

Model-Integrated Development of Embedded Software

Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, Ted Bapty
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

The paper describes a model-integrated approach for embedded software development that is based on domain-specific, multiple view models used in all phases of the development process. Models explicitly represent the embedded software and the environment it operates in, and capture the requirements and the design of the application, simultaneously. Models are *descriptive*, in the sense that they allow the formal analysis, verification and validation of the embedded system at design time. Models are also *generative*, in the sense that they carry enough information for automatically generating embedded systems using the techniques of program generators. Because of the widely varying nature of embedded systems, a single modeling language may not be suitable for all domains, thus modeling languages are often domain-specific. To decrease the cost of defining and integrating domain-specific modeling languages and corresponding analysis and synthesis tools, the model-integrated approach is applied in a metamodeling architecture, where formal models of domain-specific modeling languages – called metamodels – play a key role in customizing and connecting components of tool chains.

The paper will discuss the principles and techniques of model-integrated embedded software development in detail, as well as the capabilities of the tools supporting the process. Examples in terms of real systems will be given that illustrate how the model-integrated approach addresses the physical nature, the assurance issues, and the dynamic structure of embedded software.

Keywords

model-based development, system and software engineering, model verification, automated software engineering, software generators, design-space exploration, model-integrated computing, embedded systems

1. Introduction

The development of software for embedded systems is difficult as these systems are part of a physical environment whose complex dynamics and timing requirements they have to adhere to. Embedded real-time systems should produce not only correct outputs, but should produce them at the right time. Furthermore, a “reasonable” behavior is expected from these systems, even under fault scenarios, when hardware or software components fail. Conventional software development considers timing, reliability, robustness, power consumption as “non-functional” requirements, which are typically secondary to the logical (or functional) correctness of computations. In embedded software development logical correctness is only one aspect of the design; physical characteristics of computations are equally important hence they must be included in the design process.

Unfortunately, current design and implementation strategies used in the practice do not provide enough support for this. Consequently, decisions in seemingly unrelated aspects of the design can impact the physical behavior of the resulting embedded computing systems unexpectedly. For example, an aircraft will handle rather differently if its fly-by-wire system works with 25Hz frame time instead of 80Hz. The increase in the length of the frame time may be the result of a thread allocation decision, for instance placing a highly utilized computing thread on a processor with a slow network connection. Such impact of design decisions is often not detected until it is too late —at system integration time, or, even worse, in operation.

In this paper we describe an approach, called Model-Integrated Computing (MIC) [1], which is based on *models* and *generation*, and which provides a flexible framework to address essential needs of embedded software development. MIC fully adopts the model-based development paradigm: models are used not only to design and represent, but also to synthesize, analyze, integrate, test, and operate embedded systems. Models capture not only what the dynamics and the expected properties of the system are, but also what is assumed about the system’s environment.

MIC introduces modeling languages that allow representing all relevant information in the form of models. We do not believe that a single modeling language is suitable for all embedded systems. Rather, embedded systems should be modeled using *domain-specific modeling languages (DSML)* that are tailored to the needs of the particular domain. To address the needs of defining and implementing DSMLs, MIC has a built-in extension mechanism: *metamodeling*, and the corresponding *meta-generation*. Domain-specific modeling tools can be created using meta-programmable modeling environments, and the models created by those tools can be translated into other forms using meta-generation technology.

The paper introduces the concepts of MIC through an example, then it discusses the modeling and metamodeling techniques and the supporting tools. Next, it shows how model-based generators could be implemented, and gives an example for the model-based development process. The paper concludes with the overview of related work and a discussion on research directions for the future.

2. Overview of MIC

To illustrate the concepts and techniques of MIC, we consider an example: the building of a high-performance digital signal processing (HDSP) system. The term “high-performance” means that high I/O bandwidth and throughput requirements mandate the use of multiple signal processors configured in an application-specific hardware architecture. There are a number of highly relevant application areas for HDSP systems, e.g. high-frequency vibration analysis of mechanical systems [45], real-time image processing [44], distributed sensor networks [46], and adaptive target acquisition and tracking systems [47], which require high throughput, and high degree of flexibility in the architecture. If the HDSP system is not a point design but need to serve a category of applications, an additional requirement is emerging: the system has to be configurable by engineers who understand the HDSP application domain in terms of the “language” of signal processing and hardware architecture, without the need to deal with low-level hardware and software details.

A specific example we mention here is a simplified version of an adaptive *automatic target recognition (ATR)* system, reported in [47]. The ATR system has to execute significantly different signal processing algorithms in different operational modes: target acquisition, long-range tracking, mid-range tracking, short-range tracking and aim-point selection. Performance requirements, changing power constraints and difference in the algorithms require that not only the signal flow, but the hardware architecture, too, is changing between the operational states. This means that the adaptive ATR needs to be implemented on a configurable HDSP platform, where configurability of the hardware architecture is supported by FPGA components [48]. For the designers, the adaptive ATR represents five different applications with different hardware and software architecture.

The design of any application on the configurable HDSP platform requires the design of the hardware and software configuration. These two aspects of the design are obviously interdependent due to performance requirements: one cannot design and build one independently from the other. This implies that integration of the two aspects should be a concern of the designer from early on in the process. Furthermore, an HDSP application is designed for a specific application environment. The environment imposes explicit requirements on the design (e.g. expected I/O bandwidth, throughput, data rates, power constraints.), against which the design must be validated. To summarize, the designer must carefully consider and balance the selected two aspects of the design, and create the final application accordingly.

In MIC, the key vehicle to facilitate the design process is the *model*. A model is a formal structure representing selected aspects of the engineering artifact and its environment. Models are suitable for formal reasoning about the properties of the system, and for deriving and generating a significant portion of the implementation. In the configurable HDSP platform example, the models encompass the following aspects:

- The *hardware architecture aspect*: represents the hardware components used (e.g. DSP-s, network switches, communication links), describes their structure and characterizes their properties (e.g. processing power, communication link throughput) on that level of abstraction, which is suitable for verifying the design and configuring the application.
- The *signal-flow aspect*: represents the software components of the application (DSP algorithm blocks, user interface components) the configurable components of run-time platform components (I/O channels, scheduler), describes their structure and characterizes their properties (Worst-Case Execution Time (WCET), IT latency, etc.)
- The *environment aspect*: represents the assumptions about the system's environment (e.g. sampling frequency required on each input channels, the number and types of targets expected, etc.)

When MIC is used to build applications on a configurable HDSP platform, modeling languages are designed first that allow capturing the models described above. Loosely speaking, a modeling language consists of a *collection of concepts* (e.g. DSP processor, DSP algorithm block, etc.) with *attributes* (e.g. WCET), a *set of composition rules* for building complex models (e.g. how to form DSP signal flow models from elementary processing blocks), a *concrete syntax* (textual or visual), and *semantics* that captures what a model means. For the HDSP example, we have defined a number of "small" modeling languages, with visual syntax that allowed constructing systems of this category [55]. Figure 1 shows example models built using these languages. On the top window, the signal flow model is shown as a collection of signal processing blocks and the data flows among them. On the bottom left, the hardware architecture model is shown, as hardware nodes (with communication ports) and hardware connection among them. On the bottom right, the model shows another aspect of the design: the allocation of a (software) signal processing block to a hardware resource: a node. Models of the hardware architecture are defined in terms of networks of processors with communication ports; signal flow models are defined in the form of block-oriented signal-flow diagrams, and models for software/hardware allocation in the form of diagrams that mix objects from the two previous models.

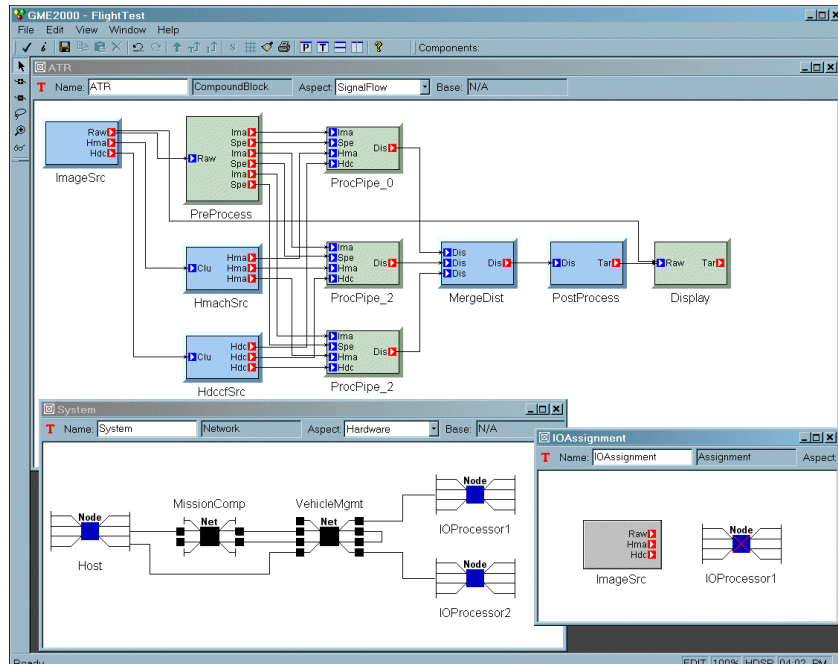


Figure 1: Example HDSP models

The signal-flow aspect and hardware architecture aspect of the design cannot be considered independently. The aspects interact, and design decisions made in one aspect have a significant impact on the other because of crosscutting design constraints. For example, latency between the input from the image sensors and an update in the target track depends on the selected processing algorithms, the structure of the signal flow, its allocation to a node of the hardware architecture and the characteristics of the processors and communication links involved. The overall performance of the design depends on the properties of the software: algorithms and architecture, the hardware: processors, networks, and architecture, and the assignment: the mapping of software components to hardware resources. The final, overall properties of the design emerge from the properties of and the interactions among the three aspects. Therefore, it is a key requirement for any complex modeling language that it must allow modeling from multiple, interacting points of view.

Models not only represent the system under design and its environment, but also used to *predict* characteristics of the design. In the HDSP application, the models can be used to configure a functional simulator (like Matlab's Simulink [3]) to verify the signal processing algorithms, or a performance simulator (like SPN [4]) to predict expected performance. However, the simulator's modeling language may be syntactically and/or semantically different from the one used in the domain modeling. The solution requires the *translation of models* between two different domains: the domain of the application-specific design models (e.g. HDSP) and the domain of the simulators (e.g. Simulink). *Model translation* is a common, fundamental ingredient of the model-based development process. Model translators implement syntactic and semantic mapping between domains – if such a mapping exists. Implementation of model translators is an important technology challenge in MIC.

Using models for analysis helps the designer verify that the product will work as expected —provided the models of the system are correctly translated into the analysis models. However, models can also be used in *system synthesis*; in the creation of the fully specified final, executable model of the product. The role of synthesis tools in the design process is the full or partial automation of selected phases of the design. For example, in the HDSP domain, the selection of signal flow components among alternative implementations and their allocation to hardware resources can be supported by a synthesis tool, which uses performance, resource and composability constraints in the search process [49].

After the application models are fully specified and verified, configurable components of the HDSP platform need to be configured. This step includes tasks such as generation of glue code for connecting the selected software components, generation of executable code from software models (like Matlab's Real-time Workshop) generation of structural VHDL from the hardware models specifying the hardware architecture, and generation of scheduling table for the static scheduler. *Generators* are model translators: they translate models into components of execution platforms: programming languages, executable models, and data structures used by components of the *run-time environment*.

The evolution and maintenance of systems built using the MIC approach is centered on models. As the bulk of the system is generated automatically from models (that are also analyzed and verified using design tools), one has to change the models and re-verify and re-generate the system. This approach works very well in the domain defined by the DSMLs used during the design. However, when the modeling language itself has to undergo changes (e.g. new concepts and/or new semantics have to be introduced), then the migration of existing models becomes a requirement.

