20 The Model-Integrated Computing Tool Suite

Janos Sztipanovits, Gabor Karsai, Sandeep Neema, and Ted Bapty

ISIS - Vanderbilt University, Nashville TN 37203, USA

Abstract. Embedded system software development is challenging, owing to a tight integration of the software and its physical environment, profoundly impacting the software technology that can be applied for constructing embedded systems. Modeling and model-based design are central to capture all essential aspects of embedded systems. Vanderbilt University's Model Integrated Computing tool suite, driven by the recognition of the need for integrated systems and software modeling, provides a reusable infrastructure for model-based design of embedded systems. The suite includes metaprogrammable model-builder (GME), model-transformation engine (UDM/GReAT), tool-integration framework (OTIF), and design space exploration tool (DESERT). The application of the MIC tool suite in constructing a tool chain for Automotive Embedded System (VCP) is presented.

20.1 Introduction

The tight integration of the software and its physical environment has profound impact on the nature of the software technology to be applied for constructing embedded systems. The reason can be best explained by Brook's argument, which remains valid nearly 20 years after its publication: the essential complexity of large-scale software systems is in their *conceptual construct* [1]. In embedded systems, the conceptual construct of the software is combined with the conceptual construct of its physical environment; therefore, the methods and tools developed for managing complexity must include both the physical and computational sides. The common "denominator" for representing, relating and analyzing all essential aspects of embedded systems is *modeling* and *model-based design.*

The significance of modeling in software engineering has been recognized from the early nineties (see. e.g. Harel [2]). The recognition of the need for integrated system and software modeling led us to pursue the construction of a reusable infrastructure for model-based design. We have built several generations of tool environments since the late eighties directed to a wide range of application domains starting with signal processing [3]. Important milestones in the development of the Model-Integrated Computing (MIC) tool suite have been the following: (a) introduction of multiple-view modeling, programmable model builder and model-based integration of distributed applications in chemical process industry [4], (b) use of complex, multiple-view graphical modeling tool

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 369–376, 2010.
© Springer-Verlag Berlin Heidelberg 2010

with object-oriented database backend for the diagnosability analysis of the International Space Station design [5], (c) generation of high-performance parallel signal processing applications from models [6], and (d) use of embedded models for the structural adaptation of signal processing systems [7].

The technology advancements related to different applications led to the formulation of the MIC architecture concept [8] with domain-specific modeling languages (DSML) and modeling at its center. The main challenge has been the dichotomy between domain-specificity and reusabality. The construction of the meta-level tool architecture [9] [10] was followed by the first appearance of the metaprogrammable model builder [11] that ultimately led to the subsequent development of the metaprogrammable MIC tool suite [12].

20.2 Components of the MIC Tool Suite

Domain-specific modeling is at the center of the MIC development approach: domain-specific models are created in the design process, they are analyzed via formal and simulation-based analysis techniques, and they are used to construct and generate the implementation of applications, i.e. the actual software code that performs, for instance, control functions. The domain-specific models in an MIC engineering process are constructed in domain-specific modeling languages (DSML) whose syntax and semantics are precisely defined using metamodels and model transformations. However, MIC cannot exist without tooling: tools that assist the designer and developer in modeling, analysis, generation, evolution, and the maintenance of systems. This section provides a summary of the tools of MIC: the Generic Modeling Environment (GME), the Universal Data Model (UDM) package, the Graph Rewriting and Transformation language (GReAT), the Design Space Exploration Tool (DESERT), and the Open Tool Integration Framework (OTIF). These tools form a metaprogrammable tool suite that are connected by shared meta models as shown in Fig. 20.1. The tools can be extended by 'best of breed' analysis, simulation and verification tools connected via model transformation to build domain-specific toolchains.

20.2.1 The Generic Modeling Environment (GME)

GME is the core MIC tool [13] that is used for both meta-modeling and modeling. GME is metaprogrammable: it can load metaprograms generated from metamodels and "morph" itself into a domain-specific modeling environment. GME is primarily a visual modeling tool (although textual model elements are also supported). GME is equipped with a metaprogram that configures it to behave as the metamodeling tool: it understands a UML-like notation (the metamodeling language), and an associated translator program can generate the metaprogram from the metamodel. The GME metamodeling approach is based on the use of stylized UML class diagrams and Object Constraint Language (OCL) constraints [14]. These metamodels capture the abstract syntax and well-formedness rules of the modeling language. Abstract syntax defines the set of

