Abstractions for Modeling Complex Systems

Zsolt Lattmann, Tamás Kecskés, Patrik Meijer,
Gábor Karsai, Péter Völgyesi, and Ákos Lédeczi

Vanderbilt University, Institute for Software Integrated Systems
Nashville, TN 37212, USA
{zsolt.lattmann,tamas.kecskes,patrik.meijer,
gabor.karsai,peter.volgyesi,akos.ledeczi}@vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract. The ever increasing popularity of model-based system- and software engineering has resulted in more and more systems—and more and more complex systems—being modeled. Hence, the problem of managing the complexity of the models themselves has gained importance. This paper introduces three abstractions that are specifically targeted at improving the scalability of the modeling process and the system models themselves.

Keywords: model, metamodel, DSML, inheritance

1 Introduction

Model Driven Engineering (MDE) and a related technique called Model Integrated Computing (MIC) are begin applied in more and more domains. MIC advocates the use of domain specific modeling languages (DSML) relying on a tool infrastructure configured automatically by metamodels [22]. The MIC open source toolsuite centered on the Generic Modeling Environment (GME) [11] has been applied successfully in a broad range of domains by Vanderbilt [14, 8, 15, 10, 12] and others [1, 2, 4, 6, 19, 23]. Design space exploration of embedded systems [16] and the seamless integration of multiple complex simulators [7] are some of the most compelling examples of the power of MIC.

The recent trend in computing is to move away from desktop tools to cloud- and web-based architectures for better scalability, maintainability and seamless platform support. The latest generation MIC toolsuite called WebGME follows this trend. It is a web-based software infrastructure to support the collaborative modeling, analysis, and synthesis of complex, large-scale information systems. The number one design goal of WebGME was to better support the modeling of complex systems. This includes features targeted specifically for an enhanced modeling process including collaborative editing similar to Google Docs and a git-like database backend supporting model version control. WebGME also introduced a number of novel abstractions to support the scalability of the models of complex systems which themselves are necessarily complex. The focus of this paper is exactly these abstractions: crosscuts, model libraries, and *mixins*.

Cross-cutting concepts are always difficult to model and visualize. WebGME introduces the concept of a crosscut that is a collection of objects that the modeler wishes to view together independent of where they are located in the hierarchical model structure. As a simple example, consider a system with software and hardware models. A crosscut is a straightforward way to capture the assignment of software components to hardware units even though they reside in disjoint model hierarchies.

A unique feature of GME has been its support for prototypical inheritance. Each model at any point in the composition hierarchy is a prototype that can be derived to create an instance model. Derivation creates a copy of the model (and all of its parts recursively, i.e., a deep copy), but it establishes a dependency relationship between the corresponding objects. Any changes in the prototype automatically propagate to the instance. WebGME extended this concept to merge metamodels with models. Meta-information, that is language specification, can be captured anywhere in the model composition and inheritance hierarchies. It is exactly the mechanism of inheritance that enforces the language rules by propagating it down the inheritance hierarchy. Consequently, metamodels and models are tightly integrated and any changes in the former are immediately propagated to the latter.

However, the combination of the model composition hierarchy and prototypical inheritance introduces quite interesting inter-dependencies among models. For this reason, GME only allowed multiple-inheritance for metamodels. Since WebGME merged metamodels and models and since multiple inheritance for metamodels proved to be a highly valuable feature, WebGME introduces mixins that provide the useful attributes of multiple inheritance for metamodels, but avoid its pitfalls.

The rest of the paper is organized as follows. Section 2 describes the main ideas behind WebGME and its architecture. Section 3 describes the crosscut abstraction. Section 4 covers model libraries. Section 5 is dedicated to the mixin concept. Finally, a brief overview of related work and conclusions are presented.

2 Overview

The metamodel specifies the domain-specific modeling language. The metamodeling language consists of a set of elementary modeling concepts. These are the basic conceptual building blocks of any given approach and corresponding tools. It is the meta-metamodel that defines these fundamental concepts. These may include composition, inheritance, a variety of associations, attributes, and other concepts. Which of these concepts to include, how to compose them, what editing operations are to operate on them and which are the most important design decisions that affect all aspects of the infrastructure and the domain modelers who will use it.