To summarize, the model-integrated development of systems includes:

- *Modeling* of the system and its environment, from multiple, interacting aspects
- *Automated synthesis* of design models to accelerate the modeling process
- *Analysis* of the system using analysis/simulation tools, which may necessitate *model transformations*
- Generation of the executable system using generator tools (that map models into implementation domain artifacts).

Figure 2 below gives a high-level view of this process.

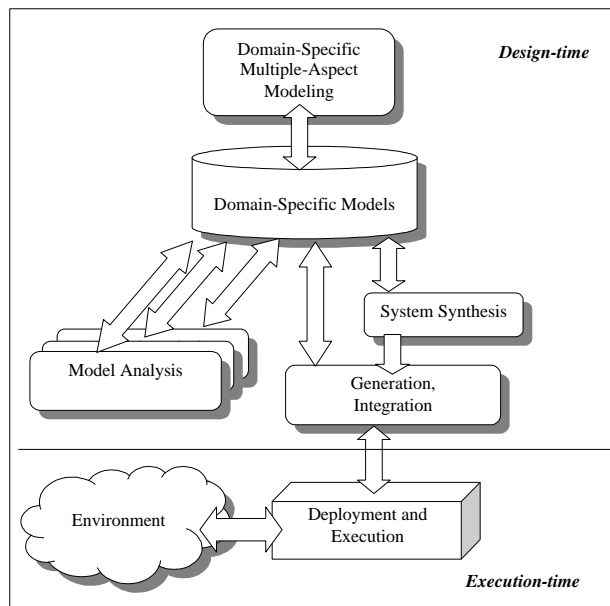


Figure 2: Notional diagram for the Model-Integrated development process

3. Modeling

Model-based system development uses DSMLs that can be textual or visual. “Programmers” of model-based systems are primarily modelers, whose domain knowledge allows them to construct models of (1) the application to be built and (2) the environment providing its context. These models are analyzed to verify required properties of the design, and used for generating components of the final application.

3.1 Domain-Specific Modeling Languages

Domain-specific Modeling Languages (DSML) are declarative, use domain-specific symbols, and have a restricted yet precise semantics. In order to define a DSML, one has to specify its (1) concrete syntax (C), (2) abstract syntax (A), and (3) semantic domain (S) and semantic and syntactic mappings (M_S , and M_C) [12]. The *concrete syntax* defines the specific (textual or graphical) notation used to express models, which may be graphical, textual or mixed. The abstract syntax defines the *concepts*, *relationships*, and *integrity constraints* available in the language. Thus, the abstract syntax determines all the (syntactically) correct “sentences” (in our case: models) that can be built. (It is important to note that the abstract syntax includes semantic elements as well. The integrity constraints, which define well-formedness rules for the models, are frequently called “static semantics”.) The S semantic domain is usually defined by means of some mathematical formalism in terms of which the meaning of the models is explained. The $M_C : A @ C$ mapping assigns syntactic constructs (graphical, textual or both) to the elements of the abstract syntax. The $M_S : A @ S$ semantic mapping relates syntactic concepts to those of the semantic domain.

Formally, a modeling language is a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S , and M_C):

$$L = \langle C, A, S, M_S, M_C \rangle$$

Any DSML, which is to be used in the development process of embedded systems, requires the precise specification of all five components of the language definition. In addition, rapid composition of DSMLs requires modularity, compositionality and tool support in the language definition itself; otherwise the high cost would prevent us to take advantage of domain-specific languages. Consequently, the languages, methods and tools we use for defining and composing DSMLs are *the* fundamental issues in model-based design. Since such languages are used for defining modeling languages, we call them *meta-languages* and the concrete, formal specifications of DSMLs *metamodels*.

3.1.1 Modeling Abstract Syntax

The specification of the abstract syntax of DSMLs requires a meta-language that can express concepts, relationships, and integrity constraints. In MIC, we adopted the UML class diagrams [13] and the Object Constraint Language (OCL) [15] as meta-language. This choice was made for the following practical reasons: (a) UML/OCL is an OMG standard that enjoys widespread use in the industry, (b) tools supporting UML are widely available, and (c) familiarity with UML class diagrams helps to mitigate the conceptual difficulty of the metamodeling language architecture. It is important to note that adopting UML for metamodeling does not imply any commitment to use UML as a domain modeling language, though it can certainly be used where appropriate.

To give an example, consider the signal-flow aspect of the HDSP application. Such domain can be modeled with hierarchical signal-flow diagrams (HSFD), widely used in signal processing, control engineering, and simulation. Informally, the abstract syntax of the HSFD can be described as follows:

1. HSFD models consist of processing blocks, and directed edges representing signal flows.
2. Each processing block has a set of input ports and a set of output ports that it uses for obtaining data to be processed, and sending the result of processing to downstream blocks.
3. Processing blocks can be either primitives or compounds. Primitive processing blocks are associated with an algorithm, compound processing blocks are HSFD models composed of primitive blocks and/or compound blocks, and have their own input and output ports.
4. Signal flow edges connect the ports of processing blocks as follows: an input port of a block can be connected to an input port of an immediate child block, an output port of a child block can be connected to the input port of a child block or the output port of the block. The direction of the dataflow is indicated by the direction of the connection.

The UML diagram on Figure 3 depicts the metamodel for the software aspect (SW) of the HSFD language. The SW class: the top-level construct in the language¹ represents the container of models of type `Block`. Blocks contain `Ports` that can be either `InputPorts` or `OutputPorts`. Blocks are subclassed into `CompoundBlocks` or `PrimitiveBlocks`. Through inheritance, they can both contain ports.

¹ Note that the top-level model is equivalent to the “sentence” (S) non-terminal in textual languages represented using, for instance, a context-free grammar.

Compounds can also contain other blocks: that can be either compounds or primitives. Furthermore, compounds also contain `Dataflow` objects, which associate ports to ports. The well-formedness rules are expressed by the OCL expressions. On the right side, a few OCL constraints are presented illustrating correctness rules (that cannot be expressed with the UML class diagram only).

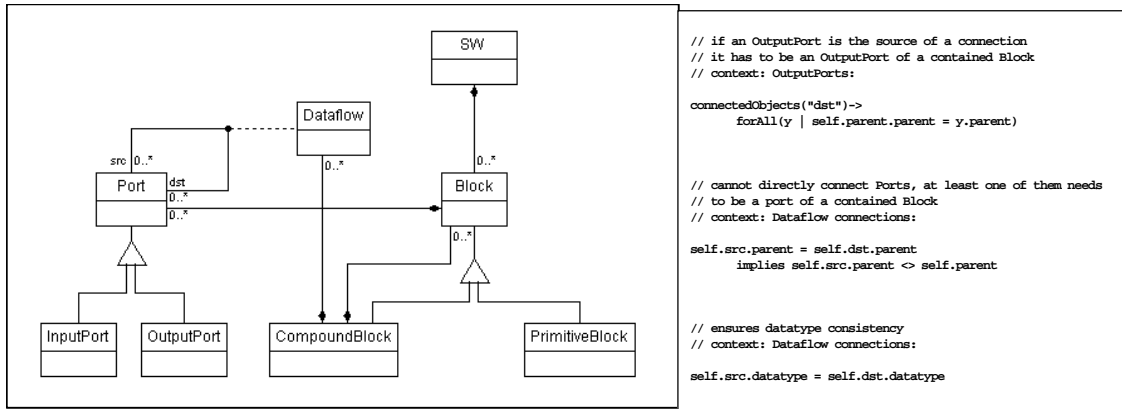


Figure 3: HSFd Metamodel with OCL constraints

It is clear that the UML/OCL meta-language is another example for DSMLs. As such, it also needs to be specified precisely, otherwise ambiguities in its specification would propagate to the DSMLs. In UML, the abstract syntax of object diagrams is defined with the UML/OCL meta-language, following a process called *meta-circular specification*. Accordingly, the specification of the abstract syntax of UML class diagrams is the meta-metamodel. In the current version of UML, the meta-circular specification is not complete (e.g. OCL is an “external” language.) These inconsistencies will be resolved in the expected UML 2 release [50].

3.1.2 Modeling Concrete Syntax and Syntactic Mapping

The concrete syntax might be considered as a *mapping* of the abstract syntax onto a specific domain of rendering. While the purpose of the abstract syntax is to define the data structures that can represent our models, the concrete syntax captures how they can be rendered for human interaction or for machine-to-machine communication. In Figure 4 we show two common examples for assigning concrete syntax to HSFd. A simple block diagram notation is used to represent a simple HSFd model in Figure 4 (a). Concepts and relations defined in the abstract syntax are assigned to visual constructs: *blocks*, *lines*, *connections* and *alphanumeric symbols*. This assignment can be done formally using e.g. UML class diagram notation (see e.g. [52]) or informally. A frequently used alternative for concrete syntax is XML [51]. Figure 4 (b) shows the same HSFd model in XML format. The graphical form of the model is related to the model in XML form as follows. From the common metamodel one can derive the visual grammar of a modeling language, as well as the schema (DTD, in XML parlance) for the XML file. Every element in the graphical language will then correspond to an element in the XML file. For instance, the IP0 input port in the graphical notation corresponds to the element `<inputport name=IP0>` in the XML file.

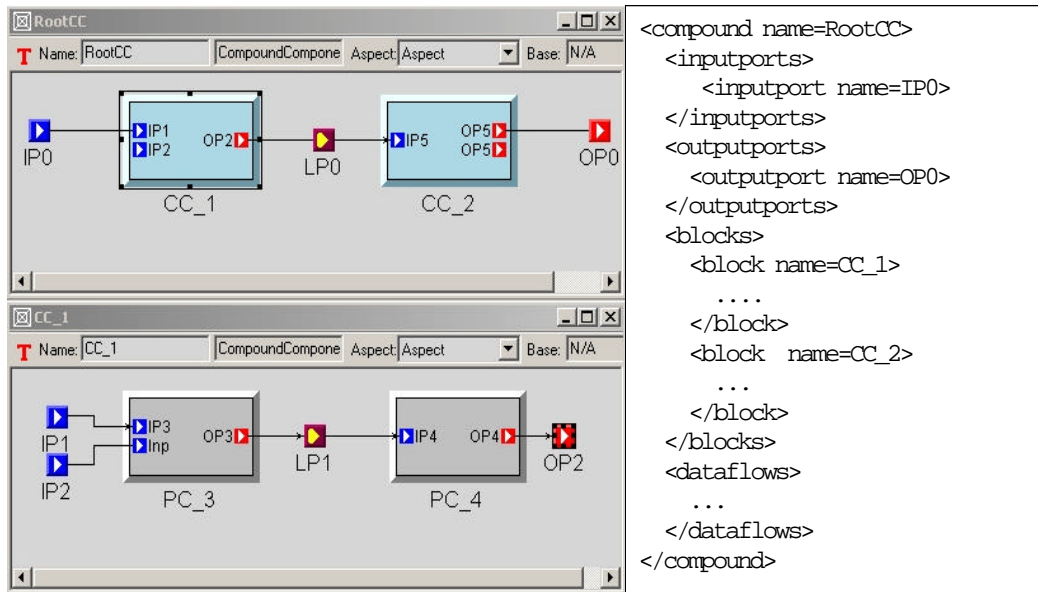


Figure 4: (a) Graphical syntax for HFSM (b) XML syntax for HFSM

Making the distinction between the concrete and abstract syntax has a profound impact on the technology of model building: editing and manipulation. The modeler interacts with the concrete syntax: creates and modifies structures using the “primitive” concepts of the concrete syntax. If the DSML is graphical, then the modeler uses direct manipulation on the graphical objects. However, these graphical objects are just a rendering of the underlying objects of the abstract syntax, and thus the effects of changes on the graphical objects shall result in changes on the underlying objects. We argue that DSMLs, especially if they are graphical, should be accompanied with appropriate tools to support these manipulations, such that direct manipulation results in immediate changes in the structure and properties of underlying objects.

