Multi-Language Development of Embedded Systems

Thomas Kuhn
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2177
thomas.kuhn@iese.fraunhofer.de

Soeren Kemmann
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2218
soeren.kemmann@iese.fraunhofer.de

Mario Trapp
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2272
mario.trapp@iese.fraunhofer.de

Christian Schäfer
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserslautern
+49 631 6800 2121
christian.schaefer@iese.fraunhofer.de

ABSTRACT

Graphical, well focused and intuitive domain specific languages (DSLs) are more and more used to design parts of embedded systems. These languages are highly specialized and often tailored to one domain; one single language therefore cannot describe all relevant aspects of systems and system components. This raises the need for heterogeneous modeling approaches that are capable of combining multiple DSLs into holistic system models. Our CompoSE modeling approach focuses on this problem; it does not only cover system modeling with DSLs, but provides also interfacing of language specific generators and harmonization of generated code. In this paper, we describe the principles of CompoSE, together with the integration of an existing modeling language with industrial strength tool support into CompoSE. Supporting the integration of existing languages is of particular importance in the domain of embedded systems, because modern modeling approaches will only be accepted in industry if they support existing and proven technologies.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General – *System specification methodology*

General Terms

Design, Languages

Keywords

System modeling, Domain specific languages, Multi formalism development

1. INTRODUCTION

Development of embedded systems in research and industry is more and more shifting from code based development to model driven development (MDD) approaches, which are founded on high-level modeling languages. Modeling languages are not as generic as general purpose programming languages, they provide more specialized language constructs, e.g. for the creation of data flow based systems (cf. Simulink) or for the creation of system models (cf. SysML). These MDD approaches are supported by industrial strength tool chains; prominent examples of MDD tools that are applied in both research and industry are Simulink, ASCET, SCADE, Rhapsody, Artisan, and MagicDraw. MDD tools implement modeling languages, provide infrastructure support, e.g. tailored editors and code generators, and include runtime libraries and frameworks that support execution of generated code. Domain specific languages (DSLs) are more specialized than generic MDD approaches; being tailored to a specific application domain, they enable domain experts to express themselves with native constructs of their respective domains. One example for DSLs is a graphical language for creating wiring diagrams. These modeling languages are either implemented as language profiles, e.g. as UML profiles, or they are built on top of existing language frameworks (cf. Eclipse GMF or MetaEdit+). In both cases, DSLs provide their own tool chains and model to code transformations.

This is a major challenge for the development of embedded systems: generic and domain specific modeling languages are limited and support some aspects of embedded system development only. Simulink for example supports definition of data flow based behavior only, UML based languages support the definition of software architecture and control flow, and SysML supports the definition of system architectures. Graphical editors, code generators, and language frameworks only support one or a limited set of modeling languages. Detailed modeling of all aspects of complex embedded systems therefore requires the combination of models defined in multiple modeling languages and tool chains to provide one holistic system model. Code generation needs to be done with multiple independent generators in this case. This yields the situation that developers need to combine multiple generated artifacts and runtime libraries, and need to connect required inputs and provided outputs of models, which may even implement different semantics. One common execution model is required that supports all relevant modeling languages. This nontrivial task currently limits the applicability of DSL approaches in development of complex software systems, since the effort required for integrating modeling languages may outweigh the additional benefits of modeling languages.

CompoSE is our multi formalism modeling approach that supports the integration of modeling languages on language, infrastructure and runtime levels. Being independent of concrete modeling languages, it defines a common host component model, which supports multi-formalism development of embedded systems. Guest modeling languages are integrated as language components; these languages are applied for modeling functional and non-functional details of system components. Support is provided for the integration of general purpose, and domain specific modeling languages. Language components address all of the three aforementioned layers: they provide integration of modeling languages at language, infrastructure, and runtime levels.

The remainder of this paper explains CompoSE principles and is structured as following: Section 2 describes the basic principles of our multi-formalism development approach. Section 3 describes the CompoSE host language in greater detail. Section 4 provides an application example that illustrates briefly the integration of Simulink into CompoSE. Section 5 surveys and discusses related approaches. Section 6 draws conclusions and lays out future work.