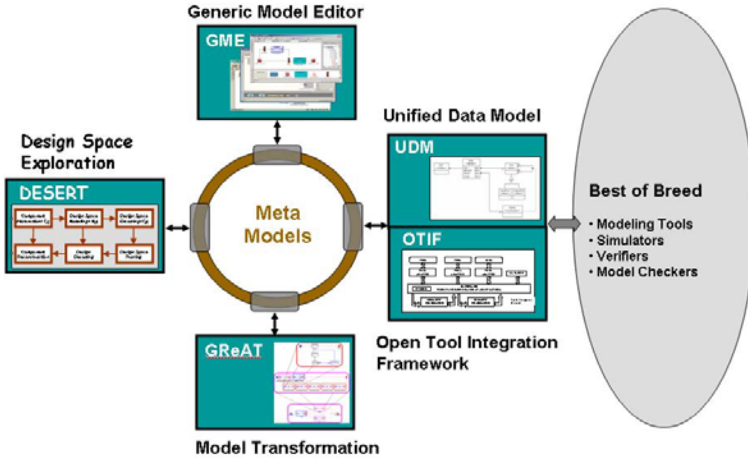

Fig. 20.1. The MIC Tool Suite

concepts, their attributes, and relationships one can use for building models in the language. For example, in a control system design language that supports event-driven control, the abstract syntax includes concepts like "states", "events", and "finite state machine", etc., relationships like "transitions", and attributes like "guard expression", "initial state", etc. The well-formedness rules of a language formally describe the constraints that the models need to satisfy in order to be syntactically correct.

20.2.2 Transforming the Models: UDM and GReAT

GME is a general purpose modeling environment, and it provides a set of Application Programming Interfaces (API-s) to access models. These low-level programmatic interfaces allow building software tools using traditional languages that access and possibly manipulate models. As a higher-level, more formal alternative to the API-s we have created tools that allow structured access to models on one hand, and allow the transformation of those models into other kinds of objects on the other hand. The first step is facilitated by the tool called "Universal Data Model" (UDM), and the second is done using the "Graph Rewriting and Transformation" (GReAT) language.

UDM is a metaprogrammable software tool [15] that generates domain-specific classes and API-s to access the models within GME, in XML form, in ODBC-based data-bases. The advantage of using UDM is that tools that access models could be developed using the concepts from the domain-specific modeling language (e.g. "Assembly", and "TimeTrigger"), instead of the generic concepts of GME. GReAT is a (graphical) modeling language for describing transformations on models [16]. The transformation specification is built upon metamodels of the input and the target of the transformations, and is expressed with the help of sequenced rewriting (or transformation) rules. The key point here is that both

the input and the target must have a defined metamodel (i.e. abstract syntax with well-formedness rules). Often target models use some lower level modeling language, like a modeling language of simple finite transition systems. Note that in the ultimate, a target metamodel may represent the instruction set of a (real or virtual) machine. In practice, target metamodels often consist of concepts that correspond to code patterns (e.g. while-loop) that are instantiated with the values of attributes of the concept instances.

20.2.3 Integrating Design Tools: The Open Tool Integration Framework

The MIC toolsuite is often used to build not only a single tool, but tool chains consisting of various modeling, analysis, and generation tools, where many tools could be non-MIC tools. In this case one faces a tool integration problem: namely, how to construct integrated tool chains from tools that were not designed to work together. The MIC toolsuite includes a framework, called Open Tool Integration Framework (OTIF) that supports the construction of such integrated tool chains [17]. The tool integration problem that OTIF provides a tool for is as follows. In an engineering workflow various design tools are used, and the data ("models") need to be exchanged between the design tools. Each design tool has its own format (i.e. DSML) for storing models. The workflow implicitly specifies an ordering among the tools, and the direction of "model flow" defines the producer/consumer relationships between specific pairs of tools. We also assume that models are available in a packaged, "batch" form. OTIF provides a skeleton architecture for building tool chains that follow this model. OTIF has been implemented as a set of components, some of which are metaprogrammable, and it relies on the UDM and GReAT tools. It has been used to construct a number of tool chains consisting of MIC and other tools.

20.2.4 Design Space Exploration

When large-scale systems are constructed, in the early design phases it is often unclear what implementation choices could be used in order to achieve the required performance. In embedded systems, frequently multiple implementations are available for components (e.g. software on a general purpose processor, software on a DSP, FPGA, or an ASIC), and it is not obvious how to make a choice, if the number of components is large. Another meta-programmable MIC tool can assist in this process. This tool is called DESERT (for Design Space Exploration Tool) [18]. DESERT expects that the DSML allows the expression of alternatives for components in a complex model. A model, with hierarchically layers alternatives, defines a design space. Once a design spaces is modeled, one can attach applicability conditions to the design alternatives. These conditions are symbolic logical expressions that express when a particular alternative is to be chosen. Conditions could also link alternatives in different components via implication. One example for this feature is: "if alternative A is chosen in component C1, then alternative X must be chosen in component C2". During the design

process, engineers want to evaluate alternative designs, which are constrained by high-level design parameters like latency, jitter, power consumption, etc. DESERT provides an environment in which the design space can be rapidly pruned by applying the constraints, thereby restricting the applicable aletrnatives.

