Proceedings of the 2006 IEEE
Conference on Computer Aided Control Systems Design
Munich, Germany, October 4-6, 2006

The Model-Integrated Computing Toolsuite: Metaprogrammable Tools for Embedded Control System Design

Gabor Karsai, *Senior Member, IEEE* Akos Ledeczi, *Member, IEEE,* Sandeep Neema, *Member, IEEE*
Janos Sztipanovits *Member, IEEE Fellow*

Abstract—**Model-Integrated Computing is a development approach that advocates the use of Domain-Specific Modeling throughout the system development process and lifecycle. This paper describes and summarizes the generic and reusable software tools that support MIC and which can be tailored to solve a wide variety of modeling, analysis, and generation problems in an engineering process.**

I. Introduction

MODELS and modeling are at the center of many (if not all) engineering disciplines, so it is somewhat surprising that their relevance has only recently been recognized in software development [1]. For control engineers it is natural to think in terms of mathematical and physical models of systems that they control, and the (software) implementation of control applications can often be "naturally" derived from these models. A number of software tools are available on the commercial market [2][3] that directly support the modeling, simulation, and eventual construction of control applications using models and modeling languages that are close to the notation and language used by control engineers. These notations are then used in generating executable code that implements the control functions.

However, control applications often necessitate a modeling language that is closer to the application domain (than the abstract, mathematical language of classical control theory), or the use of novel heterogeneous modeling frameworks (e.g. hybrid systems [4]) for which modeling tools are not readily available. Often a single modeling approach is not sufficient to represent all aspects in a project, and occasionally completely new approaches are needed. All these factors provide a motivation for tools that allow the use of a wide variety of modeling approaches in an integrated, coherent framework.

The Model-Integrated Computing (MIC) approach [5] and its supporting toolsuite offer a *conceptual* and *software* infrastructure for addressing these problems in a simple, coherent form that could potentially serve as the basis for various, domain-specific engineering processes and tool chains [6]. The domain-specificity of MIC means that tools are tailored to the practice and needs of specific engineering disciplines. For instance, in MIC a modeling tool used in building flight control systems might be different from a modeling tool used in designing control systems for discrete manufacturing plants.

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. In the rest of the paper first we introduce 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). We also discuss how these tools could be used to define syntax and semantics of modeling languages, and finally we describe one from the many domain-specific tool chains that we have built using them. This paper provides a summary of the tools and shows how they fit together to form a tool 'meta-architecture' that can be used to build other toolchains. As the focus here is on overview and integration via meta-programmability, the reader is kindly requested to consult the references for more details about the tools.

II. The Metaprogrammable Tools of MIC

A. Metamodeling

Strictly speaking, the tools in the core MIC toolsuite are not tools but meta-tools. By this we mean that they can be customized (effectively "programmed") to suite the needs of specific domains, and thus after this customization they become (domain-specific) tools. We enact the process of customization through metamodeling, which yields a "model of the modeling language", i.e. a metamodel. This metamodel is then turned (automatically) into a metaprogram that is

Manuscript received February 10, 2006. The DARPA/IXO MOBIES program and USAF/AFRL under contract F30602-00-1-0580, and the NSF ITR on "Foundations of Hybrid and Embedded Software Systems" have supported in part, the activities described in this paper.

Gabor Karsai is with the Institute for Software-Integrated Systems at Vanderbilt University, Nashville, TN 37235, USA (phone: 615 343-7471, fax: 615-343-7440; e-mail: gabor.karsai@vanderbilt.edu).

Akos Ledeczi, Sandeep Neema, and Janos Sztipanovits are with the Institute for Software-Integrated Systems at Vanderbilt University, Nashville, TN 37235, USA.