Hierarchical decomposition is the most widely used technique to handle complexity. This is the fundamental organization principle in WebGME, too. Copying, moving, or deleting a model will copy, move, or delete its constituent parts.

To help manage the complexity of any one model, aspects are also supported. An aspect is a view of a model where only a certain subset of its children are visible. For example, the model of a car can have separate aspects for the mechanical, electric and hydraulic components.

Prototypical inheritance is a unique feature that lets the modeler reuse and refine models. Just as there is a single composition tree, there is a single inheritance tree with a single object at its root. Rules specified by the metamodel, as well as actual model parts, propagate down this tree. Deleting a model will delete all of its descendants in the inheritance hierarchy too.

This approach is markedly different from inheritance in OO programming languages or in other modeling languages such as UML. First of all, it combines composition and inheritance. Note that Smalltalk and JavaScript have prototypical inheritance also, but that does not create new instances down the composition hierarchy. Second, inheritance is a live relationship between models that is continuously maintained during the modeling process. That is, any changes to a model propagate down the inheritance tree immediately.

The novel idea in WebGME is to blur the line where metamodeling ends and domain modeling begins by utilizing inheritance to capture the metamodel/model relationship. Every model in a WebGME project is contained in a single inheritance hierarchy rooted at a model called *FCO*, for First Class Object. Metamodel information can be provided anywhere in this hierarchy. An instance of any model inherits all of the rules and constraints from its base (recursively all the way up to *FCO*) and it can further refine it by adding additional metamodeling information. This is a form of multi-level metamodeling with a theoretically infinite number of levels. As a result of this approach, 1) metamodel changes propagate automatically to every model; 2) metamodels can be refined anywhere in the inheritance and composition hierarchies; 3) partially built domain models can become first class language elements to serve as building blocks; and 4) different (meta)model versions can peacefully coexist in the same project.

Note that while the concept of inheritance and prototypical inheritance has been developed in the context of OO languages, their use in the context of domain-specific modeling is different. The purpose of the domain-specific modeling is to create domain models (that roughly correspond to the object instances at run-time) that are based on metamodels (that roughly correspond to the classes created at design-time) - but these are created and edited in the same tool environment and coexist in the same database. Statically typed, compiled OO languages often draw a distinction between classes and instances and instances cannot be edited in the same editors as the one used for editing the class definitions, although some dynamic OO languages (like Python) have some capabilities that allow such operations.

Other important concepts in the meta-metamodel are pointers which are one to one associations and sets which are one to many associations. A pair of pointers can be visualized as a connection. For example, the default WebGME editor takes any object with two pointers with the reserved names of src and dst, displays the object as a connection and supports the customary editing op-

erations. Otherwise, connections are ordinary models; they can contain children, have other pointers and can be derived, etc. Therefore, the connection concept as such is not part of the meta-metamodel. However, pointers may cut across the hierarchy and hence, are not easily visualizable in an intuitive manner. The next section will describe the crosscut abstraction that were designed to overcome this problem.

Finally, textual attributes can be attached to models as well. Just as a simple illustration of the power of inheritance, there is a textual attribute called name added to the root object of the inheritance tree. The result is that every single modeling object has a name attribute. So, the actual WebGME application code does not need to have a specific concept for a model name.

3 Crosscuts

Cross-cutting concepts are always difficult to model. The typical way to capture relationships between models in different branches and/or levels of the composition hierarchy is through pointers and sets. However, the visual depiction of such associations is not intuitive at all since most tools display models according to composition, that is, they usually show the children of one model in one window (grandchildren may show up as ports). The target of a pointer can be indicated by its name and navigation to it can be supported, for example, by a double click operation, but an intuitive visual depiction of such relationships is sorely missing. For example, a connection between far away objects is supported by the meta-metamodel, yet there is no way to actually show it. To address this problem, WebGME introduces the concept of *crosscuts*.