3.1.3. Modeling Semantic Domain and Semantic Mapping

The semantic domain and semantic mapping defines the semantics of a DSML. The role of semantics is to describe the properties (meaning) of models that we want to create using the modeling language. Naturally, models might have different interesting properties; therefore DSMLs might have different semantics associated with them. For example, *structural* and *behavioral* semantics are frequently associated with DSMLs. The structural semantics of a modeling language describes the meaning of the models in terms of their composition: the possible configuration of components and relationships among them. Accordingly, the structural semantics can be formally represented by set-relational mathematics. The behavioral semantics describes the evolution of the state of the modeled artifact along some time model. Hence, behavioral semantics is formally modeled by mathematical structures representing some form of dynamics, such as Finite State Machines (FSM) [37] or Hybrid Systems [53].

Although specification of semantics is commonly done informally using English text (see e.g. the specification of UML 1.3 [13]), the desirable solution is explicit, formal specification. There are two frequently used methods for specifying semantics: the *metamodeling approach* and the *translational approach*.

- In the *metamodeling approach* (see e.g. [40]), the semantics is defined by a meta-language that already has a well-defined semantics. For example, the UML/OCL meta-language that we use for defining the abstract syntax of a DSML has a structural meaning: it describes the possible components and structure of valid, syntactically correct domain models. The semantics of this meta-language can be represented by set theory, i.e. by using a formal language, which enables the precise definition of sets and relations on sets. Candidates for such languages are Z [41] or algebraic specification languages like Larch [42]. By developing the formal semantics for UML class diagrams and OCL - let's say in Z - the metamodel

of DSMLs specifies not only their abstract syntax, but their structural semantics² as well. An early example for this direction is Bourdeau and Cheng’s work on formal semantics of object diagrams [43].

- The *translational* approach specifies semantics via specifying the mapping between a DSML and another modeling language with well-defined semantics. For example, we may want to adopt a synchronous dataflow (SDF) behavioral semantics as defined in [23] for HSFD. This can be accomplished by defining the mapping between HSFD and SDF.

Although, we can assign different semantics (such as structural and behavioral) to the same abstract syntax, we need to be aware that they are not independent. Formulation of the well-formedness rules in the abstract syntax requires a thorough understanding of the S semantic domain so as to ensure that the semantic mapping of each well-formed model leads to a consistent semantic model. (This strong interrelation is the reason of considering well-formedness rules as “static semantics”.) For example, if HSFD has synchronous dataflow (SDF) semantics as defined in [23], one of the well-formedness rules must prohibit the connection of two different output ports to the same input port (the constraint expression is in OCL):

$$\text{Self.InputPorts()} \rightarrow \text{forAll}(ip \mid ip.\text{src()} \rightarrow \text{forAll}(x1, x2 \mid x1 = x2))$$

However, if we use for HSFD dynamic dataflow (DDF) semantics [23], the same constraint may be omitted.

In our experience, DSMLs that are designed for modeling real-life embedded systems tend to become complex. In these domains, crucial requirements for robust, domain-specific modeling include careful formulation of metamodels, keeping them consistent with the associated semantic domains, and checking whether the created models are well-formed.

3.2 Composition of metamodels and models

One justification for using the model-based approach to system development is that models offer better ways to manage complexity, than currently used procedural or object-oriented languages. Composition in model-based design appears on two levels (1) composition of DSMLs by means of metamodel composition and (2) composition of models in the context of specific DSMLs. In this section we discuss the composition of domain-specific modeling languages and the composition of models.

3.2.1 Metamodel Composition

Compositional construction of DSMLs requires the $L_n = L_1 \parallel L_2 \dots \parallel L_k$ composition of metamodels from component DSMLs. While the composition of orthogonal (independent) sub-languages is a simple task, construction of DSMLs from non-orthogonal sub-languages is a complex problem. Non-orthogonality means that component DSMLs share concepts and well formed-ness rules span across the individual modeling aspects.

Composition of DSMLs from sub-languages is particularly important in embedded systems, which frequently require many modeling aspects. Since the current version of our metamodeling language bases UML and OCL does not support modular composition of metamodels³ we introduced new facilities in our metamodeling environment, which leave the component metamodels intact and creates metamodels that are further composable [19]. The composition is accomplished by using three new operators for combining metamodels (see Table 1). Two of these operators are a specialization of the inheritance relationship of UML. The principle here is that that language designer specifies metamodels, and then composes and extends them to create new metamodels using these operators. The operators allow specific composition operations on the (read-only) base classes to derive new classes. The first operator: “Equivalence” asserts that two classes (in different class diagrams) are to be considered identical, and in the composed metamodel they are merged. The second operator: “Implementation Inheritance” asserts that the derived classes will inherit the attributes of the base class and all those associations where the base class

² The condition for this statement is that the semantics of the meta-language is compositional.

³ The upcoming version of UML 2 is expected to include explicit support for metamodel composition

plays the role of a container. The third operator: “Interface Inheritance” states that the derived classes will inherit only those associations where the parent class is not a container⁴. Application of these operators generates a new metamodel, which conforms to the underlying semantics of UML. Decomposition of the UML inheritance operation allows finer control over metamodel composition (details are discussed in [19]).

Unfortunately, metamodel composition is not complete by executing the composition operators. If the component metamodels are not orthogonal, it is possible that the resulting metamodel is not consistent, which means that conflicting well-formedness rules are created during the composition process. This means that the metamodeling toolset needs to be extended with a validation tool, which checks the consistency of the well-formedness rules.




Operator	Symbol	Informal semantics
Equivalence		Complete equivalence of two classes
Implementation Inheritance		Child inherits all of the parent’s attributes and those containment associations where parent functions as container.
Interface Inheritance		Child inherits all associations except containment associations where parent functions as container.

Table 1: Metamodel composition operators

Figure 5 illustrates how the metamodel composition can be used to compose a HSFd modeling language from a (non-hierarchical) signal flow language and the abstract definition of hierarchy. The upper left diagram defines a signal flow modeling language, which has `Diagrams` that contain `Operators` (but the diagrams cannot contain other diagrams). The upper right metamodel defines a language with `LeafBlocks` and `Containers`, the latter containing leafs and other containers. The diagram at the bottom shows a composed metamodel, which introduces `CompoundBlocks` and `PrimitiveBlocks`. A compound is a `Container` (from the definition of hierarchy) but it is also a diagram. Note that for the latter the implementation inheritance operator is used to carry over everything that is contained within the diagram. A primitive is a `Leafblock` (from the hierarchy) but it is also an `Operator` (from the signal flow). The resulting, composed meta-model allows compounds containing other compounds and primitives, thus it defines a fully hierarchical modeling language.

Note that in this example, the semantics of models did not change: everything is still a signal flow diagram; we merely allowed the arbitrary hierarchical nesting of diagrams. The reason for this is that the hierarchy is an abstract structuring principle and does not have a run-time semantics on its own. When models are assigned semantics through translation into a run-time environment (as discussed below), the hierarchy is “compiled away”, and we execute a flat signal flow graph, with homogeneous components and behavior. This is in contrast with the next example.

⁴ Note that the composition of the two (partial) inheritance operator yields the same result as the original inheritance operator of UML.

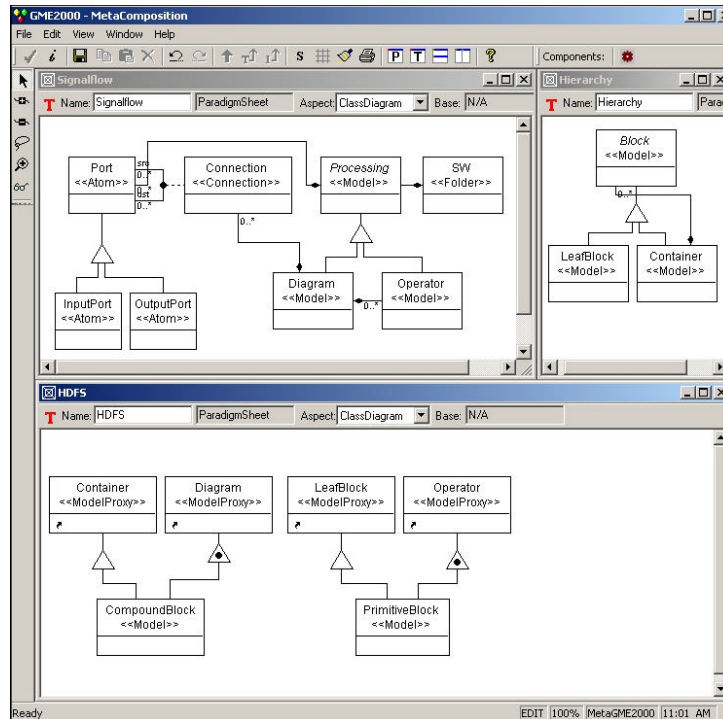


Figure 5: Metamodel composition: Signal flow + Hierarchy

Figure 6 shows another example for the composition of metamodels. The metamodel in the upper part defines a “SignalFlow” modeling language (essentially the same as the HSFd language defined above). The metamodel on the lower left defines a Finite State Machine (FSM) modeling language, consisting of states and transitions. The lower right metamodel defines a composition of the two: it introduces a new type of model: `FSMNode`, which is a new kind of `Primitive` that contains a finite state machine specifying its implementation (but otherwise it works just like a `Primitive`, has input and output signals, etc). This is expressed by the (unrestricted) inheritance between `Primitive` and `FSMNode`. However, we do not want a `State` to contain an `FSMNode`. This is expressed by the implementation inheritance between `State` and `FSMNode`. Furthermore, we want to make selected `InputSignals` and `OutputSignals` of any `FSMNode` to be mapped to certain `States` it contains using connections. (This could mean, for example, that the data values associated with those signals are accessible from the implementation associated with the given `State`.) This is expressed by the new `SignalMap` association class allowing the connecting of `FSMNodes` to `Signals`.

Note that, in this example, metamodel composition has a profound impact on semantics. The dynamic semantics of signal flow blocks and finite state machines is different. When we introduce the `FSMNode` as a variation of the `Primitive` concept, we —implicitly— allow (compound) models that contain both FSMs and signal flow blocks: primitives and compounds. The two kinds of blocks follow different models of computation, and their composition requires the precise specification of semantics for the composed model. The paper [56] describes a framework for comparing and [57] shows examples how to compose models with different semantics.

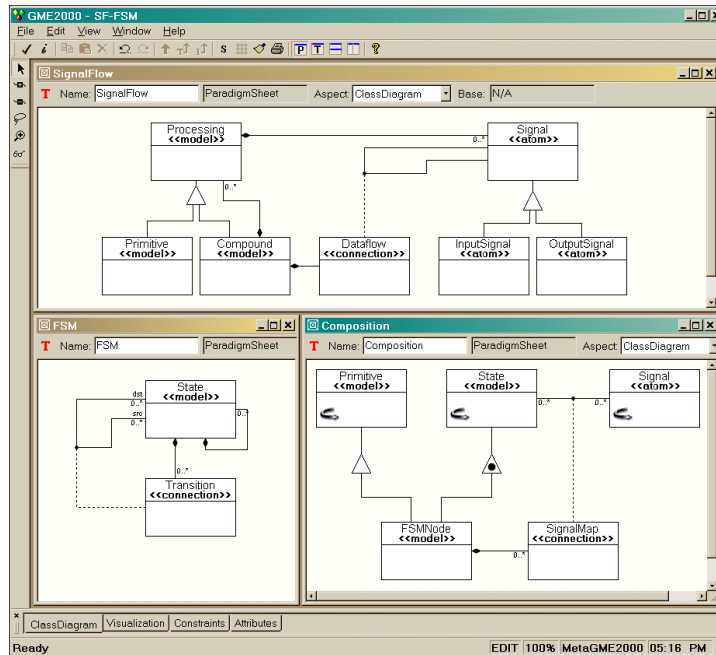


Figure 6: Metamodel composition: Signal flow and Finite State Machine models

Up to this point we have discussed the composition of DSMLs from the point of view of their abstract syntax. To derive an integrated semantics for the composed modeling language is a complex problem, which requires careful analysis. In simpler cases, for example composing a modeling language for Hierarchical Signal Flow Diagrams, the semantic domain describing behavioral semantics does not change. Therefore semantics can be defined by the straightforward application of the translational approach. However, if we intend to compose modeling language for Finite State Machines (FSM) and continuous dynamics, the integrated behavioral semantics leads to the theory hybrid systems [53], with qualitatively different characteristics from its sub-languages.