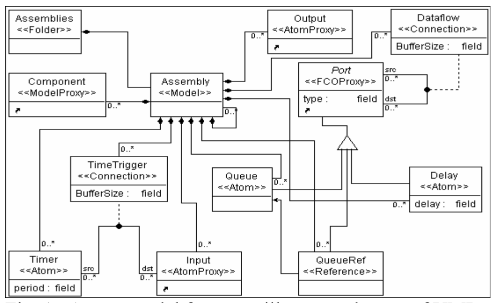

Fig. 1. A metamodel fragment illustrates the use of UML

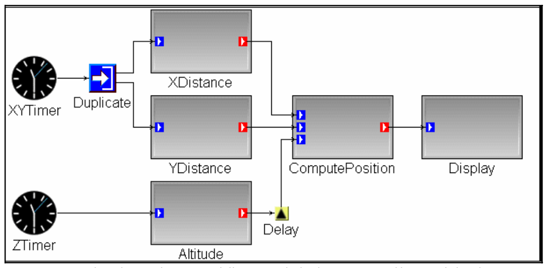

Fig. 2. Example domain-specific model that complies with the metamodel on Fig.1.

incorporated into a generic tool, which is thus turned into a domain-specific tool. We have recognized the fact the metamodeling is just another form of (domain-specific) modeling, just with the metamodeling language serving as a modeling language. In the MIC tools —for pragmatic reasons— we use (an extended subset of) the Unified Modeling Language (UML) [7] for metamodeling. Note that this use is somewhat atypical: we do *not* use UML to create domain-specific models; rather we use UML to define a modeling language which is then used to create the domain-specific models. This way, the MIC process is heavily language-based, and the syntax and semantics of the modeling language is precisely defined.

B. *The Generic Modeling Environment (GME)*

GME is *the* core MIC tool [8] that is used for both metamodeling 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 generated 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 [9]. These metamodels capture the abstract syntax and well-formedness rules of the modeling language. Abstract syntax defines the set of concepts, their attributes, and relationships one can build models from 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 correct (with respect to a static property). Continuing with the example, a well-formedness rule could be as follows: "Each finite state machine must contain precisely one state whose 'initial state' attribute must have the value true.

Obviously, the abstract syntax and the well-formedness rules are insufficient to fully define a modeling language, as they do not specify the concrete syntax, i.e. how the models are visualized. This problem was solved in GME through the use of stereotypes on the metamodel elements. Stereotypes are denoted by names between markers "<<" and ">>", and they determine the visualization of the model objects on the drawing editor. For instance, "<<Atom>>"-s are always shown as an icon, "<<Connection>>"-s are lines, and "<<Model>>"-s are containers visualized either as complex drawings (when one looks at the details) or rectangular icons with ports (when one looks at the container at a higher level of abstraction).

Note that the metamodel controls what can be present in a model, what can be connected to what, etc. For example, the metamodel of Fig.1 specifies that an assembly may contain timers and inputs, and the first can be connected to the second. Fig.2 shows a model that contains timers (`XYTimer` and `ZTimer`) and components which have inputs (like `Altitude`), `ZTimer` is connected to the input on `Altitude` component. However, timers cannot be connected to outputs (as this is missing from the metamodel) — if such a connection is attempted, the domain-specific GME will refuse it. Similarly, if models are created that violate the well-formedness constraints, the user is warned about the problem.

Once GME models are created, they are correct with respect to the abstract syntax and the well-formedness rules. However, the GME metamodel does not assign a specific interpretation to the models, i.e. the dynamic semantics of models is not specified. For pragmatic reasons, we follow a transformation-based approach for specifying dynamic semantics, as discussed later. Next, we describe how model transformations can be specified using metaprogrammable tools.

C. *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 [10] that generates domain-specific classes and API-s to access the models within GME, in XML form, in ODBC-based databases. 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 [11]. 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.

GReAT uses a mixture of declarative and imperative constructs to define transformations on models [12]. Note that the transformation is expressed with the help of metamodel elements, and the transformation is applicable to models that comply with the metamodels. GReAT uses three, layered languages for describing transformations, illustrated on Fig.3.

