

Generative Programming for Embedded Systems

Janos Sztipanovits and Gabor Karsai

Institute for Software Integrated Systems
Vanderbilt University, P.O. Box 1829 Sta. B. Nashville, TN 37235, USA
{[sztipaj](mailto:sztipaj@vuse.vanderbilt.edu),[gabor](mailto:gabor@vuse.vanderbilt.edu)}@vuse.vanderbilt.edu

Abstract. Embedded systems represent fundamentally new challenges for software design, which render conventional approaches to software composition ineffective. Starting with the unique challenges of building embedded systems, this paper discusses key issues of model-based technology for embedded systems. The discussion uses Model-Integrated Computing (MIC) as an example for model-based software development. In MIC, domain-specific, multiple view models are used in all phases of the development process. Models explicitly represent the embedded software and the environment it operates in, and capture the requirements 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 from them using the techniques of program generators.

1 Introduction

Perhaps the biggest impact of the “IT explosion” in the last decade has been the emerging role of computing and software as “universal system integrator.” Systems are formed from interacting components. The new trend is that an increasing number of components and interactions in real-life systems, which were previously physical, are becoming *computational*. From large-scale systems, such as manufacturing processes or command and control (C^2) systems, to small systems, such as automobiles and simple appliances, interaction and coordination of physical components increasingly involves digital information processing and communication. For example, the automotive industry is currently testing “brake-by-wire” systems, where the currently dominant hydraulic and mechanical brake systems will be replaced by position sensors observing the brake pedal, electrical actuators exerting braking force, and a distributed, embedded computer system, which computes the optimum force distribution according to the driving conditions. Two important consequences of this change are the following:

- **Increased fusion of software into application domains.** In many application domains, computing has become the focal point of complexity and the primary source of new functionality. For example, over 90% of innovations in the automotive industry come from embedded computing [33]. The increased significance of computing means that unless unique characteristics of the

application domain are reflected directly in the software development paradigms, application engineering considerations must be mapped manually onto general-purpose software engineering concepts and tools, which is tedious and error-prone for both the domain experts and the software experts. The difficulty of this manual mapping process creates the need for sharply tailored capabilities, such as domain-specific languages, generators, and composition platforms that enable building of applications from components.

- **Physicality in software design.** In embedded computing applications, the role of the embedded software is to configure and control the operation of programmable computing devices so as to meet physical requirements at their sensor-actuator interfaces. This deep integration of computing with physical systems implies that essential physical characteristics of systems (such as latency, noise, power consumption) are strongly influenced — or simply determined by — software. Consequently, software requirements become multi-faceted, i.e., computational platforms and software must satisfy functional and physical requirements *simultaneously*.

The goal of this paper is to discuss challenges and opportunities of generative programming for embedded software development. We use the term “generative programming” in a broad sense: systems or components of systems are produced automatically from abstract terms [28]. The examples for possible solutions are based on our experience gained from the *Model-Integrated Computing (MIC)* effort at the Institute of Software Integrated Systems (ISIS) at Vanderbilt University [1].

The outline of the paper is the following: In Section 2, we examine the challenges of composing complex systems from components and describe a model-based extension of composition platforms. Section 3 summarizes the challenges and discusses the MIC approach in modeling languages and model building. In Section 4, we provide an overview of generator technologies in the MIC framework. Section 5 summarizes some of the relevant approaches and compares them with MIC.

2 Significance of Generative Programming

Composition and component-based design are key tools in modern software engineering for managing complexity. The concept of component-based design is straightforward: systems are built by composing software components with precisely defined interfaces using standardized interconnection mechanisms. “Plug-and-play” construction is supported by an underlying composition framework, such as CORBA or COM+, which facilitates the component interactions by providing standard services such as a request broker, interface repository, location service, and others. Unfortunately, this solution tends to work either for building systems with relatively coarse-grained components using small amounts of “glue code”, or for very small systems with a few components because of the following two problems:

1. Component interfaces in conventional standards-based composition frameworks only capture the signature, but not the semantics of components. Consequently, design integrity for the composed system may quickly be lost and inconsistencies

may emerge by simply combining components based on compatibility of interface signatures.