A crosscut is a collection of objects that the modeler wishes to view together. The selection can be manual, that is, the user can drag objects into a crosscut view. Alternatively, a script can be provided that executes a one-time query to collect models from anywhere in the composition hierarchy. Existing associations between objects in a crosscut are depicted by various lines between the objects. For example, inheritance is shown similar to UML class diagrams, while pointers are visualized with lines and arrows. In addition to visualization, the main utility of crosscuts is that they serve as association editors. The target of pointers and set membership can be edited here. Deleting a model from a crosscut does not delete the object from the project, it simply removes it from the given crosscut.

Each crosscut has a context model, the designated container for new model elements created in the crosscut. (Note that it is atypical to create new models since a crosscut is meant to be a collection of already existing models. However, a connection is a model with two pointers, so allowing new connections in crosscuts was the motivation behind this design decision.) The default context is the root of the composition hierarchy called the $ROOT$. As $ROOT$ can contain anything, crosscuts can be freely constructed. However, if the modeler chooses a context different from $ROOT$, the composition rules of the metamodel apply (even though crosscut containment is not composition). This is actually a great way to control and manage crosscuts. On the flip side, if one wants a crosscut

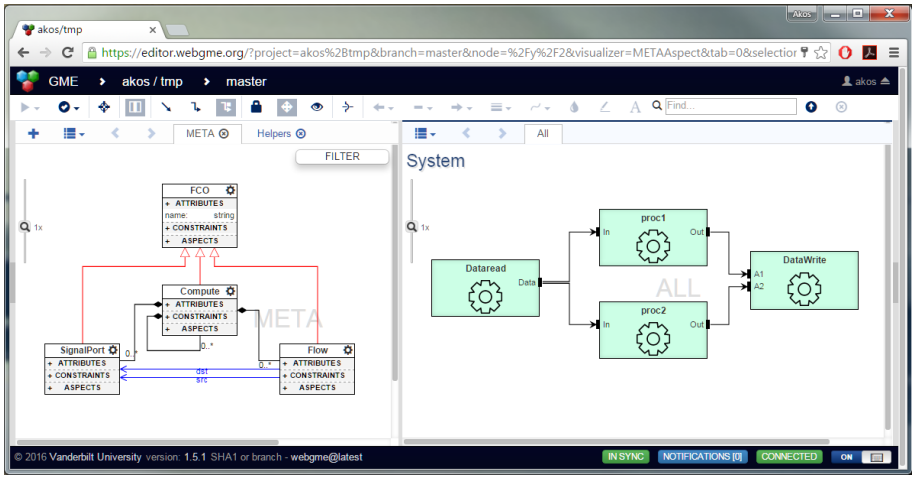

Fig. 1. Hierarchical Signal Flow Graph

with no constraints, but wants to avoid creating too many crosscuts in *ROOT*, one can simply create a model and specify that it can contain First Class Objects (*FCO*) which is the root of the inheritance hierarchy. Any instance of such a model can now serve as the context for unconstrained crosscuts.

One special use for crosscuts in WebGME is for metamodeling. Recall that meta information can be specified anywhere in the composition hierarchy. Therefore, there is no single model to show to edit the metamodel of the DSML. In WebGME, a crosscut is created for the metamodel where the user drags in all models that need to contain DSML specification. It is there and only there, where meta information can be specified. Of course, the metamodel is a special crosscut, because a new association created there does not actually create a new instance of a pointer, for example, but instead specifies that the given kind of pointer of a model can point to the selected model (and its instances).

Consider Figure 1 that depicts the metamodel of a simple hierarchical signal flow graph (SF) on the left and an example SF domain model on the right. The metamodel shows that Compute nodes, SignalPorts and Flows are all derived from *FCO*. Note that unlike in any other tool we are aware of, this inheritance relationship was not drawn explicitly by the user. Instead, when these models were created in the first place, they were instantiated from a model, in this case, *FCO*. The metamodel displays these already existing inheritance relationships but they can be edited as well. On the other hand, the associations in Figure 1 were created in the meta crosscut. For example, the *src* and *dst* pointer specifications were drawn by the user specifying that a Flow represents a relationship between two SignalPorts. The default WebGME editor, in turn, will show these as connections (explained above) as expected in an signal flow graph.