First, GReAT has a language for describing patterns of model objects and graphs (top box, with the two `OrState` and one `State` objects and links between them). This language allows the specification of graph patterns, where each vertex in the pattern corresponds to a model object and is typed with a corresponding metamodel element (i.e. a UML class), and where each edge corresponds to a link among model objects and is typed with a corresponding metamodel element (i.e. a UML association). These patterns give a concise description of a complex structure that will be searched for in the input of the model transformation.

Second, GReAT has a language for describing graph rewriting and transformation rules. The rules contain a graph pattern (describe using the pattern language), optional new graph elements (typed nodes and edges), an optional Boolean-valued expression called the guard, an optional code block called the attribute mapping block, and have input and output ports denoting input and output parameters of the rule. Furthermore, some of the nodes in the graph pattern could be marked with a "delete" marker.

A rule has an executable semantics as follows. Before the rule is executed its input ports are bound to specific nodes in the object graph(s) the rule is operating on. The binding of these ports must be type-correct, i.e. on the inside of the rule the port should be connected to a pattern node of a compatible type. Next, a pattern graph matching process is invoked

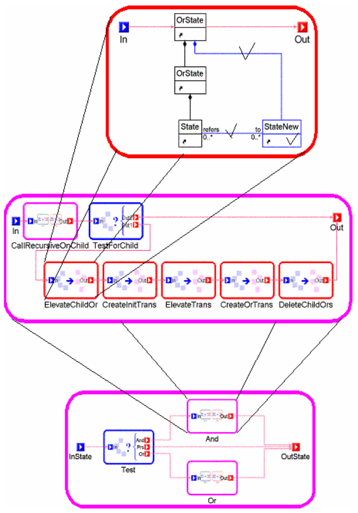

Fig 3. The layered languages of GReAT

that finds (a) matching sub-graph(s) given the initial binding and the topology of the pattern graph. If a match is found, all pattern nodes and edges will be bound to specific elements from the graph and the rule execution proceeds (otherwise stops). Next, the optional guard expression is evaluated. This expression may access attributes of objects bound to the pattern graph elements, for example. If the guard expression evaluates to true (or if it is absent), then the rule execution proceeds, otherwise stops. Next, pattern elements marked with "delete" will be removed from the graph. Next, the new graph elements: nodes (class objects) and edges (object links) are created. Next, the optional attribute mapping code is executed that could, for instance, read and write the attribute values of objects and links currently accessible. Finally, the output ports of the rule are bound to objects they are connected to within the context of the rule. Note that these objects could be elements in the pattern or newly created. On Fig.3. in the upper box, the rule has one input port of type `OrState`. When the rule is executed, the rule finds another `OrState` that is contained within the first one, and which also has a `State`. After matching, it creates a `StateNew` under the top-level `OrState` and links it to the bottom-most `State` object. Finally, the top-level `OrState` is bound to the output port. The third language in GReAT is the rule sequencing language which specifies the order of execution for the transformation rules. This is shown in the middle and bottom box of Fig.3. Rules are sequenced by connecting their input and output ports: the direction of connection implies the execution sequence. Rule sequences can be grouped into blocks that imply transformation steps that are applied as one unit. Furthermore, constructs for conditional execution and recursion are available that allow data-driven control structures as well as iterative constructs. GReAT has a formally defined semantics that was created using the Z-Object notation [11].

In summary, GReAT provides a formal and visual way of describing complex rewriting and transformation operations on models. One key aspect of the language is that because the structural operations on graphs are formally specified, mathematical reasoning can be applied when analyzing transformation programs.

D. *Formally Specifying the Dynamic Semantics of Domain-Specific Modeling Languages*

GReAT has been used to build a number of model transformation tools, including template-based code generators. However, it is also suitable for specifying the dynamics semantics of DSML-s using a GReAT transformation as the specification. We call this approach "Semantic Anchoring" (SA) [13].

The key idea for specifying semantics is as follows. Suppose we have well-defined, accepted, and well-understood modeling languages whose semantics is simple and defined in a non-ambiguous, preferably executable way. Let's call these core modeling languages as "semantic units". One such semantic unit could be, for example, the language of simple finite state machines. Now if one defines a new DSML, one has to define the semantics of this new language by showing how the models built in the new language could be reduced to (or transformed into) the well-defined semantic units. The principle is illustrated on Fig.4.