2. To achieve flexibility, components are frequently designed to be customizable to different application contexts via parameterization or via the selection of alternative implementations. In large systems, these choices represent complex, interacting assumptions about operating conditions, which may lead to brittleness.

The success of building applications based on coarse-grained components is the result of strong restrictions on component interactions: the dominant components (such as databases or web browsers) preserve the overall design integrity. In case of small systems, the system designers can preserve design integrity without extensive tool support through heroic manual effort.

Although software composition is rarely easy, the problems of software composition for embedded systems are particularly hard. Physical processes surround embedded computers, which receive their inputs from sensors and send their outputs to actuators. When viewed from their sensor and actuator interfaces, embedded computing devices act like physical processes with dynamics, noise, fault, size, power and other physical characteristics. *The role of the embedded software is to “configure” the computing device in order to meet its physical requirements* [2]. The mapping of logical behavior into physical behavior is influenced by the detailed physical characteristics of the devices involved, including their physical architecture, instruction execution speed, bus bandwidth, power dissipation, etc. In addition, modern processor architectures introduce complex interactions between the software and essential physical characteristics of the underlying devices.

It is not surprising that using current software technology logical/functional composability does not imply physical composability. In fact, *physical properties are not composable*, rather, they appear as cross-cutting constraints in the development process. The effects of such cross-cutting constraints can be devastating for the design. Meeting specifications in one part of the system may destroy performance in others, and, additionally, many of the problems will surface at system integration time rather than during unit development and testing. Consequently, we need to change our approach to the design of embedded software: productivity increases must come from tools that directly address the design of the whole system with its many different physical and logical aspects.

There are several comprehensive approaches, such as Model-Integrated Computing (MIC) [1], Aspect-Oriented Programming [3], Intentional Programming [4], GenVoca Architecture [5], that attempt to answer problems of composing complex systems. In our discussion, we focus on MIC and point out some of the similarities and differences with the other approaches.

Figure 1 shows an overview of the MIC architecture. The applications and infrastructure software (left side) are defined by application models and platform models. The difference between MIC for embedded and non-embedded systems is particularly significant regarding the scope and composition of models. In order to make the physical properties of the embedded system computable and analyzable, models capturing only the logical characteristics of applications and infrastructure software are insufficient. The models must include physical properties of the platforms and the mapping between the application and platform models. The scope of modeling and the required level of abstraction are highly domain-specific. We

cannot expect that the same types of models are used to design controllers for break-by-wire system in cars (where safety, timing and cost are the critical properties) and to design mobile phones (where besides cost, power, security, and feature richness are the most important factors).

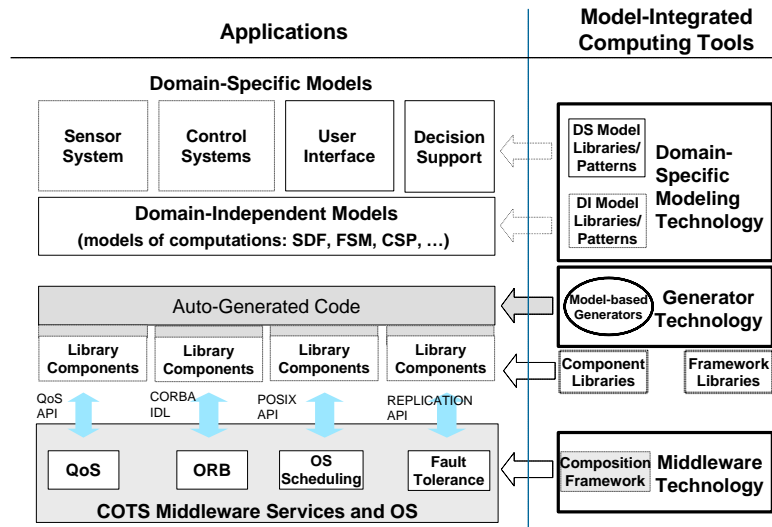


Figure 1: Model-integrated approach to software composition

Model-based approaches are not practical without extensive tool support. The right side of Figure 1 shows the MIC tool components. Support for modeling includes tools for building domain-specific modeling environments, tools for analyzing and synthesizing models, and model translators that help to integrate large, heterogeneous tool environments. Section 3 discusses the main components of the MIC modeling framework.