2. MULTI-FORMALISM DEVELOPMENT

CompoSE supports multi-formalism development through the integration of independent modeling languages and tools into one multi formalism framework (*MFF*). Despite the independency of the different languages and tools, the MFF ensures their seamless combination for the creation of integrated system models. This is achieved through the application of component based development basic principles to the domain of language engineering.

CompoSE is based on the principle of one host language and several guest languages. The host language defines system components and a basic set of views for modeling system architectures; it also defines language constructs for the integration of language components. Language components integrate guest languages into CompoSE. As shown in Figure 1, language components address the three main elements of modeling languages to ensure their seamless integration: view types provide integration on language level, infrastructure interfaces integrate tool chains, and runtime interfaces ensure interfacing of generated code with a common runtime model.

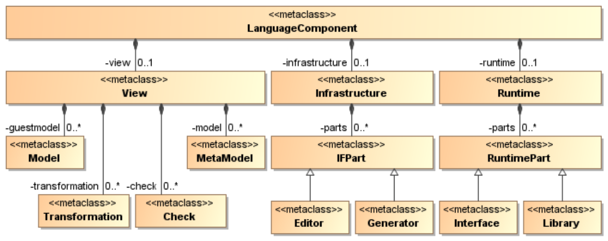

Figure 1: Language component meta model

Language components provide one or multiple view types that integrate modeling languages. Compose additionally provides several predefined view types that support definition of interfaces (*InterfaceView*), aggregations (*AggregationView*), and coupling (*CouplingView*). Native models of integrated existing modeling languages are stored in guest models, which are containers that conform to some unknown, guest specific format, and are therefore not directly accessible. Meta models support the (bidirectional) projection of guest model parts into the host model through transformations – this way, information stored in guest models is made accessible, and is shared and synchronized between guest models and language components. Additionally, views represent their modeling languages to developers and therefore support the manipulation of their underlying models – for this reason, infrastructure parts (*IFParts*) are used to expose language infrastructure, e.g. graphical editors.

Infrastructure parts (*IFParts*) enable the integration of existing language tool chains. These parts implement proxies that provide common interfaces to the CompoSE MFF and hide native interfaces of language specific tools. Runtime parts of language components define the runtime interfaces of generated code; when existing tool chains are integrated into CompoSE, they model the interface between generated code and existing, language specific runtime frameworks. A CompoSE runtime framework then provides glue code that interfaces generated code for each language component with each other and that conforms to a common runtime specification. Note that CompoSE does not include a specific runtime environment, but it defines common requirements that conforming runtime environments need to fulfill. These requirements define syntactic and semantic constraints that runtime framework implementations need to conform to. Adapter code that is generated by generator proxies serves as interface between the generated code from language specific tools (whose interface is defined through language components) and the runtime framework. Figure 2 provides an example – two host components are defined: the component *Control* that realizes a data low based controlling algorithm, and a *Filter* component that preprocesses data for the *Control* component. The *Control* component is realized with Simulink, the *Filter* component is realized with a domain specific language. Therefore, two language components provide necessary views, infrastructure, and runtime support.

The Simulink language component provides the Simulink realization view, integrates the native Simulink tool chain, which consists of a code generator (*Simulink Generator*) and of the Simulink runtime framework. It also includes the *Simulink Proxy* that generated adapter code (*Simulink Adapter*), which interfaces generated code by Simulink with the common runtime framework. The DSL language component provides a view that supports editing models based on its domain specific language together with a code generator. No proxies and adapters are required, since the code generator outputs conforming code directly.