For practical purposes, a tool has been created that uses the Abstract State Machine Language (ASML) [14] to specify semantic units. ASML allows building these semantics units using the Abstract State Machine concepts [15], i.e. essentially as transition systems with sophisticated data structures representing the state of the system. A number of prototype model transformations have been built that show how a non-trivial DSML (like, for example, a Statechart-like language) can be formally defined through the transformation. These form the beginnings of an SA Toolsuite where one can define the metamodel of a language, together with its semantics using semantic units and transformations. The interesting part is that ASML is executable, thus one can rapidly prototype and experiment with DSML-s, by executing their models as ASML "programs".

E. *Model management via 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) [16]. DESERT expects that the DSML used allows the expression of alternatives for components in a complex model. Fig.5. provides an illustrative example.

In the example, the `Correlation` component has two alternatives called `SpectralCorrelation` and `SpatialCorrelation`, respectively. Note that if models are built using these "alternative" constructs the models do not represent point designs anymore, rather *sets* of point designs. If hierarchy is allowed (i.e. alternatives could contain alternatives, etc), a very large design space could be represented

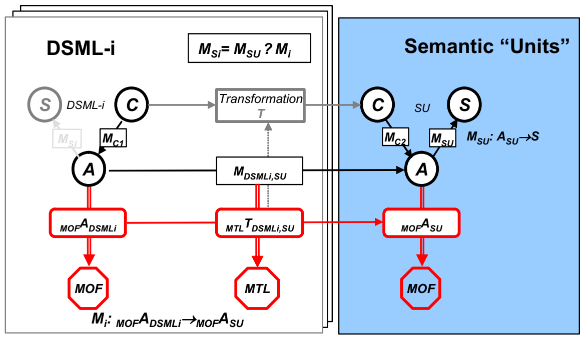

Fig. 4. Illustration of defining the semantics of a DSML (DSML-i) via a transformation (M_DSMLi,SU) and a well-defined semantic unit (SU).

using a compact model.

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 pruned by applying the constraints, and alternatives rapidly generated and evaluated.

DESERT consumes two types of models: component models (which are abstract models of simple components) and design space models (that contain alternatives). Note that for design space exploration the internals of simple components are not interesting, only a "skeleton" of these components is needed. DESERT uses a symbolic encoding technique to represent the design space that is based on Ordered Binary Decision Diagrams (OBDD-s) [19]. OBDD-s are also used to represent applicability constraints, as well as design parameters.

Once the symbolic representation is constructed, the designer can select which design parameters to apply to the design space and thus "prune away" unsuitable choices and alternatives.

This pruning is controlled by the designer, but it is done on the symbolic representation: the OBDD structures. Once the pruning is finished, the designer is left with zero, one, or more than one designs, indicating that no combination of choices could satisfy the design parameters, a single solution is available, or more than one means multiple alternatives are available that cannot be further pruned based on the available information. This latter case often means that other methods must be used to evaluate the alternatives, e.g. simulation. The result of the pruning is in symbolic form, but it can be easily decoded and one (or more) appropriate (hierarchical) model structure reconstructed from the result.

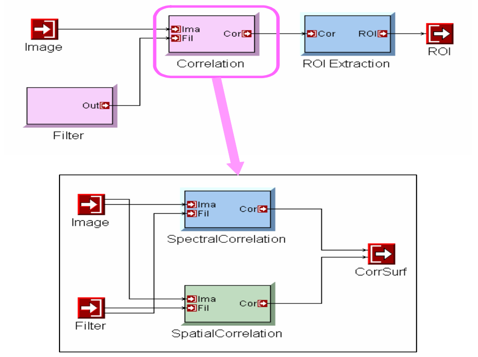

Fig. 5. Modeling design alternatives.

DESERT is metaprogrammable as it can be configured to work with various DSML-s (however all of them should be capable of representing alternatives and constraints). The metaprogramming here happens by providing the two left-side stages on the DESERT workflow diagram: one for the model skeleton generation, and the other one for the model reconstruction.