The composition and inheritances trees of this simple example are shown in Figure 2. The composition hierarchy on the left shows that Root has four

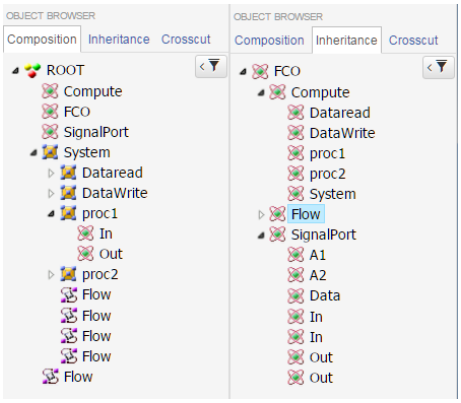

Fig. 2. Composition and Inheritance Trees

children, Compute, SignalPort, Flow and System. The metamodel in the previous figure only showed the first three. That is exactly because a metamodel is a crosscut and we only needed to specify meta information for these models. The System model is an instance of the Compute model (the inside of which is shown on he right side of Figure 1 above). The inheritance hierarchy on the right side of Figure 2 shows this as well as the four other instances of the Compute model that are in turn contained by System.

Finally, we present a domain model example to illustrate the utility of crosscuts. The introduction outlined the use case of modeling a component based software system and a simple parallel hardware architecture that uses multiple compute nodes. If we want to explicitly model the assignment of software component to hardware nodes, we need a placeholder for expressing that relation. Since the hardware and software models should have their own model composition hierarchy, a crosscut is a straightforward way to specify the assignment. Figure 3 shows the crosscut created for this purpose. Note the assignment connections that connect software components to a hardware nodes. A crosscut is an only place where they can be visualized since the source and destination models as well as the connections themselves have different parents in the composition hierarchy. Creating a modeling concept specifically for this purpose is feasible, but it would unnecessarily complicate the metamodel.

4 Libraries

The concept of software libraries have been used widely for decades because of its utility in code reuse and evolution. There is a similar need for reusing models and modeling languages. Since the concepts of the metamodel and model are merged in WebGME, that is, the modeling language specification is embedded in the actual domain models, it is even more important to support model libraries for

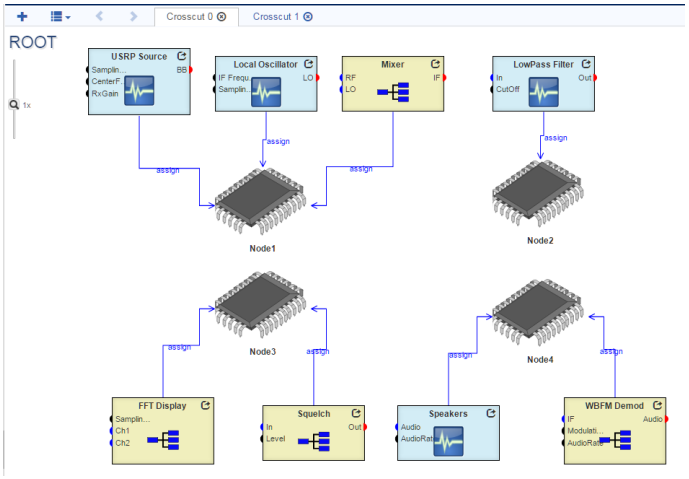

Fig. 3. Software to Hardware Assignment Example

reuse and language evolution. Hence, WebGME supports model libraries in a seamless and intuitive manner.

Creating a library is as easy as identifying a subset of a project and exporting it. A library is defined along the composition hierarchy. This means that any meta- or domain model can be designated as a library and its complete containment sub-tree will be included automatically. The library, in turn, can be imported into another project. Pointers and sets that refer to models in the library from outside of it are not affected at all. Instances of models contained in a library that reside outside of the library cause no problems either. However, what happens when a prototype of a model in the library is not part of the library? Or if a pointer or set refer from inside to the outside of the library? The design decision was made to allow this situation, but importing such a library would only work where the container project also has the exact same models so that these outward pointing relations can be restored. In other words, this works for very closely related projects. The recommended and most typical use of model libraries are for cases when this does not happen. For example, exporting a metamodel does not run into this problem because metamodels are inherently self contained. The other most typical use is collecting a set of basic domain models as a component library where the only external dependencies are to the metamodels. In those cases, the metamodels are typically contained in a library and domain models in another and they are updated at the same time. Note that the *ROOT* and the *FCO* are present in all WebGME projects, so external references to those are fine.

