# Generative Development of Embedded Real-Time Systems*

**Gerd Frick** **Klaus D. Müller-Glaser**

**FZI Forschungszentrum Informatik**
**Dept. of Electronic Systems and Microsystems**
**Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany**
**esm@fzi.de**

## Abstract

Embedded real-time systems is a real-world application domain where state-of-the-art software development processes are already generative. Therefore it is a source of experience and also of additional technological requirements that may be of a general interest. After describing the special setting of Generative Programming in this area, the paper presents some lessons learned, open issues, and further directions. We conclude with a survey of the technology being under development at our site.

## 1 Introduction

According to [3], Generative Programming is a paradigm for engineering software system families such that, given a particular requirements specification, a customized and optimized system (a member of the family) can be automatically constructed from elementary, reusable implementation components by means of configuration knowledge. The essential elements are a *means of specifying* family members, the *implementation components*, and the *configuration knowledge* mapping from specification to implementation. The configuration knowledge is captured in program form, called a *generator*.

In other words, a *domain-specific language* (DSL) is used to describe a system on a high abstraction level, a compiler for this language translates this abstract system description into a more concrete description given in an object language in terms of a base library of abstractions. The base library consists of the predefined implementation components, the object language is the *composition language* which is necessary for constructing the system from the components and the existence of which is implicily assumed in the definition of Generative Programming rendered above.

In the case of a GUI builder, a well-known example kind of a program generator, the specification language is a visual drag&drop (together with some property dialogs) GUI design language, there is a library of GUI components (e. g. MFC, Swing, or Motif), and a general-purpose programming language (such as C++ or Java) is used for composition. Of course, the generated program is only one increment of the system (the user interface) and has to be completed by the application logic etc., which implementation will probably be done manually in the programming language which has been used for user interface composition.

It is not uncommon that the implementation components itself are implemented in terms of a programming language which is also used as the composition language. But this is not necessarily the case; the language for component implementation and composition may be completely different. There need not even be a library of components; the generated system may also be completely constructed in terms of the object language.

In the terminology of [3], generators, while translating a source representation of the system into a target representation, can perform *vertical* transformations (*refinements*) or *horizontal* transformations (*optimizations*) or both. Refinements transform a higher-level representation of a sys-

* ECOOP 2002 Workshop on Generative Programming, June 10, 2002, Malaga, Spain

tem into a lower-level one, preserving the general structure of the system description. Concepts of the higher abstraction level are implemented in terms of concepts on the lower abstraction level. Optimizations transform a representation of a system into an equivalent one on the same level of abstraction, changing the system structure to a more efficient one with respect to some implementation goal. Generators primarily based on vertical transformations are also called *compositional*, generators primarily based on horizontal transformations are also called *transformational*.

The construction of generators, especially transformational ones, can be supported by *transformation systems*. As surveyed by [3], transformation systems provide a common format for internal program representation, input and output facilities for the internal representation, and a transformation engine. Transformations are specified in a declarative or procedural language against the internal representation.

In the following sections we will shortly describe an application domain where generative software development is just maturing to become state-of-the-practice, some lessons learned, open issues and further directions, and requirements that we think should be met by technology supporting this engineering paradigm. We conclude with a survey of the technology being under development at our site.

## 2 Generative software development for embedded systems

Embedded electronic systems, or *embedded systems* for short, are a key technology for product innovations in many areas: automotive, aerospace, telecommunication, household and industrial applications—to mention some of the most important. For a wide range of applications the favorite way to realize an electronic system is a software-customized low-cost microprocessor (called microcontroller) taken 'of the shelf'. The system to be developed then essentially is a computer program.

The embedded system, being embedded, is interfaced to a physical environment by means of sensors (delivering input data to the system) and actuators (consuming output data of the system). The task of the system itself normally is to control the environment (in an open or closed loop). Control systems are a kind of *real-time* systems, i. e. the correctness of the input-output behaviour is determined not only by data but also by time. Thus, a notion of time is an integral part of the description of the behaviour of an embedded system.