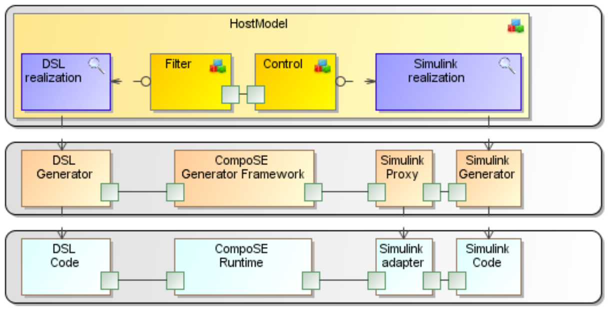

Figure 2: Compose Multi Formalism Framework

3. THE COMPOSE HOST LANGUAGE

The CompoSE host language implements a component modeling approach that is based on components, properties, ports and links. Components represent parts of the developed system, which are either black or white boxes. Ports belong to components and define points of interaction that links are connected to. Properties store component information – three types of properties are defined by CompoSE: Guest model properties store complete guest models in their native format. Meta model properties are based on a CompoSE conforming meta model definition, and represent containers that store models conforming to those meta models. Primitive properties store one type, e.g. an integer or a structured type. Properties are subdivided into two property types: Specification properties define whole or partial component specifications. Component specifications define what a component does, and how a component is to be used. Specification properties are associated with specification views. Realization properties define how a component is realized – they are associated to realization views. Component realizations always need to conform to the specification of their component.

3.1 CompoSE language components

Language components integrate new modeling or domain specific languages into CompoSE that are used for defining component details. On language level, language components consist of views, transformations, and models.

Views present data, which is represented by models. For this reason, two model types are distinguished: guest models and meta models. Existing modeling languages that ship with their own tool chains usually store data in their own container format, e.g. a language specific binary representation. Models stored in such a container are referred to as guest models. Other language components cannot access this data, since file format and structure is not known – guest model properties are therefore black boxes for other language components. New, CompoSE conforming DSLs store all of their data in containers that conform to defined meta models instead, therefore, this data may be accessed by other language components – this is a white box representation.

3.1.1 Guest model synchronization

If a language component uses a guest model representation for storing models, these models are not accessible for other language components, which may be cumbersome. For example, a Simulink view defines component realizations as data flow between input and output flow ports of components. The *InterfaceView* (see below), which is a predefined and therefore language independent view of CompoSE, defines component ports as part of the component interface as well. Both views therefore store the same information in different properties: the Simulink view stores component ports as part of its Simulink guest model, the interface view stores component ports in a meta model property.

This situation is not satisfactory for developers using CompoSE – they need to manually ensure consistency between views. Existing tools for UML for example provide this synchronization between diagrams that operate on the same model automatically – changes in the model through one diagram are immediately reflected in all other diagrams. CompoSE provides a similar functionality through transformations in a manner that supports multiple modeling languages. Transformation components implement model to model transformations; they are part of language component views and therefore implement a bridge between host and guest models. Transformations may be applied to transform models conforming to one meta model into a model that conforms to another meta model of the same component, to modify models, and to transform models into guest models and back. Guest models may only be accessed by the language component that defines them, and each guest model type may be defined by one language component only. Through transformations, complete guest models or parts of it are projected into models that conform to defined meta models, and are therefore accessible by other language components (see Figure 3).

In the example defined by Figure 3, two views are attached to the system component type *Control*. This component type has three properties – the first property *Interface.interface* defines the component interface and contains data conforming to the interface meta model defined by the common *InterfaceView*. It is manipulated through the *interface specification* view. The *Simulink.simulink* property holds the guest model of the Simulink realization, and is manipulated through the Simulink realization view. Model transformations synchronize the Simulink guest model and the interface meta model with the Simulink flow meta model, which is a common white box representation. This model is not manipulated directly through a view, and stored in the *Simulink.flow* property.

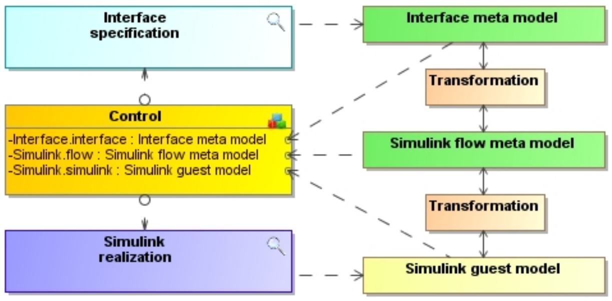

Figure 3: Components, models, views, and properties

As shown in the example, component data is stored in properties. Therefore, complex components possibly require a large number of properties to store the data of all views. Additional data that is shared between views, for example ports, attributes, and operations are stored in properties as well. Therefore, to prevent namespace pollution, the name of a property is composed out of a language component that its type belongs to, together with its identifier (see Figure 3).