Library updates are not generating notifications automatically but users of the library can request updates when the source of the library is updated. These updates are automatically propagated through the whole project, so the new features are available instantly for the existing instance models while the outdated

features will be removed automatically. Any custom extensions of the library will remain intact as long as their prototypes are still present in the library.

This style of model libraries provide a good basis for domain composition. It can be seen, that as long as a metamodel library depends only on the *FCO*, any other metamodel library becomes compatible with it allowing quick and easy integration. Any associations between such metamodels in the target project are transparently supported without losing the ability to evolve with the original metamodels. Section 5 will present an example of how such metamodel composition works.

5 Mixins: Multiple Inheritance for Metamodels

To enhance the overall re-usability of modeling concepts, WebGME implements a *mixin* feature as an extension to prototypical inheritance. Mixins augment the desirable features of single inheritance, but they only apply to metamodels. Mixins allow metamodels to share specifications, be derived from multiple sources, or extend each other's behavior. Mixins provide a tradeoff to successfully address the problems inherent to multiple inheritance [18] when it comes to prototypical inheritance between object models while keeping the end result as simple as possible.

We can extend the specifications of a metamodel by assigning multiple mixin nodes to it. The resulting node will not only inherit its definitions from its prototype, but from its mixins as well. However, as opposed to prototypical inheritance, the metamodel will not inherit any actual children of the mixins.

The mixin definition is an ordered array of nodes, so if a given meta rule could be derived from multiple sources, the first occurrence will always be used. Also, the specification inherited from the prototype has priority over the specifications obtained from the mixins. Hence, prototypical inheritance is not affected by mixins. It also takes care of the problem of repeated inheritance as any data can only be inherited from a single base, so even if the mixin nodes define colliding properties, the source of every rule remains clear. Furthermore, as metamodels are never pre-compiled in WebGME, the tool is able to give immediate feedback if a rule collision happens as a result of a change in the mixin definitions or the mixins themselves.

Mixins support combining existing domains to model complex systems in a seamless manner. Take the example language of a hierarchical signal flow graph shown in Figure 1. If we want to combine this language with a hierarchical state machine language—shown in Figure 4—where the behavior of a *State* can be modeled by a signal flow graph, we just need to import the two languages as libraries into a new project and define the mixin relation among the elements of the two domains as shown in Figure 5.

As we can see in Figure 5, the mixin relation is visualized similarly to the inheritance relation, but with a dashed line to make a distinction. These relations can be freely added or removed at any point with the following two exceptions. No node can be in a mixin relation of itself or any of its ancestors in the pro-

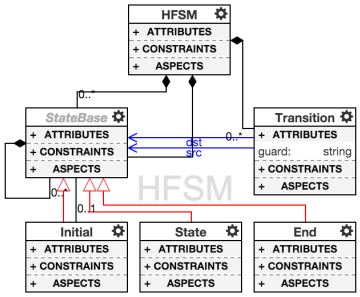

Fig. 4. Hierarchical Finite State Machine

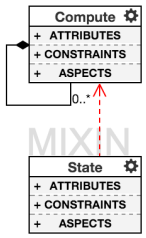

Fig. 5. Combining two languages

totypical inheritance tree, as that kind of relation will not add anything to the already existing rule-set. On the other hand, cycles among the mixin relations are allowed, even though they are not necessarily meaningful, because even with these loops, the order of definitions remains unambiguous and consistent.

Figure 6 shows a simple domain model for the combined state machine-signal flow graph DSML. There is a state machine model on the left side, while on the right, the inside of the state *Processing* is shown that contains the example signal flow from Figure 1. So mixins enabled the composition of the two domains by performing a few simple steps.