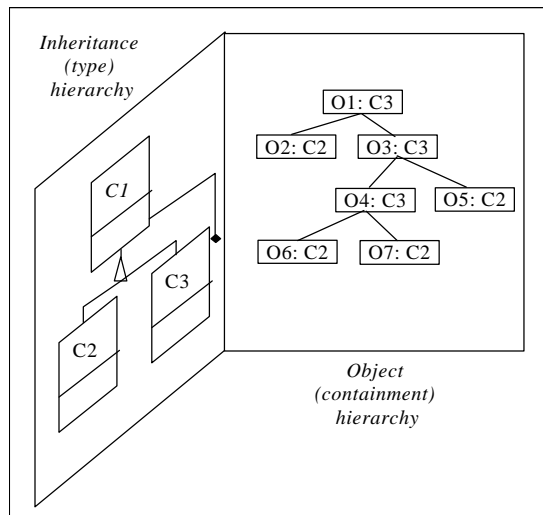


Figure 7: Abstraction in two hierarchies: inheritance and containment

3.2.2 Model Composition

Composition of models in a DSML is an essential task, and thus a number of techniques must be made available for the modeler. The abstract syntax of a DSML defines what composition techniques are available in the language. However, these composition techniques are always specific to that language, so it

is very hard to make general statements about composition. On the other hand, there are a number of techniques that are relevant across many application domains.

1. *Abstraction* is arguably the most powerful technique in modeling. Here, we define it as the capability for representing systems on different levels of detail, simultaneously. A DSML should allow creating such models, but also their seamless *vertical* composition: i.e. a higher-level, more abstract model should always be compatible with a lower-level, less abstract model, and substituting a lower-level model with a higher-level one must be allowed. The most common way to support abstraction is hierarchy: both in the part-whole and general-special sense. Thus, models built using the abstraction technique are usually organized into two, distinct trees: one tree for the part-whole hierarchy, and another one for type hierarchy. See Figure 7.
2. *Modularization* is an implementation technique, which helps not only in the construction, but also in supporting abstraction in the practice. To model a complex system, one needs to break it down into self-consistent entities: modules, and models expressed in a DSML should form modules and should be composable. *Composability* means that the composition of two or more modules should also be a module itself. In the HSDF example, the modules are the processing blocks, and compound blocks can contain primitive blocks or other compound blocks. Note that a compound block also serves as the vehicle for abstraction: if it is considered as a “black-box” it hides the details of its internal implementation, thus providing an actual implementation for abstraction.
3. *Interfaces and ported components* (a.k.a. module interconnection language) extend the simple part-whole hierarchy concept with modules that have distinguished components for connecting them to other modules. Composition through connecting ports of objects carries a domain-specific semantics. In the HSFD example we have used it to represent dataflow streams, but other semantics are also possible. For instance, in a DSML modeling biochemical processes on the cellular level the “ports” and “flows” may represent “receptacles” and “chemical interactions” among processes.
4. *Multiple aspects* allow controlling the complexity by restricting the information presented to (and edited by) the modeler. Conceptually, it is the same technique known from the field of databases: one can define specific views on the data in order to reduce what is presented and processed. Note that the same approach can be used in two different modes: at model creation time the modeler can edit the models from different aspects, while at model viewing or interpretation time the models can be “processed” from different aspects. Having support for multiple aspects in a modeling environment necessitates sophisticated mechanisms for view and consistency control, as the aspects can be dependent on each other and consistency across aspects can be maintained.
5. *References* reduce complexity by allowing linkage across levels and sub-trees of a part-whole hierarchy. Without references, using only the part-whole and ported objects techniques it is very hard to model non-local interactions across components that are located far away in the hierarchy tree. Without references the modeler is forced to introduce extra ports on objects, to replicate them in the tree up to the level first common ancestor object, and to establish all required intermediate connections. With references, the solution is trivial: the reference cuts across the tree and establishes a direct link between two, distant objects. See Figure 8. Although references provide a powerful technique, some caution is advised; when the target of the reference is removed, the reference itself becomes invalid.

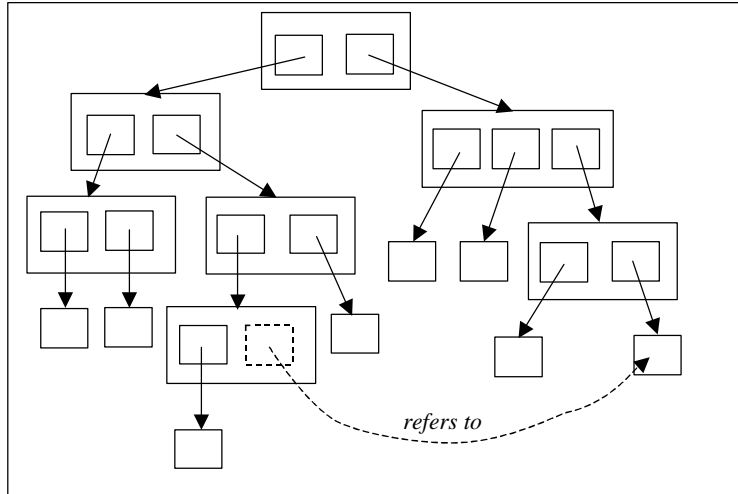


Figure 8: The concept of reference

These modeling techniques support well various engineering domains in practice. While most of them can be defined in the abstract, the multiple aspect modeling has a very concrete grounding in systems engineering. The art and science of systems engineering is perhaps best defined as the engineering of cross-cutting concerns in large-scale systems. This is precisely what multiple-aspect modeling addresses: each aspect describes the system from the viewpoint of a particular concern (in systems engineering: a discipline), and the key issue is how these aspects interact with each other.

Note that the model composition techniques correspond to specific idioms used in the metamodels. As the metamodel defines the modeling language, if a metamodel instantiates a specific idiom for composition, the modeling language defined will support that composition technique. In the table below we indicate the corresponding metamodel construct that enables the specific technique.

Technique		Metamodel idiom
Abstraction	Inheritance	(Nothing special, enabled by default)
	Part-whole	Containment association between classes
Modularization/Composability		(Nothing special, enabled by default)
Interfaces and ports		<<port>> stereotype for a class
Multiple aspects		Extra classes with <<aspect>> stereotype and containment within an <<aspect>> class.
References		<<reference>> stereotype for a class

Table 2: Model composition techniques and their metamodel idioms

One example illustrating the power and necessity of multiple-aspect modeling comes from previous work done on the International Space Station program [39]. The task was to build a modeling and analysis environment for fault detection, isolation, and recovery (FDIR). Models had to be created that captured the behavior of the system under normal operation and in the presence of faults. The system was designed by a large number of engineers, working on different subsystems and in different disciplines. There were four primary disciplines: electrical, hydraulic, information, and pneumatic, and many components had to be considered from multiple disciplines. Faults in components propagated in the system not only to other components but also to other disciplines, e.g. an electrical system fault leading to a breakdown in the hydraulic system. Using the multiple aspect technique, the modeling task could be organized as follows: each discipline was mapped into an aspect, and a further aspect was added where cross-discipline interactions were modeled. The various aspects were related and linked (through a shared, underlying data structure), and model changes in one aspect were automatically propagated to other aspects. While model creation and editing was done on a per-aspect basis (often by domain engineers), the analysis was performed on the integrated models, where all the interactions were fully traceable. To summarize, the

multiple aspect modeling technique offers not only a technique for managing model complexity through view control, but it also explicitly supports modeling activities that cut across domains.

3.3 Tools for Domain-Specific Modeling

The core ingredient in a model-based system development process is the DSML. However, it is typically very expensive to develop a new DSML for every application domain. The expense stems from many factors, including the cost of defining a new language, the cost of training the modelers to use it, and the cost of the tools that support the language. Obviously, these costs can be reduced in different ways: for instance, if the modelers are already familiar with the concepts of the language because they are experts in the engineering domain, or if the language definition and tool development can be made simpler and more effective. This second aspect could be best addressed by a technique, which allows (relatively) easy language definition and tool development: metamodeling and meta-programmability.

Metamodeling is the process of defining a domain-specific modeling language. Note that the idea is not unlike how programming languages are created: when a language is designed its concrete and abstract syntax and semantics is developed and precisely documented. Metamodeling is the same activity, which results in a DSML.

We argue that metamodeling should be supported by tools that produce new, domain-specific tools to be used by the DSML modeler. One such specific tool could be a meta-programmable modeling tool, which is “programmed” via explicitly represented metamodels. See Figure 9 for an illustration of this process. The definer of the DSML, i.e. the metamodeler, creates metamodels: descriptions of the syntax and semantics of the DSML, and these metamodels are then employed in the configuration process. One crucial observation here is that *the same meta-programmable modeling tool can be used to create the metamodels themselves*: after all, the language of metamodels is just another DSML.

The semantics of a metamodel is defined as the mapping between the abstract syntax of the domain models and some model of computation. There are at least two types of “computation” that are influenced by the metamodel: (1) the computations performed during model editing and (2) model translation. Model editing takes place when the domain models are created and modified, while model translation occurs when domain models are transformed into lower-level executable models. The definition of semantics applies in both cases: the metamodel controls the lower-level model of computation.

Formally, a metamodel can be assigned an interpretation in two ways:

1. The static semantics of a metamodel defines the well-formedness and static correctness rules of domain models. This means that there exists a decision procedure, which tells whether a domain model is correct with respect to the metamodel or not.
2. The dynamic semantics of a metamodel defines how to interpret a domain model, i.e. what is the semantic mapping between the abstract syntax of a domain model (also captured in the metamodel) and some model of computation.

One can actualize this interpretation of a metamodel as follows:

1. A meta-programmable, generic modeling environment can be “programmed” by a metamodel, such that the environment supports and only allows the creation of models that comply with the static semantics of the modeling language, as prescribed by the metamodel.
2. A meta-programmable translator framework can be “programmed” by a metamodel such that the instances of the framework: the translators are capable of mapping (well-formed) domain models into models of computation, supported by some run-time system.

Figure 9 illustrates the steps of this process.

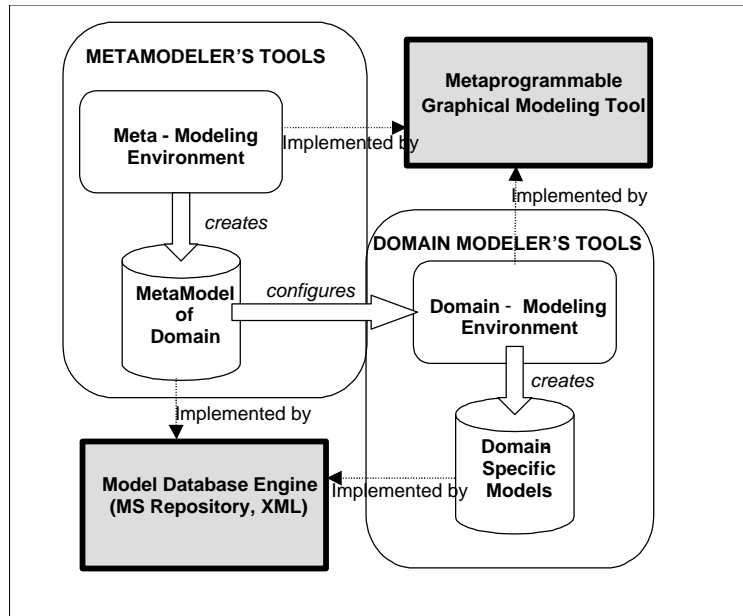


Figure 9: Connecting metamodeling and domain modeling

The technique of generating modeling environments and model interpretation/translation tools from metamodels offers a powerful approach to building customized, domain-specific problem solving environments. There is an economic argument for using this approach: instead of a point solution, one can create a product-line solution by developing first the tools to produce the products, and the cost of the initial tool development can be amortized over all the products in the product-line. Naturally, the key concern here is how to develop the meta-programmable tools: modeling environments and translator frameworks that can be used in different product lines.