20.3 Application Example: Vehicle Control Platform

The MIC tools have been used in numerous projects, and various tool chains have been constructed using it. Application experience includes large systems engineering projects such as the International Space Station diagnosability analysis [5], automotive manufacturing execution systems deployed in major plants [19] and prototypes for integrated design environments [12]. In this section we describe a tool chain that illustrates how the metaprogrammable tools have been used to solve the construction and integration of non-trivial tool architecture for embedded control applications. This is called the Vehicle Control Platform (VCP) tool chain [20] and it was built for constructing vehicle control software. The design flow starts with specifying controller components in the form of behavioral models, using Simulink/Stateflow.

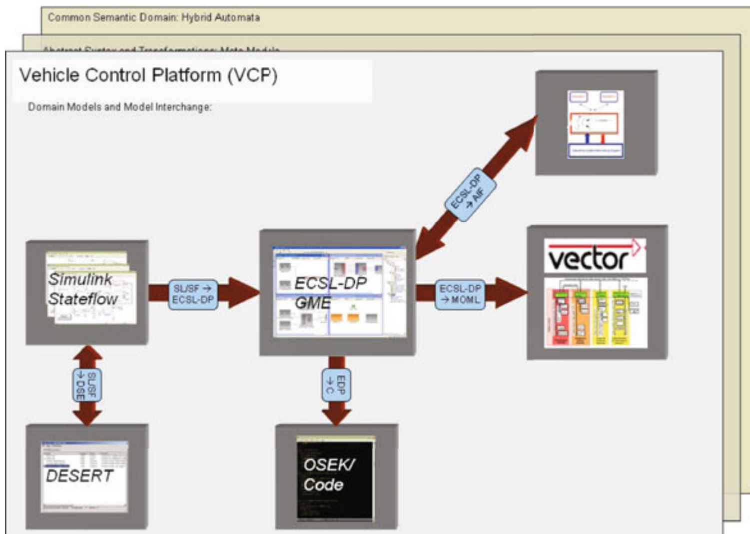

Fig. 20.2. The VCP Tool Suite

This step primarily consists of building up a library of controller blocks. The next step is design space modeling, which happens in a DSML called ECSL-DP, and which is supported by GME. During the construction of the design space models, the designer constructs hierarchical designs for the controllers, with possible alternative implementations on various levels of the hierarchy. The designer

specifies component structures and component interactions. Note that the elementary components are from the behavioral models built in the first step. Once the design space modeling is finished, the designer can explore alternative designs with the help of DESERT. This stage will result in specific point design(s) that satisfy all design parameters. The specific designs are also captured in ECSL-DP. ECSL-DP has provisions for mapping designs into distributed electronic control units (ECU-s) and buses in the vehicle, and this mapping is specified in the models. Once the design models are finished a number of analysis steps can take place. Here we mention two: one can perform a schedulability analysis using a tool called AIRES [21], or one can perform a behavioral simulation using tools from the Vector toolsuite [22]. Note that this behavioral simulation on the ECSL-DP models is an alternative to the simulation of Simulink/Stateflow models and it can potentially be more accurate because of the finer details ECSL-DP captures. The result of these analyses (e.g. end-to-end latency in the system) can be annotated back into the ECSL-DP models. Finally, executable code (in C) is generated that runs on the real-time operating system OSEK. Figure 20.2 shows the high-level architecture and workflow in the tool chain. For the tool chain we have built the ECSL-DP modeling tool using GME, created various tool adaptors, and built a number of model transformation tools using GReAT. The five model translators contain, on average, 50 transformation rules, and process practical models with acceptable speed: 1-2 minutes, maximum. These model translators are automatically invoked as and when they are needed by OTIF.

20.4 Conclusion

In this paper we have introduced and briefly described the metaprogrammable toolsuite for MIC. We showed the evolution of the MIC tool suite and its use in a wide range of domain-specific tool chains supporting complex design flows. In each stages of a design flow, the actual state of the design is expressed using a DSML. These languages comprise the required heterogeneous abstractions for expressing controller dynamics, software and system architecture, component behavior, and deployment. The models expressed in these DSMLs need to be precisely related to each other via the specification/implementation interfaces, need to be analyzable and their fidelity need to be sufficiently precise to accurately predict the behavior of the implemented embedded controller. In addition, the design flow is supported by heterogeneous tools including modeling tools, formal verification tools, simulators, test generators, language design tools, code generators, debuggers, and performance analysis tools must all cooperate to assist developers and engineers struggling to construct the required systems. If the DSMLs are only informally specified then mismatched tool semantics may introduce mismatched interpretations of requirements, models and analysis results. This is particularly problematic in the safety critical real-time and embedded systems domain, where semantic ambiguities may produce conflicting results across different tools. Our current efforts focus on the formal, transformational specification of structural [23] and behavioral [24] semantics for DSMLs.