6 Related Work

AToMPM [21], a web-based metamodeling and transformation tool, is the most closely related to our work. While many of the design decisions that guided the development of the tool are similar to ours, the fusion of metamodeling and prototypical inheritance, mixins, and crosscuts are unique to WebGME.

To the best of our knowledge, very little work is being done on abstractions to handle model complexity beyond the traditional hierarchical decomposition. Collaborative editing also helps in building large complex models and there is relevant work in the technical literature on this aspect. Various collaborative tools are used in specific domains such as mechanical engineering [13], automotive industry [9], and UML [3]. SLIM [24] is a prototype of a collaborative environment

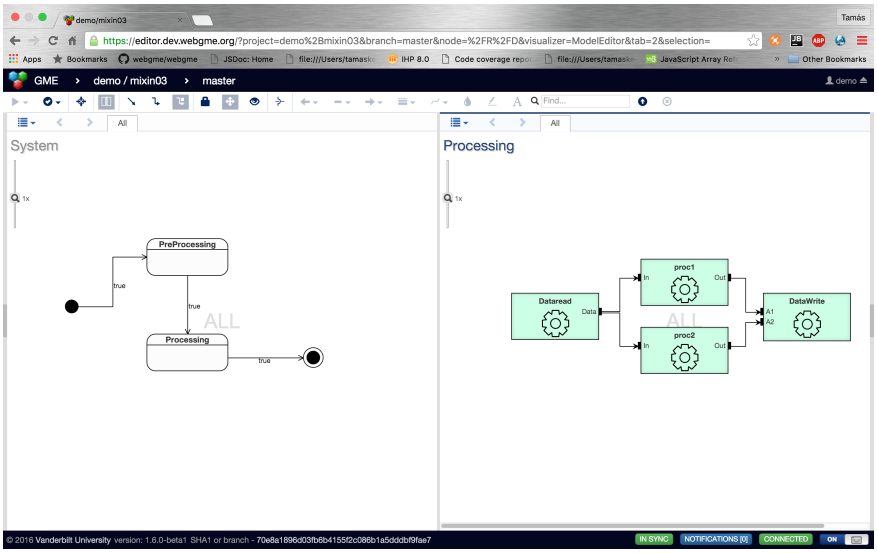

Fig. 6. SF with SM together

executed in a web browser. The Connected Data Objects (CDO) [20] is a model repository and a run-time persistence framework for EMF. It supports locking, offline scenarios, various persistence backends, such as Hibernate, and pluggable fail-over adapters to multiple repositories. CAMEL [5] is also an eclipse plugin that supports collaborative interaction via modeling, drawing, chatting, poster-boards, whiteboards, and it is capable of replaying online meetings. Its focus is on collaborative communications rather than versioning and collaborative use of domain-specific languages.

7 Conclusions

The paper presented three abstractions specifically designed to support the management of complexity of large system models. When the modeling language itself has hundreds of concepts and domain models reach tens of thousands of objects [17], traditional methods such as hierarchical decomposition are no longer sufficient to ensure a manageable modeling process. The purpose of crosscuts is to provide an intuitive way to capture and visualize relations between models in different parts of the composition hierarchy. Model libraries are extremely helpful in managing metamodels, i.e., modeling languages, and support reusable repositories of component models. Combined with prototypical inheritance, they enable modeling language and model evolution in a seamless fashion. The mixin feature presents a trade off between full-scale multiple inheritance and single inheritance. Essentially it enables multiple inheritance for metamodels which is where it is needed most. This novel feature allows combining existing DSMLs to support the modeling of truly complex systems.