3.1.2 Specialization

Language component hierarchies support the concept of specialization, which is known from other languages, e.g. from the MOF or from the UML. However, specialization of language components needs different semantics to ensure proper handling of views, infrastructure, and runtime. Specialized language components inherit all elements of more generic components and may override them. Specialized components, for example, define new guest models, new transformations, and new meta models. Existing transformations and meta models are possibly extended by specialized language components. Infrastructure parts of parents may be inherited or overridden. In the latter case, the existing infrastructure (tools, editors, code generators…) of the parent may be used by the infrastructure of the specialized language component. Runtime interfaces may be inherited or overridden, but overriding is only permitted with more specialized interfaces that at least provide the functionality of the base interface type. Figure 6 illustrates an example for language component specialization. The base component *GenericLanguageComponent* defines a framework for all subsequent language component definitions. The component *DataflowLanguage* redefines the language view and the runtime. The *DataflowView* view adds a data flow meta model, the *DataflowRuntime* component adds a data flow runtime interface. The Simulink language component extends all three views. Therefore, all existing elements are inherited first. The meta model *SimulinkMM* may only extend the more generic *DataflowMM* meta model, since it is its specialization. The guest model definition is a new language component element. Simulink infrastructure are new language component elements as well. The Simulink runtime interface *SimulinkRuntimeIF* replaces the old *DataflowRuntimeIF* with a derived and specialized interface.

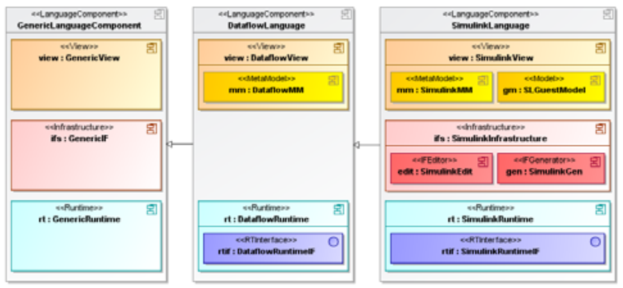

Figure 4: Language component specialization

This inheritance scheme supports language component hierarchies, e.g. the data flow hierarchy from the example. Specialized language components may introduce new guest models and tool chains but still re-use meta models of parent language components. Guest and meta model properties that are qualified with the type of their defining language component retain their type specifier; derived meta model types or replaced guest models therefore appear with their original qualifier. Therefore, meta models may only be extended through specialization and therefore are downward compatible to meta models of base components. Guest model access restrictions ensure that only transformations and infrastructure of one language component type may access the same guest model, and therefore also ensure that specialization does not lead to type conflicts.

3.1.3 Conflicting view types

The special relation *conflictingView* may be defined for any pair of views that must never be used together on one component. Specialized view types inherit this property from their base views. This way, it is ensured that conflicting realization views are never used together to define one component. This is handy if two language components are not sufficiently synchronized, but enable definition of similar things. For example, both Simulink and ASCET views define (different) data flow models. In order to operate properly on the same component, both views need to synchronize their whole model into a common meta model. While this is possible with CompoSE through transformations, this is impractical in real world applications. For this reason, both view types could be marked as conflicting instead, preventing developers to use them together on the same component.

3.1.4 Checks

Automated checks are executed similar to transformations every time when a connected property was modified. In contrast to transformations that produce output models, checks validate predefined properties or consistency rules. Typical application areas for automated checks in the CompoSE framework are DSL specific consistency checks across views. Similar to transformations, checks are currently developed in Java; for subsequent implementations, we plan the development of a DSL for specifying *OpenArchitectureWare* (OAW) based checks and model transformations using OAW's extend language.