DESERT has been used to implement domain-specific design space exploration tools for embedded control systems and embedded signal processing systems.

F. 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. The skeleton architecture is shown on Fig. 6.

OTIF is based on the requirement that models need to be translated from the format of one tool to the format of another tool. However, in addition to the translation, the workflow must also be enacted. An OTIF tool chain consist of the tools that form the elements of the tool chains, tool adaptors (TA) that couple tools to the framework, a backplane (BP) component that facilitates the data- and workflow, and a set of semantic translators (ST) that perform the model transformations.

OTIF decomposes the translation process into three phases: syntactical transformation of the source models into a canonical form (that can be manipulated by a model transformation program implemented in GReAT), the semantic translation that focuses on the semantic issues (and is implemented using GReAT), and another syntactical transformation from the canonical form of the models into some physical form that the destination tool can directly understand. This decoupling of the syntactical and semantic transformations makes possible the use of generic tools and isolating the transformation process from the physical details of model representations.

The TA, ST, and BP components can run on different hosts in a network and communicate via CORBA calls and interfaces. The TA is responsible for interacting with the tool: it reads the tool's data converts it into the canonical form and sends it to the BP, as well as it receives data in canonical form from the BP and converts it into the physical form as needed by the tool. The ST implements the point-to-point translation on the models coming from a source tool and to be sent to a target tool. It receives and sends models in the canonical form from and to the BP, and performs the transformation in between. The BP incorporates a workflow engine that enacts the workflow and routes the messages

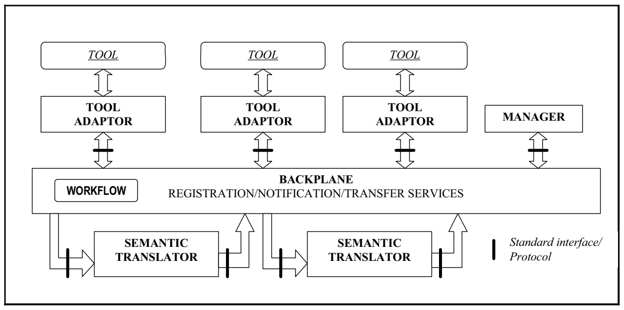

Fig. 6. Skeleton architecture of a toolchain using OTIF.

between the components. BP acts as a server, but it could be controlled through a management tool.

In this framework the BP, the skeletons for the TA-s and the ST-s are reusable across tool chains. However, BP and the skeletons are metaprogrammable: they are configured through metamodels of the DSML-s of the tools.

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.

III. EXAMPLE: THE VEHICLE CONTROL PLATFORM (VCP) TOOLCHAIN

The MIC tools have been used in numerous projects, and various tool chains have been constructed using it. In this section we describe a tool chain that illustrates how the metaprogrammable tools could be used to solve the construction and integration of non-trivial tool architecture for embedded control applications. This is called the Vehicle Control Platform tool chain and it was built for constructing automotive control software [18].

The purpose of VCP is to provide a tool suite for the model-based development of controllers in automotive applications. The design flow starts with specifying controller components in the form of behavioral models, using Simulink/Stateflow. 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.

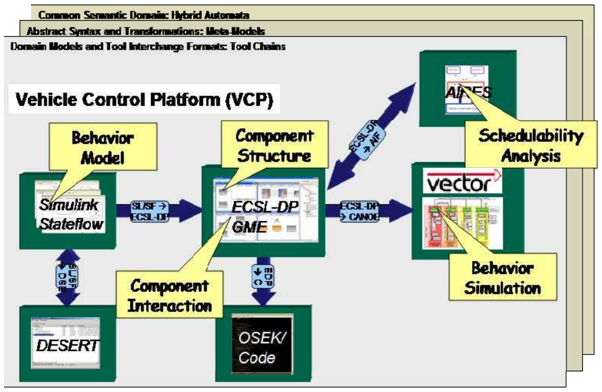

Fig. 7. The VCP tool chain.