References

1. Bagheri, H., Sullivan, K.: Monarch: model-based development of software architectures. Model Driven Engineering Languages and Systems pp. 376–390 (2010)
2. Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., Kurtev, I.: Bridging the generic modeling environment (GME) and the eclipse modeling framework (EMF). In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA. vol. 5. Citeseer (2005)
3. Boger, M., Graham, E., Köster, M.: Poseidon for uml. Pode ser encontrado em http://gentleware.com/fileadmin/media/archives/userguides/poseidon_users_guide/book1.html (2000)
4. Bunus, P.: A simulation and decision framework for selection of numerical solvers in. In: Proceedings of the 39th annual Symposium on Simulation. pp. 178–187. ANSS '06, IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/ANSS.2006.9
5. Cataldo, M., Shelton, C., Choi, Y., Huang, Y.Y., Ramesh, V., Saini, D., Wang, L.Y.: Camel: A tool for collaborative distributed software design. In: Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on. pp. 83–92. IEEE (2009)
6. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.: Generative programming for embedded software: An industrial experience report. In: Generative Programming and Component Engineering. pp. 156–172. Springer (2002)
7. Hemingway, G., Neema, H., Nine, H., Sztipanovits, J., Karsai, G.: Rapid synthesis of high-level architecture-based heterogeneous simulation: a model-based integration approach. Simulation 88(2), 217–232 (2012)
8. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE 91(1), 145–164 (2003)
9. Kong, S., Noh, S., Han, Y.G., Kim, G., Lee, K.: Internet-based collaboration system: Press-die design process for automobile manufacturer. The International Journal of Advanced Manufacturing Technology 20(9), 701–708 (2002)
10. Lattmann, Z., Nagel, A., Scott, J., Smyth, K., Porter, J., Neema, S., Bapty, T., Sztipanovits, J., Ceisel, J., Mavris, D., et al.: Towards automated evaluation of vehicle dynamics in system-level designs. In: ASME 2012 Computers and Information in Engineering Conference. pp. 1131–1141. ASME (2012)
11. Lédeczi, Á., Bakay, A., Maroti, M., Volgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing domain-specific design environments. Computer 34(11) (2001)
12. Levendovszky, T., Balasubramanian, D., Coglio, A., Dubey, A., Otte, W., Karsai, G., Gokhale, A., Nyako, S., Kumar, P., Emfinger, W.: Drems: A model-driven distributed secure information architecture platform for managed embedded systems. IEEE Software p. 1 (2014)
13. Li, M., Wang, C.C., Gao, S.: Real-time collaborative design with heterogeneous cad systems based on neutral modeling commands. Journal of Computing and Information Science in Engineering 7(2), 113–125 (2007)
14. Long, E., Misra, A., Sztipanovits, J.: Increasing productivity at saturn. Computer 31(8), 35–43 (1998)
15. Mathe, J.L., Ledeczi, A., Nadas, A., Sztipanovits, J., Martin, J.B., Weavind, L.M., Miller, A., Miller, P., Maron, D.J.: A model-integrated, guideline-driven, clinical decision-support system. Software, IEEE 26(4), 54–61 (2009)
16. Mohanty, S., Prasanna, V., Neema, S., Davis, J.: Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. ACM SIGPLAN Notices 37(7), 18–27 (2002)

17. Neema, H., Lattmann, Z., Meijer, P., Klingler, J., Neema, S., Bapty, T., Sztipanovits, J., Karsai, G.: Design space exploration and manipulation for cyber physical systems. In: IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL'2014), Springer-Verlag Berlin Heidelberg (2014)
18. Singh, G.B.: Single versus multiple inheritance in object oriented programming. ACM SIGPLAN OOPS Messenger 6(1), 30–39 (1995)
19. Stankovic, J., Zhu, R., Poornalingam, R., Lu, C., Yu, Z., Humphrey, M., Ellis, B.: Vest: an aspect-based composition tool for real-time systems. In: Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE. pp. 58–69 (May)
20. Stepper, E.: Connected data objects (cdo). Website http://www. eclipse. org/cdo/documentation/index. php, seen November (2012)
21. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: Atompm: A web-based modeling environment. MODELS (2003)
22. Sztipanovits, J., Karsai, G.: Model-integrated computing. Computer 30(4), 110–111 (1997)
23. Thramboulidis, K., Perdikis, D., Kantas, S.: Model driven development of distributed control applications. The International Journal of Advanced Manufacturing Technology 33(3), 233–242 (2007)
24. Thum, C., Schwind, M., Schader, M.: Slima lightweight environment for synchronous collaborative modeling. In: Model Driven Engineering Languages and Systems, pp. 137–151. Springer (2009)