References

[1] Brooks Jr., F.P.: No silver bullet: Essence and accidents of software engineering. IEEE Computer Magazine, 10–19 (April 1987)

[2] Harel, D.: Biting the silver bullet. IEEE Computer Magazine, 8–19 (January 1992)

[3] Sztipanovits, J., Karsai, G., Biegl, C.: Graph model based approach to the representation, interpretation and execution of real time signal processing systems. International Journal of Intelligent Systems 3(3), 269–280 (1988)

[4] Karsai, G., Sztipanovits, J., Franke, H., Padalkar, S., DeCaria, F.: Model-embedded on-line problem solving environment for chemical engineering. In: Proceedings of the International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, FL, pp. 361–368 (November 1995)

[5] Misra, A., Sztipanovits, J., Underbrik, A., Carnes, R., Purves, B.: Diagnosability of dynamical systems. In: Third Intenational Workshop on Principles of Diagnosis, Rosario, Orcas Island, WA, pp. 239–244 (May 1992)

[6] Abbott, B., Bapty, T., Biegl, C., Karsai, G., Sztipanovits, J.: Model-based software synthesis. IEEE Software, 42–53 (May 1993)

[7] Sztipanovits, J., Wilkes, D., Karsai, G., Lynd, L.: The multigraph and structural adaptivity. IEEE Transaction on Signal Processing 41(8), 2695–2716 (1993)

[8] Sztipanovits, J., Karsai, G., Biegl, C., Bapty, T., Ledeczi, A., Malloy, D.: Multigraph: An architecture for model-integrated computing. In: Proceedings of the International Conference on Engineering of Complex Computer Systems, Ft. Lauderdale, FL, pp. 361–368 (November 1995)

[9] Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE Computer 22(5), 110–112 (1997)

[10] Nordstrom, G., Sztipanovits, J., Karsai, G.: Meta-level extension of the multigraph architecture. In: Engineering of Computer-Based Systems Conference, Jerusalem, Israel, pp. 61–68 (May 1998)

[11] Nordstrom, G., Sztipanovits, J., Karsai, G., Ledeczi, A.: Metamodeling - rapid design and evolution of domain-specific modeling environments. In: Proceedings of the IEEE ECBS 1999 Conference, Nashville, TN, pp. 68–74 (April 1999)

[12] ISIS: Mic tool distribution

[13] Ledeczi, A., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J.: Composing domain-specific design environments. IEEE Computer Magazine, 44–51 (November 1997)

[14] Object Management Group: UML 2.0 OCL Specification (2003)

[15] Bakay, A.: The udm framework

[16] Karsai, G., Agrawal, A., Shi, F.: On the use of graph transformations for the formal specification of model interpreters. Journal of Universal Computer Science 9(11), 1296–1321 (2003)

[17] Karsai, G., Lang, A., Neema, S.: Design patterns for open tool integration. Journal of Software and System Modeling 4(1) (2004)

[18] Neema, S., Sztipanovits, J., Karsai, G., Butts, K.: Constraint-based design space exploration and model synthesis. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 290–305. Springer, Heidelberg (2003)

[19] Earl, L., Amit, M., Janos, S.: Increasing productivity at saturn. IEEE Computer Magazine, 35–44 (August 1998)

[20] Porter, J., Karsai, G., Volgyesi, P., Nine, H., Humke, P., Hemingway, G., Thibodeaux, R., Sztipanovits, J.: Towards model-based integration of tool and techniques for embedded control system design, verification, and implementation. In: Chaudron, M.R.V. (ed.) Models in Software Engineering. LNCS, vol. 5421, pp. 20–34. Springer, Heidelberg (2009)
[21] Zonghua, G., Wang, S., Kodase, S., Shin, G.K.: An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software. In: 24th IEEE International Real-Time Systems Symposium (RTSS 2003), Cancun, Mexico (September 2003)
[22] Vector Informatik Group: The vector tools
[23] Jackson, E., Sztipanovits, J.: Formalizing the structural semantics of domain-specific modeling languages. Journal of Software and Systems Modeling (2009) (to appear)
[24] Chen, K., Sztipanovits, J., Neema, S.: Compositional specification of behavioral semantics. In: Design, Automation, and Test in Europe: The Most Influential Papers of 10 Years DATE, pp. 253–256 (April 2008)