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 [20], or one can perform a behavioral simulation using tools from the Vector toolsuite [21]. 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 a real-time operating system OSEK. Fig.7. 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.

IV. CONCLUSIONS

In this paper we have introduced and briefly described the metaprogrammable toolsuite for Model-Integrated Computing (MIC). We showed how the MIC tools could be configured through metamodels, how metamodels could be used to define new domain-specific modeling languages, how their semantics could be formally specified (such that the resulting specification is executable), how model transformations can be specified using a formal technique, how large design spaces can be rapidly explored evaluated, and how heterogeneous design tools can be integrated using metamodels.

REFERENCES

[1] Model-Driven Architecture http://www.omg.org/mda/
[2] Matrix-X tools http://www.ni.com/matrixx/
[3] Matlab, Simulink and Stateflow tools http://www.mathworks.com
[4] Henzinger, T.A.: The Theory of Hybrid Automata. In Proc. of IEEE Symposium on Logic in Computer Science (LICS'96), pages 278--292. IEEE Press, 1996.
[5] Karsai, G.; Sztipanovits, J.; Ledeczi, A.; Bapty, T.: Model-integrated development of embedded software, Proceedings of the IEEE, Volume: 91, Issue:1, Jan. 2003 Pages:145 – 164
[6] Agrawal A., Karsai G., Ledeczi A.: An End-to-End Domain-Driven Software Development Framework, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Domain-Driven Development Track, Anaheim, CA, October, 2003.
[7] The UML documentation, http://www.uml.org/#UML2.0
[8] Ledeczi, A.; Bakay, A.; Maroti, M.; Volgyesi, P.; Nordstrom, G.; Sprinkle, J.; Karsai, G.: Composing domain-specific design environments, IEEE Computer, Nov. 2001, Page(s): 44 –51.
[9] Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.
[10] A. Bakay, "The UDM Framework," http://www.isis.vanderbilt.edu/Projects/mobies/.
[11] Karsai, G., Agarwal, A., Shi, F., Sprinkle, J. On the Use of Graph Transformation in the Formal Specification of Model Interpreters, Journal of Universal Computer Science, Volume 9, Issue 11, 2003.
[12] G. Karsai, Aditya Agrawal, Sandeep Neema, Feng Shi, Attila Vizhanyo: "The Design of a Simple Language for Graph Transformations", in review for Journal on Software and System Modeling.
[13] K. Chen, J. Sztipanovits, S. Neema, M. Emerson and S.Abdelwahed, "Toward a Semantic Anchoring Infrastructure for Domain-Specific Modeling Languages," 5th ACM International Conference on Embedded Software (EMSOFT'05), 2005, in press.
[14] The Abstract State Machine Language, http://research.microsoft.com/fse/asml/
[15] Börger, Egon, Stärk, Robert : Abstract State Machines : A Method for High-Level System Design and Analysis, Springer, 2003.
[16] Neema S., Sztipanovits J., Karsai G., Butts, K.: Constraint-Based Design Space Exploration and Model Synthesis, Lecture Notes in Computer Science on EMSOFT 2003.
[17] Karsai, G., Lang, A., Neema, S.: Design Patterns for Open Tool Integration, Vol 4. No1, DOI: 10.1007/s10270-004-0073-y, Journal of Software and System Modeling, 2004.
[18] Neema, S. and Karsai, G.: Software for Automotive Systems: Model-Integrated Computing, to appear in LNCS volume on Automotive Software System Development Workshop, San Diego, 2004.
[19] Bryant R., "Symbolic Manipulation with Ordered Binary Decision Diagrams," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-92-160, July 1992
[20] Zonghua Gu, Shige Wang, Sharath Kodase, and Kang G. Shin: An end-to-end tool chain for multi-view modeling and analysis of avionics mission computing software, 24th IEEE International Real-Time Systems Symposium (RTSS 2003), Cancun, Mexico.
[21] Vector Informatik Group, http://www.vector-informatik.de/