3.2 CompoSE components

CompoSE components represent all system components – in this paper, we focus on the definition of software components though. Components are defined through properties – property values are modified through views. CompoSE supports two basic relations between components that are known from the UML: Component aggregation and component specialization. Currently, no distinction between aggregation and composition is made in CompoSE. However, due to the view concept, the behavior of both principles needs to be adapted.

Component aggregation is supported through the basic view type *AggregationView*. Aggregated components, i.e. components that consist of other components, are created through component aggregation only; no other non-aggregation realization views may be assigned to that component type. The two view types *AggregationView* and *NonAggregationView*, from which all non-aggregating view types derive are therefore marked as conflicting views. The realization of aggregated components is therefore only defined through the aggregation view – no other realization views may be applied to that component. Component specifications are not affected by aggregation views. Therefore, specifications of aggregated components are defined through specification views similar to any other view type. The aggregation view of CompoSE is similar to the composite structure diagram of UML that defines component substructures through instances, ports, and links. Figure 5 illustrates an example component aggregation. In the example, the component *CruiseControlSystem* is aggregated out of one instance of the component type *Filter* and one instance of the component type *Control*.

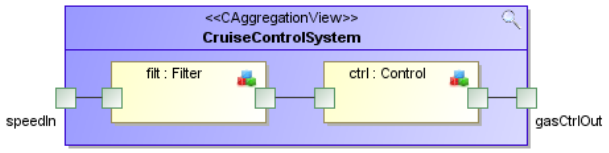

Figure 5: Aggregation view

Component specialization is more complicated than component aggregation, because it affects both component specification and realization views with unknown language semantics. Component specifications and realizations are defined through properties. Specialized components initially inherit all properties of their derived components. In addition, specialized components also may override properties of their base components, and therefore replace their value. This must be done through a compatible view that is able to modify the property in question. However, when replacing property values, the following additional restrictions apply, which are specific to guest languages and therefore are validated automatically by checks (cf. Section 3.1.4).

- Properties defining component specifications may not be lowered by specialized components - everything that was defined by the specification of the parent component must still be part of the specification provided by derived components.
- Properties defining realizations may be overridden and changed by specialized components as long as the component specification, and therefore inherited component specifications are met.

Depending on the guest language, language specific specialization constructs may be available. For example SDL and UML languages provide such concepts, while Simulink does not support type inheritance. If specialization constructs are available in a guest language, these constructs may be used for creating specialized guest models based on their parent guest models from parent components. Figure 6 illustrates this with an example.

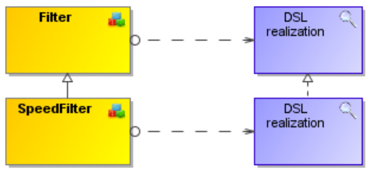

Figure 6: CompoSE component specialization

The example from Figure 6 illustrates CompoSE component specialization. The *SpeedFilter* component specializes the more generic filter component type. It also replaces the DSL based component realization. DSL dependent specialization constructs support specialization of the original realization of the base component. CompoSE aggregation and specialization are intentionally not strict and enforcing, because this would limit the applicability of CompoSE to a smaller number of guest modeling languages. CompoSE defines a necessary and sufficient set of constraints for aggregation and specialization that enable creation components and views with defined semantics.

4. MULFI-FORMALISM COUPLING

Up to now, we defined the integration of language components and therefore new modeling languages into Compose. View types define properties, meta models, and therefore containers for storing information. Transformations map models from one meta model into another, and bridge between guest models. Views also provide means to modify model elements by integrating infrastructure parts that represent existing language infrastructure, e.g. editors. Runtime adapters provide an interface between code generated by different language infrastructures, i.e. code generators, and bridge generated code. This works well as long as languages with conforming runtime semantics are coupled – for example Simulink and ASCET provide similar semantics, and therefore coupling is simple. However, multi formalism development requires the combination of modeling languages that implement different semantics. Bridging these languages is a challenging task, and the approach used for providing this coupling is an important design decision.