A meta-programmable modeling environment can be architected as a componentized system, with a general-purpose editing engine, and separate viewer-controller GUI, and a configurable persistence engine. Figure 10 below shows the architecture of such an example environment: Generic Modeling Environment (GME). GME is a direct-manipulation editor that allows the modeler to visually manipulate underlying model data structures.

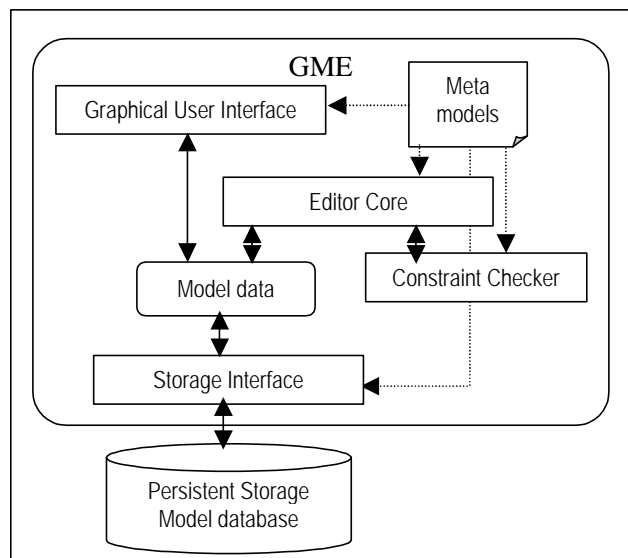


Figure 10: GME Architecture

In GME, almost all components are configured by the metamodels. For example, the metamodels determine:

- In the “Storage Interface”, the *database schema* to be used in the persistence engine,
- In the “Editor Core”, the *legal editing operations* on the domain models: composition, attribute values, etc., and
- In the “Graphical User Interface”, the *visualization* and the (legal) *user interaction techniques* used when editing the models.

To support flexibility and to provide reusable software architecture the metamodels are represented as static data structures, which govern the behavior of the individual components. The components are “generic”, but during their operation they “consult” the in-memory, metamodel data structures and behave accordingly. For instance, the persistence engine is using a single schema, with high-level concepts like “Model” (container), “Connection” (association), “Atom” (primitive entity), “Set” (a group of specific objects within a container), “Reference” (a generalized pointer to another object), and “Attribute” (property). However, each instance of these objects is tagged with a “type tag”, which relates that object to a metamodel element. The tags are stored in the persistent store together with the models. At model editing time, the tags are used to guide the editing engine: for example, to verify that a particular composition is legal or not. This interpretative nature of the modeling environment introduces some overhead, but in practical situations it is negligible.

Metamodeling is supported by UML class diagrams and OCL expressions with GME-specific stereotypes as the metamodeling language. The semantics of the metamodels is defined by their mapping onto the GME “model of computation” (MOC). GME has a number of built-in abstractions that form a (very special) model of computation: the abstractions include the concepts mentioned above: “Model”, “Connection”, “Atom”, “Set”, “Reference”, and “Attribute”, and the operations in the GME MOC include creation and manipulation of the above objects, as directly supported by the GME editing engine. A general purpose UML class diagram with OCL expressions does not carry enough information for mapping it precisely onto the GME MOC; there are many mappings possible. Therefore, the UML class diagrams must be embellished with GME-specific stereotypes that govern how exactly the mapping is to be done. For instance, the stereotype markers help to decide whether a class is a GME “Model” or a “Set”. The metamodeler is expected to provide these stereotypes, and the GME metamodel translator checks whether they comply with the well-formedness rules for metamodels.

The process of metamodeling in GME is illustrated on Figure 11. On the top of the figure, a UML class diagram shows a metamodel for the HSFD example. In the middle, the corresponding GME-style metamodel is shown, which is identical to the pure UML metamodel except the GME-specific stereotypes. Internally, GME creates and operates upon objects of type Connection, Model, Atom, etc. These objects constitute the model being visualized and edited (shown at the bottom of the figure). Each one of these objects is related to a metamodel object, e.g. the left-most atom object “In” is related to the `InputPort` object in the metamodel (indicated by the line between them).

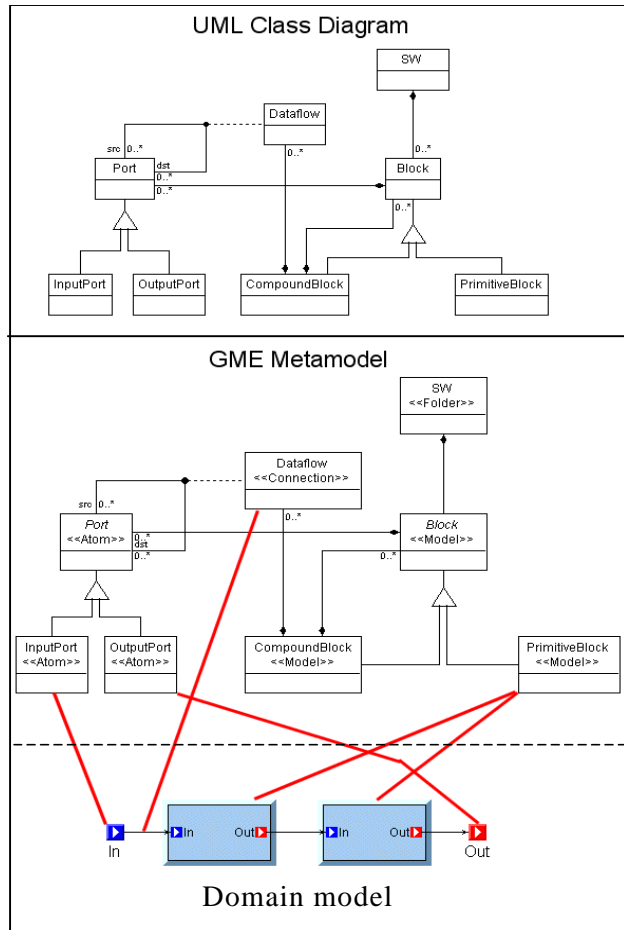


Figure 11: Metamodeling Process

4. Model synthesis and generative modeling

Modeling and model-based approaches [20][21] already do and will play a central role in embedded software and system development. The fundamental promise of model-based approaches is that experimental system verification will be largely replaced by model-based verification. However, even by taking advantage DSMLs and advanced model-editors, creating high-fidelity models is expensive. Therefore, development of efficient methods for supporting the model-building process is an important research goal. Below, we briefly discuss two important approaches: compositional modeling and model synthesis.

4.1 Compositional modeling

Building complex models by composing components $M_n = M_1 || M_2 \dots || M_k$ is a common, highly desirable technique for efficient modeling. In bottom-up composition, simpler components are integrated to obtain more complex components. The condition for *composability* in bottom-up composition is that if a property P_k holds for component M_k , this property will be preserved after integrating M_k , with other components. Unfortunately, in embedded systems, many physical properties (such as time dependent properties) are not composable [8]. Therefore DSML-s, which are formal enough to be analyzable and analysis tools, which can verify essential properties of the composed designs are crucial in model-based system/software development.

4.2 Model Synthesis

Model synthesis in the compositional modeling framework can be formulated as a search problem: given a set of $\{M_1, M_2, \dots, M_k\}$ model components (which may represent different views of the system and may be parameterized), and a set of composition operators, how to select an $M_d = M_i || M_j \dots || M_k$ design (with the required set of parameters) such that a set of $\{P_{1d}, P_{2d}, \dots, P_{kd}\}$ properties for M_d are satisfied? Fully automated synthesis is an extremely hard problem both conceptually and computationally. However, by narrowing the scope of the synthesis task, we can formulate solvable problems. For example, by using design patterns and introducing alternative design choices for component templates in generic designs we can construct a design space using hierarchically layered alternatives. This approach is quite natural in top-down engineering design processes, which makes the construction of design space using hierarchically layered alternatives relatively simple [22].

To enable the representation of design spaces, we need to expand DSMLs with the ability to represent design alternatives explicitly. As an example, Figure 12 shows the metamodel of SF extended with the concept of Alternatives. We selected the abstract Base concept for Alternative implementation, and introduced a containment relationship to enable hierarchical composition. An Alternative, in the composed SF metamodel context, can now be defined as a processing block with rigorously defined interface, which contains two or more (notice the cardinality of the containment relation highlighted in the figure) alternative *implementations*. The implementations can be Compounds, Primitives, or other Alternatives, with matching interfaces. As we mentioned above, composition of a design requires finding implementation alternatives, which satisfy a set of design properties. The complexity of this task largely depends on the computability of selected design properties from the component properties. A detailed analysis of this problem and a constraint-based design-space pruning technique is described in [23].

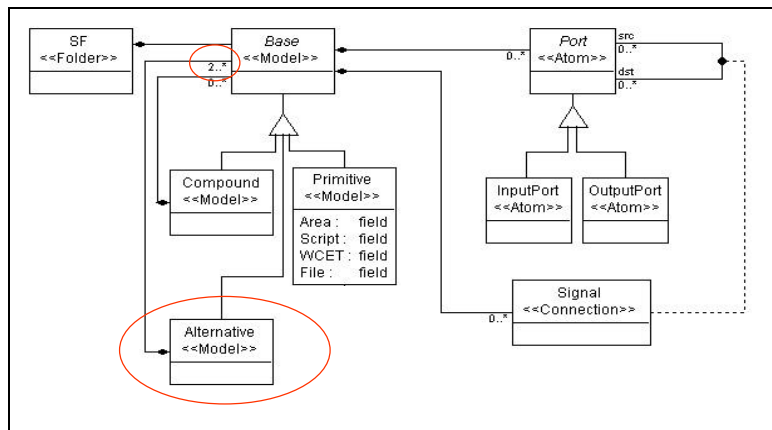


Figure 12: Metamodel extended with the Alternative construct

5. Model-based generators

The key factor that distinguishes the model-based approach from other techniques is that models are used as input to *generators* that translate them into other artifacts used in analysis and at run-time in the application. Thus, in the model-based development process modeling, analysis and system synthesis are tightly integrated activities that cannot be separated. This integration is essential to success: as all changes originate in the modeling environment, and the same, consistent models are used in analysis and synthesis, the final product is verified through analysis, and it is much easier to maintain and evolve.

5.1 The role of generators

The integration among the participating elements of the model-based development process happens through generators (a.k.a. “translators” or “model interpreters”) that transform models into other forms. There are two major applications of generators:

1. To translate models into the input language of analysis tools and to translate the analysis results back into the modeling language, and
2. To translate models executable into code, static data-structures, component configurations, customized generic components, etc, which form the executable system running on some integration platform (OS, component integration framework, etc).

In the most general sense, generators implement a semantic mapping between domain models and another domain: an analysis tool, or a run-time environment.