While models define and characterize the embedded system applications, the applications themselves are formed by a set of customized components and the underlying hardware/software platform (see Figure 1). There are numerous hardware/software platforms that are commonly used for building embedded system applications. For example, the Time-Triggered Architecture (TTA) [6] and the Real-time CORBA Event Service [7] provide composition platforms for embedded software and systems with very different properties. While TTA supports a time triggered, synchronous model of computation, the Real-time CORBA Event Service follows an event-triggered, fully asynchronous model. The platforms have a very important role in model-based design: they enable the use of idealized assumptions (such as synchrony, loss-less communication, guaranteed communication bandwidth) about the interaction and behavior of components, which largely simplifies the modeling task. In fact, these idealized and simplified abstractions (or layers of abstractions) between applications and platforms are the basis for platform-based design [8].

Building an integrated application on a platform is a complex task. It includes the consistent parameterization/customization of the selected components and of the hardware/software platform via the provided API-s and creation of any additional new code for connecting the components to the platform. In the MIC framework these steps must be tied directly to the models, otherwise the connection between model properties and the modeled system will be lost. This need makes the model-based generators and generator technology a fundamental component of the MIC tool architecture. Section 4 examines different approaches for building model-based generators.

3 Modeling and Model Building

The two fundamental problems we consider in this section are the composition of domain-specific modeling languages and the composition of models.

3.1 Domain Specific Modeling Languages

Domain-specific languages (DSLs) have significant impact on the design process [9]. In embedded systems, where computation and communication always occur in the context of a physical domain, DSLs offer an effective way to structure the information about the system to be designed along the “natural dimensions” of the applications. The chief difficulty with DSLs is the cost of developing solid semantic foundations and tools to support their use. MIC resolves this difficulty using a *meta-modeling approach* [14]. Using standard denotational semantics style (see e.g. [10]), a *modeling language* is defined as a five-tuple of concrete syntax (C), abstract syntax (A), semantic domain (S) and semantic and syntactic mappings (M_S , and M_C):

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

The C concrete syntax defines the form of representation, such as visual, textual, or mixed. The A abstract syntax defines the *concepts*, *relationships*, and *integrity constraints* available in the language. Thus, the abstract syntax determines all the (syntactically) correct “sentences” (in this case models) that can be built. (It is important to note the A 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 some mathematical formalism in terms of which the meaning of the models is explained. The $M_C : A @ C$ mapping assigns syntactic constructs (visual, 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.

Rapid composition of *domain-specific modeling languages (DSMLs)* requires tool support. The tools need representation formalism for the syntactic elements (C and A), the semantic domain, and the syntactic and semantic mapping. The languages used for this purpose are called *meta-languages* and the models describing a DSML are called *meta-models*. Since a meta-modeling language can also be considered a domain-specific modeling language (with the domain being that of “modeling languages”), it

is desirable that the meta-modeling language be powerful enough to describe itself, in a meta-circular manner. This corresponds to the *four-layer meta-model language architecture* (see Figure 2) used in UML [11] or in CDIF [12].