CompoSE supports this bridging through the *InterfaceView* and the *CouplingView* view types. The *InterfaceView* predefines port types that represent different semantics. The *CouplingView* supports connections between ports that represent similar semantics, or between port types for which a semantic mapping is defined. Basic port types that are defined by the *InterfaceView* are the following:

- Data flow ports (*FlowPort*) represent data flow semantics.
- Event ports (*EventPort*) represent asynchronously transmitted and received events.
- Control flow ports (ControlFlowPort) represent control flow semantics, e.g. the definition of operations with entry points and control flow transfer upon invocation.

The coupling of components that provide interfaces based on these port types, and therefore implement conforming semantics is defined through the coupling view. While coupling of compatible interfaces is trivial, the coupling view also defines coupling semantics that map from one interface type to another. Runtime adapters need to support these predefined mappings in order to support semantic coupling of their supported modeling languages. The following automated mappings are currently supported by CompoSE:

- Output data flow ports map to event ports by generating an event each time the output value changes. Event ports are mapped to input data flow ports by changing the respective data value each time an event is received.
- Control flow ports map to output data flow ports, if they define one operation with one parameter only that is then invoked upon parameter change. Mapping of control flow ports to input flow ports is supported if one operation is provided that carries one parameter, which will be the new value of the flow port.
- Mapping between event ports and control flow ports is supported as following: Whenever an event leaves an event port, which is connected to a control flow port, the corresponding operation will be invoked. If a response event is declared, the return value will be transmitted back upon the operation is completed. Operation execution is not synchronized with the execution of the component that transmitted the event. Mapping of control flow ports to realize required operation requires the definition of request/response pairs of events. The execution of required operations is mapped to the transmission of the request event. The calling component is suspended, until the corresponding response is required, in order to conform to control flow semantics.

By generating explicit adapter components using any supported language, more sophisticated mapping may be explicitly defined in addition to these predefined mappings. This coupling approach documents the basic rationale of compose to support black box components with defined white-box interfaces and properties; definition of component semantics are supported in a similar manner. While CompoSE is not aware of the complete semantic model of its black box components, it is aware of their interface semantics, and therefore is able to connect them to each other. Runtime semantics of components are supported in a similar way.

CompoSE predefines semantic models that are connected to views; not components. They are therefore stored in a property of the view type. These semantic models represent an abstraction of the runtime semantics of integrated modeling languages – since components may support multiple views (as long as restrictions regarding incompatible view types are not violated), they may as well implement different semantics. Predefined semantic models need to be supported by all runtime infrastructures. Additionally, new semantic models may be defined; the support of these models is then optional to runtime frameworks. The following semantic models are predefined:

- Data flow semantics realize a continuous data flow that continuously recalculates output values.
- Event based semantics provide semantics that realize views defining asynchronously executed behavior, which is triggered by events.
- Control flow semantics are passive – views using these semantics define component behavior that is triggered only through active transfer of control flow through control flow ports.

- Active control flow semantics are used to realize views which may receive control flow through control flow ports, but still provide an behavior on its own that is independent from explicit control flow transfer from other components.

Runtime frameworks, as already mentioned, need to implement at least these four semantic models. They also need to support components that apply different semantic models for different views.

5. COMPOSE APPLICATION EXAMPLE

In this section, we describe the integration of Simulink, an existing modeling language for data flow models. The Simulink language component supports the definition of component specifications and realizations through two different views, which consequently affect two different component properties. Simulink is supported by an industrial strength tool chain; this tool chain is integrated through the infrastructure interface into CompoSE.