Open loop control systems in general are reactive in nature, with a finite state space and discrete *events* triggering both *transitions* between the discrete *states* and output *actions*. Not only the inputs but also time is captured as events (timeout events). Closed loop control systems typically process input signals and generate output signals which are continuous in time[1]. Due to their primary nature, open and closed loop control systems are also (somewhat imprecisely) called *discrete* and *continuous* systems, respectively.

Visual formalisms (i. e. with graphical notations) have been used in control systems engineering for many years. The most well-known of these for discrete systems is the language of *Statecharts*, invented by Harel [8], which allows the description of finite state machines with a structured, hierarchical state space and corresponding transitions. Additional features are data variables, an internal event mechanism and transitions guarded by conditional expressions. For the description of continuous systems, it is very common to use *signal flow graphs* based on a collection of primitive 'blocks'—representing well-known functional dependencies between signals (e. g. derivative with respect to time)—as an abstraction from systems of differential equations. Both formalisms can be combined with techniques for hierarchical system decomposition.

These formalisms have lead to graphical *modeling languages* supported by tools (CASE[2] tools). They originated as means for constructing *model-based* (or *executable*) *specifications*. A formal description of the system, called a *model*, can—if constructive with respect to system behaviour, based on a certain model of computation—be executed as a *simulation* of the system. Thus, early validation of the system specification against the expected be-

[1] Within the computer system the signals of course have to be discretized, the value space to floating-point or fixed-point real numbers of finite precision, the time continuum by discrete time sampling.

[2] Computer Aided Software/Systems Engineering

haviour is possible. This is a tremendous advantage over validation of fully developed systems at the time of system integration, since it is a well-known fact that the costs of correcting errors are the higher the later they are detected.

An executable specification of a software system is a first prototype of the system itself. The virtual machine executing the prototype system is the simulator kernel running on the development computer. As a refinement of this, the model (or a refined program derived from it) may be executed on a dedicated rapid prototyping machine with hardware interfaces to the real environment. In contrast, the final system implementation will run on a small (e. g. 8 or 16 bit) and rather slow microcontroller with a small amount of memory.

The latter requirement has, up to recently, caused the necessity to implement the system by re-construcing the specified behaviour in a highly efficient way in a low-level programming language (normally C, having replaced assembly languages). In the meantime, endeavours to automate this process have lead to increasingly efficient *code generators*, which are starting to be used even for production purposes in the automotive industry. Thus, generative programming, known as 'automatic code generation' or 'autocoding', has become state-of-the-art for embedded systems software and is being accepted as a development principle.

The most prominent CASE tools used in the automotive industry are Statemate (the first tool implementing Statecharts) from I-Logix [9], Matlab (with its graphical languages Simulink and Stateflow) from The MathWorks [13], and ASCET-SD from ETAS [5]. Statemate is complemented by the tool Rhapsody in MicroC from I-Logix for target code generation; the Matlab suite can be complemented by the TargetLink code generator from dSPACE [4] or the recent RealTime Workshop Embedded Coder by The MathWorks; ASCET-SD includes a target code generator. UML [10], which contains an object-oriented variant of Statecharts, is also gaining some importance.

Code generators for discrete system models are typically compositional, refining the high-level control structure of Statecharts to standard imperative control structures. Code generators for continuous systems on the other hand are highly transformative, involving flattening of hierarchical signal-flow diagrams, mapping of functional dependencies to variables and update procedures, and expression folding, value caching and other optimizations. In both cases the specifications are implemented in terms of an imperative target language (C) and optionally also library calls to a real-time operating system (providing e. g. time and multi-tasking services).

## 3 Lessons learned, open issues, and further directions

First of all, it may be observed that generative software development using domain-specific means of describing systems and code generators to derive lower-level general purpose programming language code automatically has been accepted as and proven to be a useful principle in the domain of embedded systems design. Domain engineering is done mainly by a small number of CASE tool vendors, more or less specialized to specific application domains and equipping their tools with powerful domain-specific analysis features. The application (i. e. embedded system) developers are highly specialized software engineers, normally not affiliated with software houses but e. g. automotive manufacturers or suppliers.