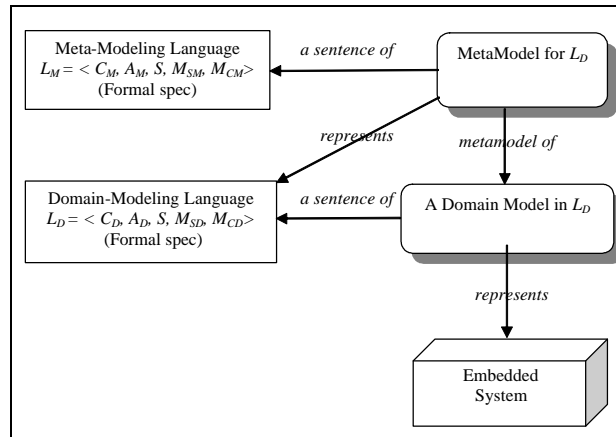


Figure 2: Meta-modeling language architecture

3.2 Tools for Domain-Specific Modeling

The ISIS toolset for Model Integrated Computing uses the *same* generic, meta-programmable modeling environment (Generic Modeling Environment – GME) for meta-modeling and domain modeling [15]. The GME tool architecture supports meta-circularity: a meta-meta-model (defined in the meta-language) configures the environment to support meta-modeling. We have adopted UML class diagrams (with stereotypes) [11] and the Object Constraint Language (OCL) [13] as the meta-language for modeling the *A* abstract syntax of DSMLs. 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 meta-modeling language architecture. It is important to note that adopting UML for meta-modeling does not imply any commitment to use UML as a domain modeling language, though it can certainly be used where appropriate. Further details on meta-modeling can be found in [14].

Consider a simple example for the abstract syntax of a DSML for signal flow modeling (SF). The A_{SF} abstract syntax of *SF* is shown in Figure 3a as a meta-model expressed in terms of a UML class diagram (we have omitted the integrity constraints). The core concepts of this language are *Compounds*, *Primitives*, *Ports*, and *Signals*. *Primitives* form the basic signal processing blocks (e.g., *Filters*, *FFT*, *IFFT*, *Correlation*, etc.). *Ports* define the I/O interfaces of these blocks, and *Signals* represent the signal-flow between the blocks. *Compounds* are processing blocks that can be decomposed into other *Compounds*, and/or *Primitives*. An abstract Base concept combines *Compounds* and *Primitives* to represent an abstract

signal-processing block. This abstract syntax is complemented with the $M_C : A @ C$ mapping to obtain a simple DSML for signal processing. (The specification of this mapping is part of the meta-programmable Graphical Modeling Environment system [15] and not shown here.) Figure 3b shows a simple hierarchical application model, which is an instance of the meta-model in Figure 3a.

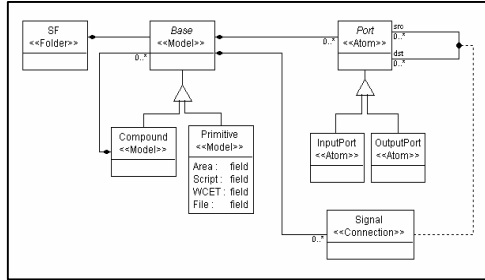


Figure 3a: Meta-model for SF

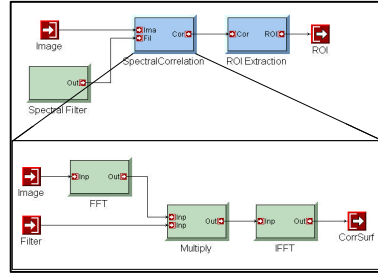


Figure 3b: Simple application model

In practice, the specification of meta-models can be quite a complex undertaking. It requires a thorough understanding of the S semantic domain and formulation of the well-formedness rules such that the semantic mapping of each well-formed model leads to a consistent semantic model. For example, if SF has synchronous dataflow (SDF) semantics [16], one of the well formed-ness rules must prohibit the connection of two different output ports to the same input port:

```
Self.InputPorts() -> forAll(ip | ip.src() -> forAll(x1, x2 | x1=x2))
```

However, if we use for SF dynamic dataflow (DDF) semantics [16], the same constraint may not necessarily apply (depending on the actual semantics). (We will discuss the assignment of semantics to SF in Section 4.)

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

3.3 Composing Meta-Models

An obvious way to decrease the difficulty of defining DSMLs is to compose them from sub-languages. This is particularly important in application domains such as embedded systems, which frequently require many modeling views. Compositional construction of DSMLs requires the $L_n = L_1 || L_2 \dots || L_k$ composition of meta-models from constituent DSMLs. While the composition of orthogonal (independent) sub-languages is a simple task, construction of DSMLs with non-orthogonal views is a

complex problem. Non-orthogonality means that component DSMLs may share concepts and well formed-ness rules may span over the individual modeling views.