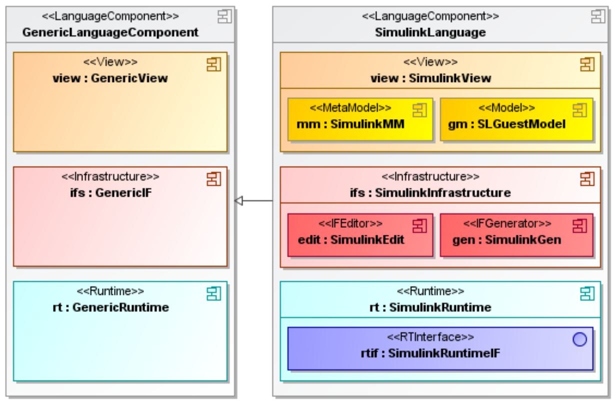

Figure 7: Simulink language component

Figure 7 describes the new language component *SimulinkLanguage*. New CompoSE language components extends directly or indirectly the type *GenericLanguageComponent*. The Simulink language component defines a white box meta model *SimulinkMM*, a guest model *SLGuestModel* that represents native .mdl files, and a model to model transformation that transforms parts of the native model into the white box model (not shown in Figure 7). Simulink views implement data flow semantics; their provided infrastructure is defined through an editor and a generator proxy. The definition of these infrastructure components is implementation specific. In our case, CompoSE was implemented into MagicDraw, which is a generic UML modeling tool. Infrastructure components are magic draw plugins that integrate code generation capabilities by calling code generators of integrated tool chains, or by invoking editors of generated tool chains. The editor connects to the Simulink editor, which is also part of the commercial tool chain. It is invoked in a similar manner as one if MagicDraws native UML diagram editors; however, changes in the model are synchronized only after saving in our implementation. Simulink diagrams are stored together with its native model representation in the guest model; after saving a diagram, transformations are invoked that extract relevant data from the diagrams and update component properties and white box meta models. The runtime interface of generated code is defined by the *SimulinkRuntimeIF* interface. Native Simulink code does not conform to that interface. Therefore, adapter code is generated by the *SimulinkGen* infrastructure proxy to mediate between the CompoSE runtime and generated Simulink code.

6. RELATED WORK

The author of [1] proposes a generic, component based framework for the evaluation of quality attributes like timeliness and safety. Each component gets four artifact types assigned: an encapsulated evaluation model, an operational/usage profile, composition algorithms, and evaluation algorithms. Based on these artifacts, a process for the evaluation of quality attributes is defined. This approach focuses clearly on evaluation of white box models; in contrast, CompoSE focuses currently on the efficient integration of new modeling languages as black box models, as well as providing an extensible framework for synchronizing information contained in different black box models.

The work presented in [2] present BIP, a component based development approach that supports multi formalism development of behavior components. BIP defines three layers per component, focusing on component behavior, interaction, and execution. The authors focus on behavioral realizations, and provide a framework for modeling components, as well as for the generation of glue code to link these components together at runtime. This approach focuses on correctness by construction and the adherence to properties while composing components, e.g. ensuring deadlock freedom. This is done via one common modeling language that component behavior is mapped to; in contrast, CompoSE provides the ability of integrating any language as language component. BIP could be used as a backend framework for performing formal analysis by defining the language of BIP as a white box meta model, and by providing transformations for guest models of language components into the formal language of BIP.

The authors of [3] present a framework for multi language development of embedded systems, which provides tool and model integration. Modeling languages are integrated into the proposed framework through adaption layers that provide a link between domain specific models and the common framework. Like CompoSE, connected models are divided into public parts that are exposed to other models, and private parts that are stored in a common repository, but will not be exposed. In contrast to this work, the basic framework described in [3] implements multi-language development on the tool level, not on modeling level. Therefore, no modeling language for the combination of multiple modeling languages is defined.

The authors of [4] provide an approach for multi formalism development that is much more tightly integrated than CompoSE; the described approach aims at integrating the meta models of used modeling languages. This is one difference between CompoSE and the approach presented in [4]: while CompoSE uses a central system model to synchronize modeling elements, the authors of [4] directly synchronize meta models with each other. While this approach is certainly appealing, the creation of the proposed consistency checking and mapping meta models costs considerable effort when defining a multi-language modeling approach for multiple already existing modeling languages. Depending on the amount of exported data and the complexity of the synchronization rules in views, CompoSE might provide such a tight synchronization as well. However CompoSE scales; in most cases full meta model synchronization is not required. Therefore, CompoSE will only synchronize a small subset of model data, resulting in a smaller and therefore less expensive synchronization approach in these situation (regarding both money and required processing power).