Analysis tools often have a different modeling language and a different model of computation than the domain modeling language. For instance, in the HDSP domain the language is that of dataflow networks. However, a performance analysis tool, which can do throughput analysis on these networks, accepts models in the form of Stochastic Petri Nets (SPN) [4]. In this case, a generator would map the HDSP dataflow networks into an equivalent SPN what the analysis tool can use to calculate performance predictions from the models of the system. Note that to perform this translation, the translator has to have the detailed knowledge of the model of computation used on the execution platform, and how it can be expressed in the form of SPN-s. Obviously, there is no guarantee that there exists a mapping between the domain modeling language (with a dynamic semantics), and the input language of an analysis tool, in general. However, if such a mapping can be developed, it can be realized in a generator, and the analysis tool may provide major benefits for the developer. When analysis tools are used, the results of the analysis must be made available for the modeler. A “reverse generator” could be used for this purpose, and the analysis results can be used to back-annotate the models in the modeling environment.

5.2 Techniques for building generators

Generators for model-based systems can be implemented in a variety of ways. Below, we discuss three major techniques.

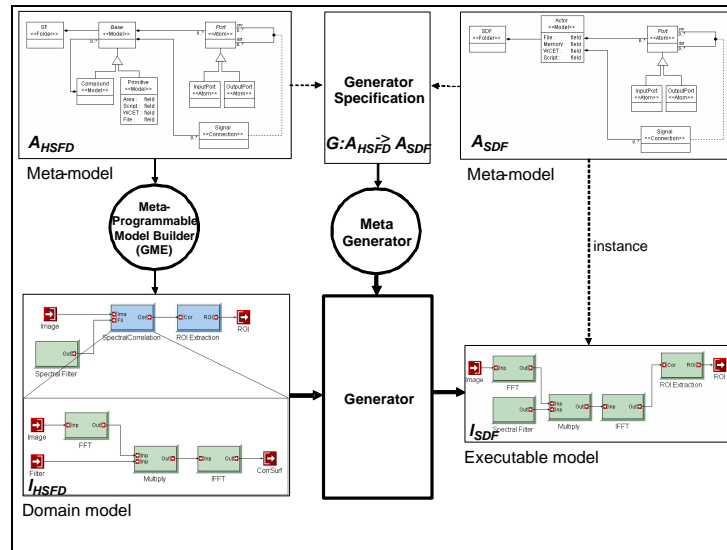


Figure 13: Model-based Generators and Meta-generators

5.2.1 Direct implementation

A generator that transforms models is similar to a compiler, although its task is more specific and somewhat simpler. While a compiler maps the abstract syntax of a programming language into high-performance executable code running on the hardware architecture, a generator maps the abstract syntax of the input language into the abstract syntax of the target language, where the target has well-defined execution semantics, with high-level “instructions”. If one omits the technical details of creating the output product in some physical form (e.g. text files), the main task of a generator is reduced to creating a “target tree” from an “input tree”. The “input tree” is the data structure that corresponds to the input abstract syntax tree of compilers, and the “target tree” corresponds to the “output tree” of compilers from which the code is directly “printed”⁵. Naturally, in the simplest of cases the output product can be directly produced from the input tree.

In the most general form, a generator performs the following operations.

1. Construct the input tree. This step is implicit in model-based systems, as the models *are* the input tree, and the modeling environment directly manipulates a tree-like representation.
2. Traverse the input tree, possibly in multiple passes, and construct an output tree. In this step the generator visits various the objects in the input tree, recognize patterns of objects, instantiate portions of the target tree, calculate attributes of output objects from attributes of input objects, etc.
3. “Print out” the product. This step creates the result of the generation in the required form: a network of objects, a text file, a sequence of commands issued to a hardware device, etc.

Following the above approach, a generator is straightforward to construct: after designing the input and output data structures (which are determined by the metamodels of the input and target languages already), one has to design the appropriate code sequences for traversal and target object construction. The implementation can follow an object-oriented approach: the traversal code can be embedded as methods of the input objects, and by directly coding the traversals and the target construction one can easily realize the generation algorithms. Similarly, the output “printing” can be embedded as methods of the target objects, and by programming the traversal of the *output* one can realize the output-producing algorithms. This direct implementation is simple and works well for situations where the transformations are easy to capture in a procedural form.

⁵ We use the term “tree” here, although in these data structures are graphs in the most general case. However, even in those cases, a spanning tree of the graph can be found, which “dominates” the structure.

5.2.2 Pattern-based approach

The scheme described above can also be implemented in a more structured way, by using the Visitor design pattern [26]. The main task of a generator involves the *traversal* of the input tree and *taking actions* at specific points during the traversal. This is clearly in the purview of the Visitor pattern, which offers a common solution for coding the above generic algorithm. In this pattern, a *visitor* object implements the actions to be performed at various nodes in the tree, while the tree nodes contain code which *accepts* the visitor object, calls the appropriate, node-specific operation on it (while passing itself to the operation as a parameter). The Visitor pattern allows the concise and maintainable implementation of generators, both for the transformation and the printing phases.

While the implementation of a generator following the Visitor pattern is straightforward, it can be significantly improved by using some automation. In previous work on design tool integration [31], we have developed a technique for the structured capturing of the “traversal/action” code of generators. The approach was based on the observation that traversal sequences and actions to be taken at specific points in the input tree are separate concerns, and that the traversal can be specified using higher-level constructs (than procedural code). A language was designed that allowed the specification of traversal paths and the capturing of actions to be executed at specific nodes. Generators written using this approach were very compact and readable, and have been successfully applied in various projects. The approach is similar to Adaptive Programming [32], but it is more focused on the needs of generators.

5.2.3 Meta-generators

The approach based on the Visitor pattern has a serious shortcoming: most of the logic of the generator is still realized as procedural code, and therefore it is hard to verify or to reason about. A better technique would allow the mathematically precise modeling of the generator’s working, and the generation of the code of the generator from that model. This process is called *meta-generation*. Figure 13 above illustrates this process.

A “model of a generator” is much simpler than that of a compiler and it can be defined operationally: it is an abstract description of what the generator does. Following the generic description of a generator above, the main task of a generator can be modeled in terms of (1) the traversal sequences and (2) the transformation actions the generator takes during its operation. The approach described in the previous section allowed the specification of (2) only in imperative ways (in a programming language), but (2) can also be specified declaratively: using graph-transformation rules.

Graph grammars and graph rewriting [33] [60] offer a structured, formal, and mathematically precise method of representing how a generator constructs the output tree from the input tree. There are several practical systems ([62] [63]) available. Graph rewriting has been used to specify various program analysis and transformation tasks successfully [61]. In MIC, its scope is extended to include transformation on the models (instead of the abstract syntax tree of a program). One elementary rewriting operation performed by a translator is called a *transform*. A transform is a specification of a mapping between a portion of the input graph and a portion of the output graph. Note that the metamodels of the input and the output of a generator is a compact description of all the possible input and output structures. If $G_{in} = (C_{in}, A_{in})$ and

$G_{out} = (C_{out}, A_{out})$ denote the input and the output metamodels consisting of classes and associations, a transform can be described using the following elements:

- $g_{in} = (c_{in}, a_{in})$: subgraph formed from a subset $c_{in} \subseteq C_{in}$ of the input classes and a subset $a_{in} \subseteq A_{in}$ of the input associations.
- $F : G_{in} \rightarrow \{T, F\}$: a Boolean condition, called *filter*, over G_{in} .
- $g_{out} = (c_{out}, a_{out})$: subgraph formed from a subset $c_{out} \subseteq C_{out}$ of the output classes and a subset $a_{out} \subseteq A_{out}$ of the output associations.

- $M: g_{in} \rightarrow g_{out}$ a mapping where $g_{in} \subseteq G_{in}, g_{out} \subseteq G_{out}$, and $F(g_{in}) = T$.

A transform is a specific rewrite rule that converts a sub-graph of the input into a sub-graph of the output. The input sub-graph must also satisfy the filter. The mapping should also specify how the attributes of the output objects and links should be calculated from the attributes of the input objects and links.

While graph transformations are very descriptive, they fare less well in implementation. Matching the left hand side of a rewriting rule against an input graph involves searching for a sub-graph, which can be of exponential complexity. However, in generators one can almost always avoid the (global) search by specifying the traversal and order in which the transformation rules should be applied, thus the search can be reduced to a (local) matching. This latter one can be accomplished by introducing “pivot nodes”, which are bound by the higher-level, traversal strategy, so the left hand side of the transform is partially bound already when the rule is fired.

To summarize, a generator can be specified in terms of (1) a graph traversal, which describes in what order the nodes of the input tree should be visited, and (2) a set of transformation rules. The implementation of the above scheme is subject of active research. Early experiments [34] indicate the viability of the approach.

As an illustration for specifying a generator using metamodels and graph transformation rules, let us consider the example of an HSDF generator, which flattens a hierarchical signal flow graph. Figure 14 shows the metamodel for the source and the target of the generator. The left hand-side metamodel is that of the hierarchical signal flow, with **Primitives** and **Compounds**, **Input**, **Local**, and **Output** signals, which are connected through **Dataflow** connections. The right hand-side, target metamodel, consists of **Actors**, which have **Receive** and **Transmit** ports that are connected to **Queues**.

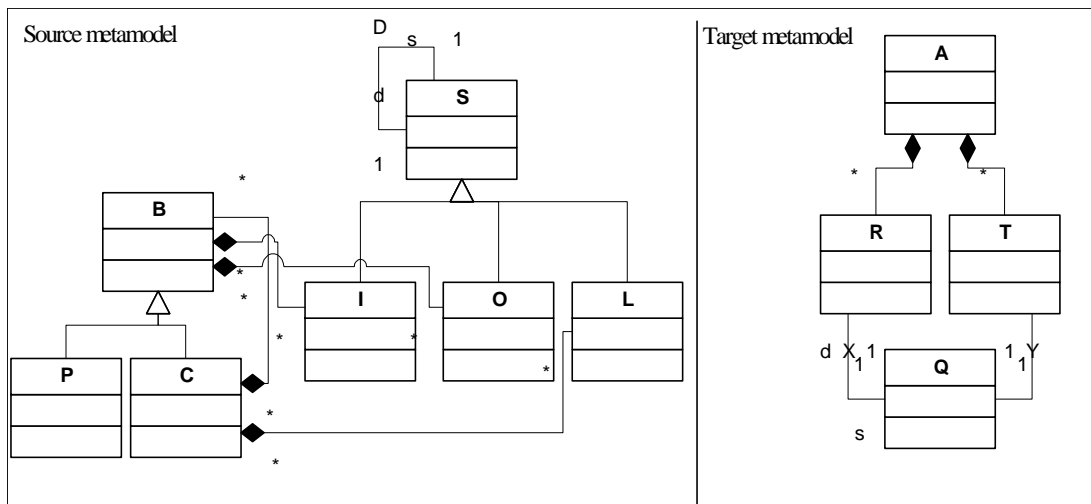


Figure 14: Source and target metamodels for example translation

On the diagrams below we illustrate a few graph rewriting rules. On the left side of a rule one can find a pattern (expressed as a —partial— UML class diagram), which is matched against a network of objects (that must be compliant with the source metamodel). On the right side of the rule one finds the network of target objects, which are created when the rule fires. This network must be compliant with the target metamodel. The arrows between the left and right hand sides illustrate correspondence: left hand side objects correspond to right hand side objects (and vice versa). The rules are for illustration only, with many details (like treating attributes) omitted for the sake of brevity.

The first translation rule on Figure 15 rewrites the input and output signal of a compound into queues. This rule shall be applied first, to the top-level compound of a hierarchical network.

