Sztipanovits, T Bapty, S. Neema, L Howard, and E Jackson. OpenMETA: A Model and Component-Based Design Tool Chain for Cyber-Physical Systems. In *From Programs to Systems – The Systems Perspective in Computing (FPS 2014)*, Grenoble, France, April 6, 2014 2014. Springer, Springer.
48. Masaru Tomita. *Efficient parsing for natural language: A fast algorithm for practical systems*, volume 8. Springer, 1985.
49. Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A Behavioral Coordination Operator Language (BCOoL). *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (Models)*, 2015.
50. Gergely Varró, Katalin Friedl, and Dániel Varró. Implementing a Graph Transformation Engine in Relational Databases. *Journal on Software and Systems Modeling*, 5(3):313–341, 2006.
51. J. White, F. Odeh, Alberto L. Sangiovanni Vincentelli, and A. Ruehli. Waveform Relaxation: Theory and Practice. Technical Report UCB/ERL M85/65, EECS Department, University of California, Berkeley, 1985.