Ptolemy, which is presented in [5], is a famous approach for the application of execution semantics in Java environments, as well as for their evaluation, simulation, and composition. The focus of Ptolemy is on the semantic coupling, and simulation of components that implement different execution semantics. However, other aspects besides runtime semantics, e.g. the integration of modeling languages, tools, and (meta) model synchronization is not covered. Therefore, both approaches, Ptolemy and ComposE have a slight overlapping, which is the coupling of semantics. This coupling is currently in Ptolemy much more developed than in CompoSE, which focuses on the integration of modeling languages and light-weight model synchronization.

The authors of [6] describe Metropolis, which is a component based modeling framework, which is based on the following core concept of separation between communication and computation, and separation of functionality and architecture. Metropolis provides a common meta model that most existing models of computation may be transformed into. The metropolis model of computation is based on concurrent execution of action sequences; actions are subdivided into communication and computation actions. The main difference between CompoSE and Metropolis is its focus: CompoSE is an approach that aims at integrating (domain specific) languages, infrastructure and runtime frameworks in a light-weight manner. Runtime frameworks are combined using common runtime interfaces – as long as a runtime adapter and semantic mappings are provided, a specific language may be integrated into CompoSE. Metropolis provides a common model of computation that languages are transformed into. This requires a much more tight integration with respect to runtime models, and therefore much more integration effort.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented CompoSE, our multi-formalism modeling framework. CompoSE has been devised by applying principles from component based software engineering to the creation of a multi formalism modeling approach. It supports multi formalism development at three levels: the modeling level, the infrastructure level, and the runtime level. The principle for multi-formalism development with CompoSE is the application of guest models and guest languages that are plugged into one host model as language components. Views provide access to guest languages at modeling level and present data stored in models. Guest models are stored in their native file format and meta model of the guest language, meta models may be used to export whole guest models or part of it into the host model, so that other language components may access this information. This transformation is performed though explicit transformations. The CompoSE approach is different from most other approaches, because it provides a light weight language integration; the degree of language integration depends on provided meta models and transformations, and may therefore be adapted. This is an important aspect for its practical applicability, where integration effort equals to money.

Through the concept of guest models, existing languages, infrastructure, and runtime frameworks may be used with CompoSE. This is especially important in industry, because multi formalism approaches are only accepted if they support established and well proven tool chains. The separation between black box guest models and white box meta models enables a rapid integration of new modeling languages, because only relevant attributes of guest models need to be synchronized with the host model; full meta model synchronization is possible, but not necessary with CompoSE. We have proven the applicability of CompoSE through the integration of the existing Simulink language as language component.

Ongoing and future work with respect to CompoSE is the definition of a set of views for systems modeling in the automotive industry. Additionally, the definition of formal semantics for CompoSE language constructs, relations, as well as for language and formalism coupling is currently ongoing work. Once this is finished, clear coupling semantics will be available, as well as an approach for the integration of new coupling semantics.

8. REFERENCES

[1] L. Grunske, *Early Quality Prediction of Component-Based Systems - A Generic Framework*, Journal of Systems and Software, Elsevier, Volume 80, Issue 5, May 2007, pp. 678-686

[2] G. Gössler, J. Sifakis, *Composition for Component-Based Modeling*, Science of Computer Programming, Volume 55(1-3), 2005

[3] J. El-Khoury, O. Redell, M. Törngren, *A Tool Integration Platform for Multi-Disciplinary Development*, Proceedings of the 2005 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'05), Porto, Portugal, 2005

[4] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, A. Zündorf, Tool Integration at the Meta-Model Level within the FUJABA Tool Suite, Proceedings of the Workshop on Tool-Integration in System Development (TIS), 2003

[5] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong: Taming Heterogeneity - the Ptolemy Approach. Proceedings of the IEEE, v.91, No. 2, January 2003.

[6] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and Designing Heterogeneous Systems, volume 2549 of LNCS, pages 228–273. Springer-Verlag, 2002.