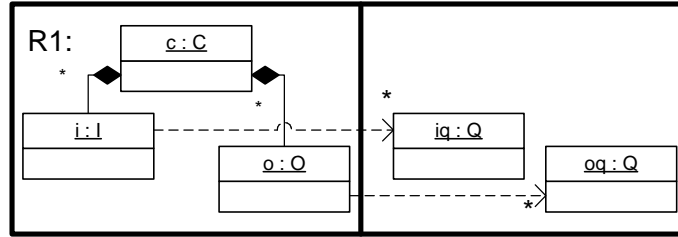


Figure 15: Rule 1: Convert (top-level) input and output ports into input and output queues

The next rule on Figure 16 rewrites the local signals of a compound into queues. This rule is applied to each compound in a hierarchy.

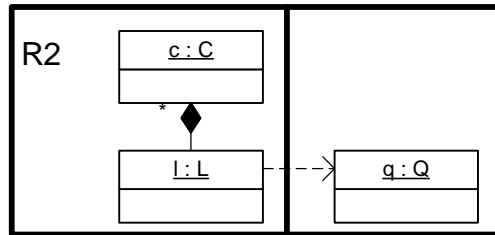


Figure 16: Rule 2: Convert local signals into (local) queues

The third rule on Figure 17 asserts that those signals of a compound C, which are connected to signals of the children compounds of C, should be mapped to the same queue. This rule ensures that queues created from higher-level compounds are “propagated down” to lower level compounds.

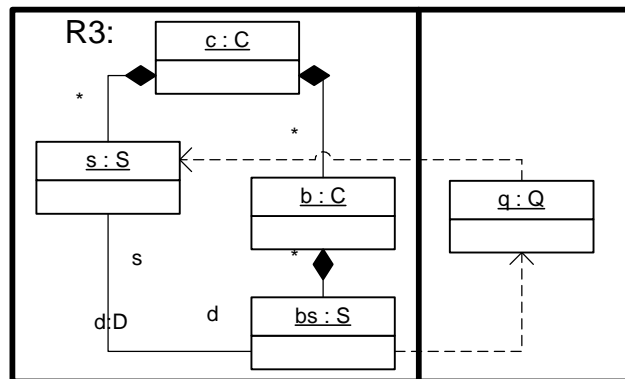


Figure 17: Rule 3: When a signal in a higher-level compound is connected to a signal of a lower-level block, the signals will be mapped to the same queue

The final rule on Figure 18 performs the most complex work: it rewrites primitives into actors. The receive and transmit ports of actors will correspond to the input and output signals of the parent primitive, and all connections between those and other signals will map to connections between queues and those ports in the target domain.

The rewriting rules presented have to be executed in certain order. First, Rule 1 is applied to the top-level compound. Next, in a depth-first manner the following sequence should be executed: Rule 2, Rule 3, and either Rule 2 recursively (if the child of the compound is a compound) or Rule 4 (if the child is a primitive).

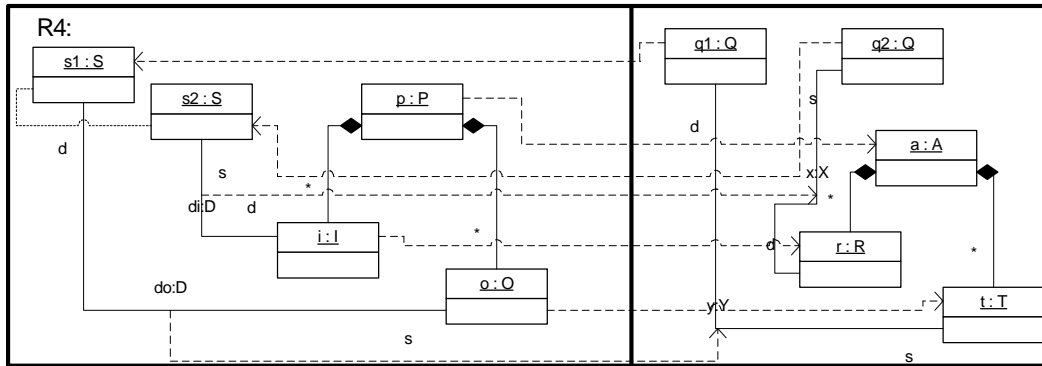


Figure 18: Rule 3: Connections from signals to the input and output signals of primitive blocks shall be replaced with connections to the receiver/transmitter ports of actors

The above example—informally—illustrates how graph-rewriting techniques can be used to model generators. The models can be employed in an interpretive settings, where a general-purpose graph rewriting engine incrementally transforms an input tree into an output tree, following the traversal sequences and transforms. The rewriting rules can also be converted (i.e. compiled) into efficient procedural code (see [54]).

6. Related Work

It has been increasingly recognized that conventional programming languages are not rich enough to provide efficient support for the composition of complex systems. In embedded systems, because of the interactions with the physical environment, these issues are even more significant. It is therefore essential to increase the level of abstraction for representing designs and to expand composition from today's hierarchical, modular composition to multi-faceted, generative composition [27]. Besides model-integrated computing approaches, several other important efforts work toward the same or similar goal. Below we mention three of these directions – Aspect-Oriented Programming (AOP), Intentional Programming (IP) and GenVoca generators, as well as new directions in using generators in the context of the Model-Driven Architecture (MDA) of the OMG [58].

6.1 Aspect-Oriented Programming (AOP)

The goal of AOP is to introduce a new decomposition concept in languages, *aspects*, which crosscut the conventional hierarchical, functional decomposition. AOP provides programmers with the opportunity to express separate concerns independently, and facilitates the merging (weaving) of components in an integrated implementation [5]. Aspect orientation fits well with the need of managing crosscutting constraints in embedded systems. Physical requirements in embedded systems, such as timing or synchrony, can be guaranteed by assigning them to a specific module, but they are the result of implementing their interaction in a particular manner. Changing these requirements may involve widespread changes to the functional components of the system, which makes component-based design and implementation using only functional composition very complicated.

AOP and MIC have strong similarity in addressing multiple-view system design explicitly. Both AOP and MIC allows the separation of design concerns in different aspects and allows capturing and managing interdependence among them. Composition of integrated systems is completed by weaving technology in AOP and model synthesis and generator technology in MIC. The main difference is in the level abstraction used. AOP research focuses on imperative languages and creates aspect-oriented versions such as AspectJ [28], while MIC targets DSMLs. Consequently, MIC is better suited for modeling, verifying and generating large, heterogeneous systems using larger components, while AOP provides better run-time performance due to the use of compiler technology in generating executable systems.

6.2 GenVoca

GenVoca is a generator technology that performs automated composition using precisely defined layers of abstractions in object-oriented languages [7]. The concept is based on the definition and explicit representation of design layers, where each layer refines the layer above. Design layers have standardized interfaces with alternative implementations. Layers are implemented using DSLs, which are implemented as extensions of existing languages. GenVoca generators convert these composition specifications into the source code of the host language. GenVoca is supported by an extensive toolsuite called the Jakarta Tool Suite (JTS), which provides a common infrastructure for extending standard languages with domain-specific constructs [29].

Regarding the level of abstraction used in describing designs, GenVoca resides between AOP and MIC. GenVoca technology still preserves the advantage of embedded DSLs: the generators output needs to go only to the level of the host language. Similarly to AOP, it results in highly efficient code due to the GenVoca-based optimization of component structure and the compiler technology of the host language environment. GenVoca strongly differs from both AOP and MIC in terms of supported decomposition strategy: at this point, GenVoca does not focus on multiple aspect composition (although extension in this direction seems feasible). Interestingly, the design-space exploration techniques used in MIC [23] and the composition validation technique in GenVoca (e.g. [30]) have strong similarities. Both of these techniques are based on a design-space definition using hierarchically layered alternatives and prune the potentially very large space using constraints (in GenVoca the goal is validation, in MIC the goal is to find configurations that satisfies the constraints).

6.3 Intentional Programming (IP)

IP is a bold experiment to transform programming from the conventional, “language focused” activity to a domain-specific, intention-based activity [6]. IP re-factors the conventional programming paradigm into intentional specification and transformers. Intentional specifications encode abstractions in graph data structures (active source) – without assigning a concrete syntax for representing them. Using various transformers manipulating the active source, the intentions can be visualized in different forms, and more importantly, can be assembled into complex programs (represented as intentions) by generators (or meta-programs).

There are many interesting parallels between IP and the other generative programming approaches discussed earlier. For example, in MIC the models (which are the expression of domain-specific constructs, designs) are represented in model databases in a format, which is independent from the concrete syntax used during modeling. Model transformation has a similarly central role in MIC as transformers in IP [59]: the MIC generators, synthesis tools are all directly manipulating the content of model databases for visualizing, analyzing, and composing models at different phases of the system development. Both in MIC and IP, efficient technology for defining and implementing transformers *is* a crucial issue. However, in MIC transformations are specified using graph-transformations: a higher-level technique than the procedural approach in IP.

6.4 Model-driven Architecture and model transformations

MDA [58] is a recent development in the industry community organized by OMG. The MDA vision outlines the relationships among various OMG standards and how they can be used in the various software development processes. One important aspect of MDA is the emphasis it places on specification models (in terms of a Platform Independent Model: PIM), on implementation models (in terms of a Platform Specific Model: PSM), and the mapping between the two. Obviously, MDA lends itself very well to transformational approaches, like the one MIC advocates. The difference is in terms of the modeling: while MIC emphasizes domain-specific modeling techniques and explicit, well-defined DSMLs, MDA tends to utilize UML for modeling.

Recent works on model transformations in the UML framework (e.g. [64],[65], and [66]) demonstrate the use of various transformational techniques in software development within the UML framework. Their key concept is to use transformations specified in the context of UML models. Similarly, transformations in MIC are expressed in a UML context, but the input to the transformations is always understood as

“sentences” in a DSML that models an embedded system. Specifying modeling languages in terms of UML metamodels (and transformations on those models) have appeared also been developed recently (see [67] [68][69] [70]). These approaches follow the same principles as MIC. However, tools support in the form of meta-programmable tools is still under development.

7. Conclusions

Successes of model-based tools, notably those of Matlab/Simulink and Stateflow, in developing embedded systems have shown not only the feasibility but also the superiority of the approach. In this paper, we argue for generalizing the approach even further: into Model-Integrated Computing, which advocates modeling as the central activity, integrated with analysis and synthesis plus generation. In MIC, Domain-Specific Modeling Languages play a key role in the creation, manipulation and transformation of the models. We have shown a method for defining these languages, in terms of: (1) their abstract syntax, (2) (visual) concrete syntax (with respect to a specific visualization engine, of GME), (3) their static semantics (in terms of well-formedness rules), and (4) their dynamic semantics (through translation). We call this process metamodeling, and have built a set of tools that support it.

We have used the MIC paradigm for developing several embedded applications [1]. Experience has shown that the metamodeling approach is extremely powerful, but it has to be used with care: changes in metamodels often invalidate existing domain models, requiring expensive re-building. Careful upfront analysis of the domain and the precise definition of the DSML (including static and dynamic semantics) have a great benefit for the entire lifetime of the systems built. MIC factors out the DSML development (i.e. the metamodeling) from the engineering process of embedded systems. The investment made in the metamodels is returned through the life of the products developed, in terms of reduced maintenance costs and ease of upgrade.

There are a number of directions for the enhancements of MIC, which are also subject of active research. Some the efforts include: (1) Metamodel composition and verification seeks to enhance the compositionality and quality of metamodels built from elements of a metamodel library. (2) Automatic domain-model migration based on the changes to the metamodels addresses the reuse of existing models under a new metamodel. (3) Configurable, alternative visual syntax for domain models deals with making the visualization (and direct manipulation) of models more flexible. (4) Incremental and component-oriented techniques for model translation and verification helps in avoiding lengthy and costly re-processing of models when small changes are made. These efforts shall further enhance the applicability of the technology.

Acknowledgement

The DARPA/ITO MOBIES program (F30602-00-1-0580) has supported, in part, the activities described in this paper.