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: Meta-model composition operators

The current design of ISIS's composable meta-modeling environment in GME leaves the component meta-models intact and creates meta-models that are further composable [17]. The composition is accomplished by introducing three (new) operators for combining meta-models (see Table 1). Application of these operators generates a new meta-model that conforms to the underlying semantics of UML. Decomposition of the UML inheritance operation allows finer control over meta-model composition (details are discussed in [17]). Unfortunately, meta-model composition is not complete by executing the composition operators. If the component meta-models are not orthogonal, it is possible that the resulting meta-model is inconsistent, which means that conflicting integrity constraints are created during the composition process. This means that the meta-modeling toolset needs to be extended with a validation tool (such as PVS [36]), which automatically checks the consistency of the well-formedness rules. (This tool is not part of the current version of GME.)

3.4 Modeling and Model Synthesis

As we pointed out above, modeling and model-based approaches [18][19] play central role in embedded software and system development, where mathematical modeling of physical characteristics is essential. The fundamental promise of model-based approaches is that experimental system verification will largely be replaced by model-based verification. This is extremely important, since testing is very expensive and cannot be exhaustive in real-life systems. Even by taking advantage DSMLs and advanced model-builders, creating high-fidelity models is expensive. Developing efficient methods for model-building is therefore an important research goal. Below, we briefly discuss two important approaches: compositional modeling and model synthesis.

3.4.1 Compositional modeling

Building complex models by composing components $M_n = M_1 \parallel M_2 \dots \parallel M_k$ 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 [6]. Therefore, DMSLs that are formal enough to be analyzable and analysis tools that can verify essential properties of designs are crucial in model-based system/software development.

3.4.2 Model Synthesis

Model synthesis in a 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 $M_d = M_i \parallel M_j \dots \parallel M_l$ 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. By narrowing the scope of the synthesis task, however, we can formulate solvable problems. For example, by using patterns [37] 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 a design space using hierarchically layered alternatives relatively simple [20].

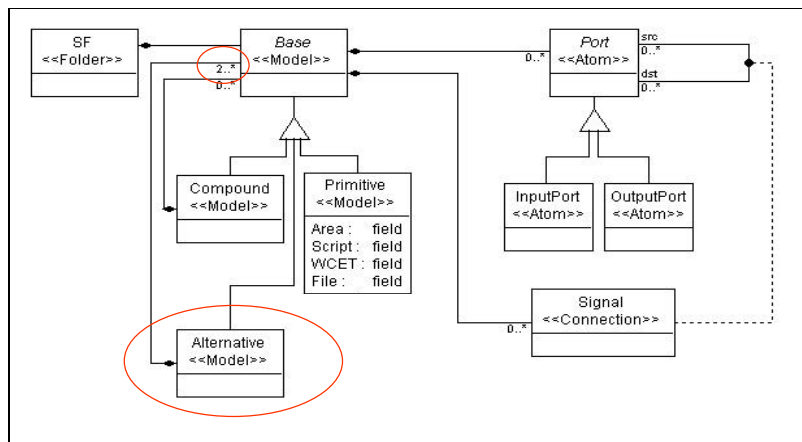


Figure 4: Meta-model of SF extended with the “Alternative” construct

To enable the representation of design spaces, we need to expand DSMLs with the ability to represent design alternatives explicitly. As an example, Figure 4 shows the meta-model 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 meta-model 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 depends largely 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 [21].

4 Generators and Model Transformations

The last step of the model-based development process is to generate the application on the underlying hardware/software platform (see Figure 1) using the verified domain-specific models. In embedded systems where model-based verification is crucial to keep the cost and complexity of experimental system verification tolerable, maintaining a direct link between the verified models and the application is mandatory. The role of the *model-based generators* in the MIC framework is to generate the “glue” required to compose the integrated application from library components, consistently parameterize the components, and customize the composition platform.