An important success factor can be attributed to the fact that the domain-specific languages have not been developed specifically for the purpose of code generation (generative programming), but have been used years before for system simulation and analysis and the purpose of executable specifications. Code generation being available, the constructive specification languages become domain-specific programming languages. Building executable specifications essentially is programming on a higher abstraction level. C code generation is just an additional stage of the compilation process.

In should be noticed that the specification languages are formal languages (although partially with graphical syntaxes) with semantics on their own; the semantics are not given primarily by the implementation components and the corresponding generation processes. In consequence, it is possible to analyse and validate the system solely on the domain-specific abstraction level. The separation of specification semantics from the generation process allows the tool developers to optimize the generators independently. Nothing else is known from compilers for conventional programming languages. It may be mentioned by the way that the domain-specific means of specification should be treated as formal *languages*, not as *data models*.

The models of computation giving semantics to the various languages for describing discrete and continuous systems are different. In practice, heterogeneous systems containing both discrete and continuous parts do appear; the developers have to deal with more than one model of computation. Unless multi-paradigm languages supported directly by a single tool can be used, the development of heterogeneous systems raises a language and tool integration problem involving model transformations between different models of computation and/or different levels of abstraction. This problem is generally left open by the tool vendors, but solutions, at least for individual cases, are requested by the developers. That is the case for an independent transformation technology supporting translations between languages and tools. It is also the original use case which initiated the development of the technology described in section 4.

Up to know, generative development of embedded systems software is not yet fully integrated, leaving additional development steps for manual postprocessing. Examples for this are not only the integration of heterogeneous subsystems and the inclusion of real-time operating systems (for which integrated solutions are already available), but especially the realization of distributed systems and the implementation of respective communication behaviour. E. g. in automotive applications one has to deal with networks of tens of electronic control units, sensors and actors, linked by a CAN bus (or other communication systems), interchanging data, and together realizing a function network. Complexities have already reached a state that urges us to find new means of abstracting low-level physical communication behaviour and to integrate them into the semantics of the specification level, i. e. into the level of the input for code generation. Including aspects of distribution and communication into system specifications will enable us to enlarge the scope of the generators in order to eliminate manual coding on the target code level.

It seems to be useful to have separated notions of *functional* vs. *physical* models as different parts of a system description (like e. g. in UML or in Statemate). The functional model would describe the behaviour of the system whereas the physical model would describe the structure of the distributed components and the configuration of the communication system. A mapping between the two models, expressing which functionality is assigned to which physical parts, would complete the specification, i.e. the input to the code generator(s). But in a real-time setting where timing is a functional property and communication delays etc. influence the behaviour of a system, a clear separation of functionality from communication behaviour is not possible. It seems that at least some properties of the communication system will have to be propagated to the functional part of the model. Therefore the relationship between functional and physical model is not a simple mapping problem, there is a need for bi-directional integration of the two. Transformations are needed for refinements of and round-trips between the two views of the system.

For the introduction of physical modelling a general experience should be respected, gained from our automotive industry collaborations and contacts, i. e. that the integration of existing languages and tools is nearly always preferred over custom developments. Possible candidate languages that could be leveraged for physical modelling are UML and ROOM [12]. For the same reason, a fully integrated software development environment like Intentional Programming (see [3]) is not realistic to be successful in this domain, since established and powerful CASE tool environments would have to be replaced. What is needed is, in our view, a backend technology for representation and tranformation of formal system descriptions, interfaced to existing domain-specific languages and tools (as the interface to the user), and also open for experimental design and integration of new abstractions (for research and prototyping of new languages).

## 4 Overview of the XFL technology

XFL (Extensible Formal Language) is the name of a technology under development at FZI [7]. A detailed exposition of its concepts is beyond the scope of this paper, so in the following just some outlines are sketched.