References:

- [1] Sztipanovits, J. and G. Karsai: “Model-Integrated Computing,” *IEEE Computer*, April, 1997 (1997) 110-112
- [2] Sztipanovits, J., Karsai, G.: “Embedded Software: Opportunities and Challenges”, in *Embedded Software, Lecture Notes in Computer Science (LNCS 2211)*, pp. 403-415, Springer, 2001.
- [3] Matlab Simulink/Stateflow tools, www.mathworks.com
- [4] Gianfranco Ciardo, Kishor S. Trivedi, and Jogesh Muppala. SPNP: stochastic Petri net package. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models (PNPM'89)*, pages 142--151, Kyoto, Japan, 1989. IEEE Computer Society Press.
- [5] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.: “Aspect-Oriented Programming,” *ECOOP'97, LNCS 1241*, Springer. (1997)
- [6] Simonyi, C.: “Intentional Programming: Asymptotic Fun?” Position Paper, SDP Workshop Vanderbilt University, December 13 - 14, 2001. <http://isis.vanderbilt.edu/sdp>
- [7] Batory, D., Geraci, B.J.: “Composition, Validation and Subjectivity in GenVoca Generators,” *IEEE Transactions on SE*, pp. 67-82, February, 1997.

- [8] Kopetz, H.: *Real-Time Systems: Design Principles for Distributed Embedded Applications* Kluwer, 1997.
- [9] Schmidt, D.C., Levine, D.L., Mungee, S.: "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, Vol. 21, pp. 294-324, April, 1988.
- [10] Sangiovanni-Vincentelli, A.: "Defining Platform-based Design," *EEDesign*, February, 2002
- [11] Hudak, P.: Keynote address at the Usenix DSL Conference, 1997.
<http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/hudak-paul/hudak-dir/dsl/index.htm>
- [12] T. Clark, A. Evans, S. Kent, P. Sammut: "The MMF Approach to Engineering Object-Oriented Design Languages," Workshop on Language Descriptions, Tools and Applications (LDTA2001), April, 2001
- [13] OMG Unified Modeling Language Specification, Version 1.3. June, 1999
(<http://www.rational.com/media/uml/>)
- [14] CDIF Meta Model documentation. <http://www.metamodel.com/cdif-metamodel.html>
- [15] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997. (1997)
- [16] Karsai G., Nordstrom G., Ledeczki A., Sztipanovits J.: Specifying Graphical Modeling Systems Using Constraint-based Metamodels, IEEE Symposium on Computer Aided Control System Design, Conference CD-Rom, Anchorage, Alaska, September 25, 2000.
- [17] Generic Modeling Environment (GME 2000) Documentation
<http://www.isis.vanderbilt.edu/projects/gme/Doc.html>
- [18] Girault, A., Lee, B., Lee, E.A.: "Hierarchical Finite State Machines with Multiple Concurrency Models," Technical Memorandum UCB/ERL M97/57, Berkeley, Aug, 17, 1997.
- [19] Ledeczki A., Nordstrom G., Karsai G., Volgyesi P., Maroti M.: "On Metamodel Composition," IEEE CCA 2001, CD-Rom, Mexico City, Mexico, September 5, 2001.
- [20] Sifakis, J.: "Modeling Real-Time Systems – Challenges and Work Directions," EMSOFT 2001, LNCS 2211, Springer. (2001) 373-389
- [21] Lee, E.A., Xiong, Y.: "System-Level Types for Component-Based Design," EMSOFT 2001, LNCS 2211, Springer. (2001), pp. 37-253.
- [22] Butts, K., Bostic, D., Chutinan, A., Cook, J., Milam, B., Wand, Y.: "Usage Scenarios for an Automated Model Compiler," EMSOFT 2001, LNCS 2211, Springer. (2001) 66-79
- [23] Neema, S., "Design Space Representation and Management for Embedded Systems Synthesis," Technical Report, ISIS-01-203, February 2001.
http://www.isis.vanderbilt.edu/publications/archive/Neema_S_2_0_2003_Design_Spa.pdf
- [24] Rice, M.D., Seidman, S.B.: "A formal model for module interconnection languages," *Software Engineering*, IEEE Transactions on Software Engineering, Volume: 20 Issue: 1, Jan. 1994 pp: 88 -101
- [25] Lee, E.A. and Sangiovanni-Vincentelli, A.: "A Denotational Framework for Comparing Models of Computations," Technical Memorandum, UCB/ERL M97/11, January 30, 1997.
- [26] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, 1995.
- [27] Porter, A., Sztipanovits, J. (Ed.): *New Visions for Software Design and Productivity: Research and Applications*. Workshop Report of the Interagency Working Group for Information Technology Research and Development (ITRD) Software Design and Productivity (SDP) Coordinating Group. Vanderbilt University, December 13-14, 2001. <http://isis.vanderbilt.edu/sdp>
- [28] AspectJ: <http://aspectj.org>
- [29] Batory, D., Lofaso, B., Smaragdakis, Y.: "JTS: Tools for Implementing Domain-Specific Languages," 5th International Conference on Software Reuse, Victoria, Canada, June 1998.
- [30] Czarniecki, K., Eisenecker, U.W.: *Generative Programming*, Addison-Wesley, 2000
- [31] Karsai G., Gray J.: Component Generation Technology for Semantic Tool Integration, Proceedings of the IEEE Aerospace 2000, CD-Rom Reference 10.0303, Big Sky, MT, March, 2000.
- [32] Lieberherr, K.: "Adaptive Object-Oriented Software", Brooks/Cole Pub Co, 1995.
- [33] Rozenberg, G. (ed.), "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol.1-2. *World Scientific*, Singapore, 1997
- [34] Tihamer Levendovszky, Gabor Karsai, Miklos Maroti, Akos Ledeczki, Hassan Charaf: Model Reuse with Metamodel-Based Transformations. ICSR 2002: 166-178
- [35] Heiner, G.: Automotive applications, Proceedings of the Joint Workshop on Advanced Real-time Systems, Austrian Research Centers, Vienna, March 26, 2001
- [36] E.A. Lee and D.G. Messerschmitt. *Synchronous data flow*. Proceedings of the IEEE, 75(9):1235- 1245, September 1987.

- [37] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998
- [38] D. Hatley and I. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, 1987.
- [39] Carnes J., Misra A.: *Model-Integrated Toolset for Fault Detection, Isolation and Recovery (FDIR)*, International Conference and Workshop on Engineering of Computer Based Systems Friedrichshafen, Germany, March 11, 1996.
- [40] Clark, T., Evans, A., Kent, S.: "Engineering Modeling Languages: A Precise Metamodeling Approach," R.-D. Kutsche and H. Weber (Eds.): *FASE 2000*, LNCS 2306, pp. 159-173, 2002
- [41] Spivey, J.M.: *The Z Notation: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1992
- [42] Guttag, J.V., Horning, J.: *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
- [43] Bourdeau, R.H., Cheng, B. E.C.: "A Formal Semantics for Object Model Diagrams," *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, October 1995, pp 799-821.
- [44] Moore M., Sztipanovits J., Karsai G., Nichols J.: *A Model-Integrated Program Synthesis Environment for Parallel/Real-Time Image Processing*, SPIE Conference on Parallel and Distributed Methods for Image Processing, pp 31-45, San Diego, CA, June, 1997.
- [45] Bapty T., Ledeczi A., Davis J., Abbott B., Howard L., Tibbals T.: *Turbine Engine Diagnostics Using a Parallel Signal Processor*, Joint Technology Showcase on Integrated Monitoring, Diagnostics, and Failure Prevention, , Mobile, AL, 1996.
- [46] M. Chu, H. Haussecker, F. Zhao, "Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks." *Int'l J. High Performance Computing Applications*, to appear, 2002. Also, Xerox Palo Alto Research Center Technical Report P2001-10113, May 2001.
- [47] Szedo G., Neema S., Scott J., Bapty T.: *Reconfigurable Target Recognition System*, Proceedings of the FPGA 2000, , Monterey, CA , February, 2000.
- [48] Nelson T.: *Implementation of Image Processing Algorithms on FPGA Hardware*, Master's Thesis, Vanderbilt University, Electrical Engineering, 2000.
- [49] Neema S., Sztipanovits J., Karsai, G.: *Design-Space Construction and Exploration in Platform-Based Design*, ISIS-02-301, June 24, 2002.
- [50] Kobryn, C.: *UML 2001: A Standardization Odyssey*, Communications of the ACM, vol. 42, no. 10, October, 1999.
- [51] Extensible Markup Language, <http://www.w3.org/XML/>.
- [52] Sprinkle J., Karsai G., Ledeczi A., Nordstrom G.: *The New Metamodeling Generation*, IEEE Engineering of Computer Based Systems, Proceedings p.275, Washington, D.C., USA, April, 2001.
- [53] Thomas A. Henzinger. *The theory of hybrid automata*. Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS), IEEE Computer Society Press, 1996, pp. 278-292.
- [54] Albert Zündorf: *Graph Pattern Matching in PROGRES*, LNCS 1073, TAGT 1994: pp. 454-468.
- [55] Ledeczi A.: *System Synthesis for Parallel Signal Processing*, International Conference on Signal Processing Applications and Technology, pp. 1507-1511, Boston, MA, 1995.
- [56] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Transactions on CAD*, Vol. 17, No. 12, December 1998.
- [57] W.-T. Chang, S.-H. Ha, and E. A. Lee, "Heterogeneous Simulation – Mixing Discrete-Event Models with Dataflow," invited paper, *Journal on VLSI Signal Processing*, Vol. 13, No. 1, January 1997.
- [58] *Model-Driven Architecture at the Object Management Group's website*: <http://www.omg.org/mda/>.
- [59] Aitken, William; Dickens, Brian; Kwiatkowski, Paul; de Moor, Oege; Richter, David; Simonyi, Charles: *Transformation in Intentional Programming*. Proceedings of the Fifth International Conference on Software Reuse. IEEE Computer Society Press, pp. 114- 123, 1998.
- [60] Dorothea Blostein, Andy Schürr: *Computing with Graphs and Graph Transformations*. *Software - Practice and Experience* 29(3): 197-217, 1999.
- [61] U. Aßmann, "How To Uniformly Specify Program Analysis and Transformation", in: 6th Int. Conf. on Compiler Construction (CC '96), T. Gyimóthy (éd.), *Lect. Notes in Comp. Sci.*, Springer-Verlag, Linköping, Sweden, 1996.
- [62] A. Schürr. *PROGRES for Beginners*. RWTH Aachen, D-52056 Aachen, Germany.
- [63] Taentzer, G.: *AGG: A Tool Environment for Algebraic Graph Transformation*, in *Proc. of Applications of Graph Transformation with Industrial Relevance*, Kerkrade, The Netherlands, LNCS, Springer, 2000.

- [64] Milicev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," IEEE Transaction on Software Engineering, Vol. 28, No. 4, April 2002, pp. 413-431.
- [65] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h.: UMLAUT: an extendible UML transformation framework, in Proc. Automated Software Engineering, ASE'99, Florida, October 1999.
- [66] David H. Akehurst: Model translation: A uml-based specification technique and active implementation approach. PhD thesis, Computer Science at Kent University (UK), December 2000.
- [67] Tony Clark, Andy Evans, Stuart Kent: Engineering Modelling Languages: A Precise Meta-Modelling Approach. FASE 2002: 159-173
- [68] Tony Clark, Andy Evans, Stuart Kent: The Metamodelling Language Calculus: Foundation Semantics for UML. FASE 2001: 17-31.
- [69] Lemesle, R. Transformation Rules Based on Meta-Modeling EDOC,'98, La Jolla, California, 3-5, November 1998, pp.113-122.
- [70] Heckel, R. and Küster, J. and Taentzer, G.: Towards Automatic Translation of UML Models into Semantic Domains, Proc. of APPLIGRAPH Workshop on Applied Graph Transformation (AGT 2002), Grenoble, France, 2002, pp. 11 - 22.