4.1. Model-based generators

Since generators always work with some selected representation of models, they can be modeled as a mapping between the abstract syntax of their L_i input and L_o output languages: $G: A_i @A_o$. Complexity of generators depends largely on the semantic relationship between the input and output languages. Our previous example, *SF*, is a declarative hierarchical module interconnection language, L_{SF} , which describes the *structure* of signal processing systems. In this sense, we can assign a *structural semantics* to L_{SF} by providing the semantic mapping between its abstract syntax and a semantic domain, which is represented by set-relational mathematics. (A closely related example for this can be found in [22].)

At this point, L_{SF} does not have *behavioral semantics*. Behavioral semantics can be assigned to L_{SF} by defining a mapping between L_{SF} and another modeling language with well-defined operational semantics, such as synchronous dataflow (SDF). For example, we may adopt the SDF behavioral semantics as defined in [23]. Of course, the precise definition of the SDF semantics has also required the definition of the structure of a dataflow graph, which can be expressed by a language L_{SDF} with abstract syntax A_{SDF} . Figure 5 shows a simple meta-model for homogeneous SDF [16], where each node firing consumes and generates exactly one data token at the input and output ports, respectively. We define the operational semantics of L_{SF} by the $G: A_{SF} @A_{SDF}$ mapping. In fact, G is also the specification of the generator, which

maps $I_{SF} \in L_{SF}$ signal processing system models into $I_{SDF} \in L_{SDF}$ synchronous dataflow models that can be executed by a synchronous dataflow virtual machine.

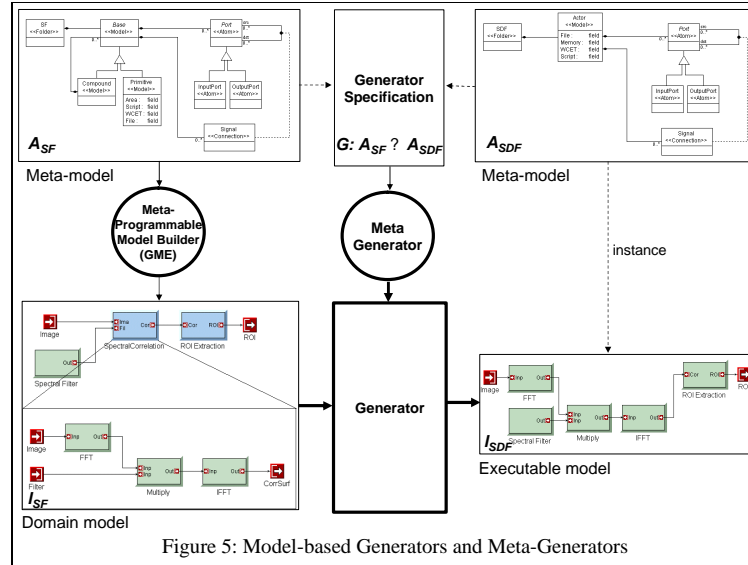


Figure 5: Model-based Generators and Meta-Generators

The specification of the mapping executed by the generators is done based on our understanding of the relationship between the semantic domains and semantic mappings. An important requirement for the generators is consistency: a generator is consistent if well formed-ness rules in the input and output modeling languages are consistent.

4.2. Building Generators

As we have shown, generators execute model transformations. During the design process, the design evolves along the iterative refinement, fusion, composition, and analysis of models. The design models change not only because of their evolution but also because of the need to transform domain-specific design models to the native models of various analysis and synthesis tools. Model transformation is the pre-condition for reusing generic tools in different, domain-specific design processes. Due to the extensive need for generators in all phases of the design process, technology for building generators is an important one. As we have shown above, the specification of generators is expressed in terms of the meta-models of the input and output languages. Below we summarize some of the approaches for creating generators based on these specifications.