XFL includes a declarative metalanguage based on a flexible type system for defining abstract syntax and semantics of formal languages [6], an XML [1] based encoding scheme providing a default concrete syntax for any language defined by the metalanguage, and an interpreter for the metalanguage manipulating target language phrases. Languages are formalized as collections of named (typed, polymorphic) abstracts. An abstract is either reducible (a *function*) or irreducible (a *constructor*). The abstractions are organized as libraries in a modular way allowing sublanguage sharing and language extensions. The composition language is as simple as the concept of *application* (term composition). In the concrete syntax, the named abstracts are applied as XML elements with the arguments being subelements. Thus, XFL leverages well-formed XML as a concrete syntax for terms. The module system leverages the XML namespace concept; the metalanguage is also XML-based and is integrated via a reserved namespace.

A translation of one set of abstracts (the source language) into another (the target language) can be defined by direct layering, i. e. the higher-level abstracts are functions reducible to the lower-level abstracts, which are constructors, or indirectly by transformation functions mapping constructors into constructors in any desired way. Thus, both vertical and horizontal transformations can be defined; the transfomations are executed by the interpreter of the metalanguage performing partial evaluation.

The XFL concept thus includes a system for XML-based (intermediate) representation of formal languages as well as a transformation system. XFL-represented languages are ordinary XML languages, which alleviates interfacing to tools or rendering via style sheets. Due to the metalanguage/interpreter approach of the transformation system there is no need for individual hard-coded generators or translators, only the essence of the transformations has to be specified declaratively.

The intended main purpose of XFL is the mediation between existing languages and tools as indicated at the end of the section 3. A previous version of the XFL system which included a subset of the above features has been successfully applied in the COMTESSA [2] project to the integration of Simulink with Statemate for heterogeneous systems modelling and to the realization of model exchange between both Simulink and Statemate and the Rodon tool [11] for model based diagnosis.

The open, library based approach (which is shared with Intentional Programming) and the XML syntax also allow for easy experimenting with new abstractions and corresponding generators. Language protoyping can be based on direct XML coding, configurable XML editors, or configurable and programmable graphical editors (like MS Visio or meta-CASE tools). Research and protoyping thus can prepare the development and improvement of domain-specific CASE technology.

## 5 Conclusion and proposal for worshop discussion

Control systems engineering has originated executable domain-specific languages for the description of real-time software systems. Generative technologies are used to automatically produce efficient target (C) code for execution on microcontrollers with limited resources. The inclusion of platform and design specific code related to operating and communication systems into automatic code generation requires additional abstractions and adds a high-level physical view to the functional view of specifications. Together with heterogeneity in different types of control systems supported by different domain-specific languages, the interaction of different languages in the description of a system appears as a general principle. Uni- or bidirectional transformations between different abstraction levels and languages can be supported by formal language representation and transformation systems like the XML-based XFL system.

Our proposed contribution to the workshop is to introduce the audience to an industrial success story of generative software development and to highlight and discuss

1. the problem of semantics of specification languages in generative programming, and
2. the relation between functional and physical system structures and its treatment in generative development.

## References

[1] Bray, T., Paoli, J., Sperberg-McQueen, C. M. (Eds.): Extensible Markup Lan-

guage (XML) 1.0. W3C Recommendation, http://www.w3.org/TR/1998/REC-xml-19980210, 1998.

[2] http://www.fzi.de/esm/projects/Comtessa/Comtessa_uk.html

[3] Czarnecki, K., Eisenecker, U. W.: Generative Programming—Methods, Tools, and Applications. Upper Saddle River, NJ, USA : Addison-Wesley, 2000.

[4] http://www.dspace.de

[5] http://www.etas.de

[6] Frick, G., Müller-Glaser, K. D.: A Type System for Language Definitions. (Submitted for publication in 2002.)

[7] http://www.fzi.de/esm

[8] Harel, D.: Statecharts—A visual formalism for complex systems. Science of Computer Programming, vol. 8, pp. 231-274, 1987.

[9] http://www.ilogix.com

[10] Kobryn, C. (ed.): OMG Unified Modeling Language Specification, Version 1.3. OMG Document ad/99-06-08, 1999.

[11] http://www.rose.de

[12] Selic, B., Gullekson, G., Ward, P. T.: Real-Time Object-Oriented Modeling. New York : Wiley, 1994.

[13] http://www.mathworks.com