4.2.1 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 a hardware architecture, a generator maps the abstract syntax of the input language into the abstract syntax of the target language, where the target has a well-defined execution semantics, with high-level “instructions”. If one omits the technical details of creating the output product in some form (e.g. text files), the main task of a generator is reduced to create a “target tree” from an “input tree”. The “input tree” is a 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”.¹ Naturally, in the simplest of cases the output product can be produced directly from the input tree.

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

1. Construct the input tree. This step is often 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 should visit 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 meta-models of the input and target languages already), one has to design the appropriate code sequences for traversal and constructions. 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 implement 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. The disadvantage is that the generator code is hard to maintain and hard to verify.

4.2.2 Pattern-based Design

The scheme described above can also be implemented in a more structured way, by using the *Visitor* design pattern [24]. The main task of a generator involves the *traversal* of the input tree and *taking actions* at specific points during the traversal.

¹ We use the term “tree” here, although 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.

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 that *accepts* the visitor object and 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 improved significantly by using some form of automation. In previous work on design tool integration [29], 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 [30], but it is more focused on the needs of generators.

4.2.3 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 precise mathematical modeling of the generator’s workings, and the generation of the code of the generator from that model. This process is called *meta-generation*.

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) in pure procedural terms, but it can also be specified declaratively using graph-transformation rules.

Graph grammars and graph rewriting [31] offer a structured, formal, and mathematically precise method of representing how a generator constructs the output tree from the input tree. 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 meta-models 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 meta-models 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 descriptive, they are computationally expensive. 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 mapping 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 [32] indicate the viability of the meta-generator approach.

5 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. 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 [25]. 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 – with the limited purpose of examining their approach to the challenges listed in Section 2.

5.1 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 [3]. 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 AspectJ [26], 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.

5.2 GenVoca

GenVoca is a generator technology that performs automated composition using precisely defined layers of abstractions in object-oriented languages [5]. The concept is based on the definition and explicit representation of designs 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 [27].

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 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 [21] and the composition validation technique in GenVoca (e.g. [28]) 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).

5.3 Intentional Programming (IP)

IP is a bold experiment to transform programming from the conventional, “language focused” activity to a domain-specific, intention-based activity [4]. IP refactors 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: 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.

6 Conclusion

DSMLs, model-based generators, and composition frameworks are important elements of software/system technology for embedded computing. In fact, the prevalence of crosscutting constraints in embedded software makes the design process intractable without the extensive use of generative programming. However, the widespread use of these technologies requires drastic decrease in the cost of supporting tools. We believe that this can be achieved by applying the same ideas a level higher: compositional construction of DSMLs via meta-modeling, development of meta-generators, and composable tool environments built on affordable model transformation technology.

It is quite remarkable to observe the intellectual similarity among MIC, AOP, GenVoca, and IP. Although emerging in quite different communities, answering diverse and different needs, and using very different techniques, they still advocate a core set of new concepts in software and system development: (1) extension of the currently dominant hierarchical, modular composition to multiple-aspect composition, (2) the extensive use and support for domain specific abstractions, and (3) the fundamental role of generative programming and generators in all phases of the system development. In addition, there is a strong trend in the component-based software development and middleware community toward the increased use of modeling and model-based composition in system building [4][35]. It will be interesting to observe whether or not these different approaches will have the strength and will reinforce each other enough to redefine programming during the next decade.

Acknowledgments

The authors would like to thank Don Batory, Gregor Kiczales, Doug Schmidt, and Charles Simonyi for stimulating discussions on the topic. This work was supported by the DARPA MoBIES contract, F30602-00-1-0580.

References

- [1] J. Sztipanovits 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] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.: "Aspect-Oriented Programming," ECOOP'97, LNCS 1241, Springer. (1997)
- [4] Simonyi, C.: "Intentional Programming: Asymptotic Fun?" Position Paper, SDP Workshop Vanderbilt University December 13 - 14, 2001. <http://isis.vanderbilt.edu/sdp>
- [5] Batory, D., Geraci, B.J.: "Composition, Validation and Subjectivity in GenVoca Generators," *IEEE Transactions on SE*, pp. 67-82, February, 1997.
- [6] Kopetz, H.: *Real-Time Systems: Design Principles for Distributed Embedded Applications* Kluwer, 1997.
- [7] Harrison T., Schmidt, D.C., Levine, D.L.: "The Design and Performance of a Real-time CORBA Event Service," *Proceedings of the OOPSLA Conference*, October, 1997.
- [8] Sangiovanni-Vincentelli, A.: "Defining Platform-based Design," *EEDesign*, February, 2002
- [9] P. Hudak: Keynote address at the Usenix DSL Conference, 1997. <http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/hudak-paul/hudak-dir/dsl/index.htm>
- [10] 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.
- [11] OMG Unified Modeling Language Specification, Version 1.3. June, 1999 (<http://www.rational.com/media/uml/>).
- [12] CDIF Meta Model documentation. <http://www.metamodel.com/cdif-metamodel.html>
- [13] Object Constraint Language Specification, ver. 1.1, Rational Software Corporation, et al., Sept. 1997. (1997).
- [14] Karsai G., Nordstrom G., Ledeczi 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.
- [15] *Generic Modeling Environment (GME 2000) Documentation* <http://www.isis.vanderbilt.edu/projects/gme/Doc.html>
- [16] 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.
- [17] Ledeczi A., Nordstrom G., Karsai G., Volgyesi P., Maroti M.: "On Metamodel Composition," *IEEE CCA 2001*, CD-Rom, Mexico City, Mexico, September 5, 2001.
- [18] "Sifakis, J.: "Modeling Real-Time Systems – Challenges and Work Directions," *EMSOFT 2001*, LNCS 2211, Springer. (2001) 373-389
- [19] Lee, E.A., Xiong, Y.: "System-Level Types for Component-Based Design," *EMSOFT 2001*, LNCS 2211, Springer. (2001) 237-253

- [20] 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
- [21] 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
- [22] Rice, M.D., Seidman, S.B.: "A formal model for module interconnection languages," Software Engineering, IEEE Transactions on , Volume: 20 Issue: 1 , Jan. 1994 pp: 88 -101
- [23] Lee, E.A. and Sangiovanni-Vincentelli, A: "A Denotational Framework for Comparing Models of Computations," Technical Memorandum, UCB/ERL M97/11, January 30, 1997.
- [24] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, 1995.
- [25] 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>
- [26] AspectJ: <http://aspectj.org>
- [27] Batory, D., Lofaso, B., Smaragdakis, Y.: "JTS: Tools for Implementing Domain-Specific Languages," 5th International Conference on Software Reuse, Victoria, Canada, June 1998.
- [28] Czarnecki, K., Eisenecker, U.W.: *Generative Programming*, Addison-Wesley, 2000
- [29] 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.
- [30] Lieberherr, K.: "Adaptive Object-Oriented Software", Brooks/Cole Pub Co, 1995.
- [31] Rozenberg, G. (ed.), "Handbook on Graph Grammars and Computing by Graph Transformation: Foundations", Vol.1-2. *World Scientific*, Singapore, 1997
- [32] Tihamer Levendovszky, Gabor Karsai, Miklos Maroti, Akos Ledeczki, Hassan Charaf: *Model Reuse with Metamodel-Based Transformations*. ICSR 2002: 166-178
- [33] Heiner, G.: *Automotive applications*, Proceedings of the Joint Workshop on Advanced Real-time Systems, Austrian Research Centers, Vienna, March 26, 2001
- [34] Object Management Group: "Model Driven Architecture" <http://www.omg.org/mda/>
- [35] Gokhale, A. Schmidt, D.C., Natarajan, B., and Nanbor Wang: "Applying Model-Integrated Computing to Component Middleware and Enterprise Applications," *The Communications of the ACM* special issue on Enterprise Components, Service and Business Rules, Vol. 45, No. 10, October, 2002
- [36] The PVS Specification and Verification Systems, <http://pvs.csl.sri.com/>
- [37] Schmidt, D., Stal, M., Rohnert, H., Buschman, F.: *Pattern-Oriented Software Architecture*, Wiley, 2000