Aggressive Model-Driven Development: Synthesizing Systems from Models viewed as Constraints

Tiziana Margaria[1,2] and Bernhard Steffen[2]

[1] METAFrame Technologies GmbH, Dortmund, Germany
{tmargaria}@METAFrame.de
[2] Chair of Programming Systems, University of Dortmund, Germany
{Tiziana.Margaria, Steffen}@cs.uni-dortmund.de

Position Paper

1 Motivation

The Problem

According to several roadmaps and predictions, future systems will be highly heterogeneous, they will be composed of special purpose code, perhaps written in different programming languages, integrate legacy components, glue code, and adapters combining different technologies, which may run distributed on different hardware platforms, on powerful servers or at (thin and ultra-thin) client sites. Already today's systems require an unacceptable effort for deployment, which is typically caused by incompatibilities, feature interactions, and the sometimes catastrophic behavior of component upgrades, which no longer behave as expected. This is particularly true for embedded systems, with the consequence that some components' lifetimes are 'artificially' prolonged far beyond a technological justification, since one fears problems once they are substituted or eliminated.

Responsible for this situation is mainly the level on which systems are technically composed: even though high level languages and even model driven development are used for component development, the system-level point of view is not yet adequately supported. In fact, in particular the deployment of a heterogeneous systems is still a matter of assembly-level search for the reasons of incompatibility, which may be due to minimal version changes, slight hardware incompatibilities, or simply to hideous bugs, which come to surface only in a new, collaborative context of application. Integration testing and the quest for 'true' interoperability are indeed major cost factors and major risks in a system implementation and deployment.

Hardware development faces similar problems with even more dramatic consequences: hardware is far more difficult to patch, making failure of compatibility a real disaster. It is therefore the trend of the late '90s to move beyond VLSI to

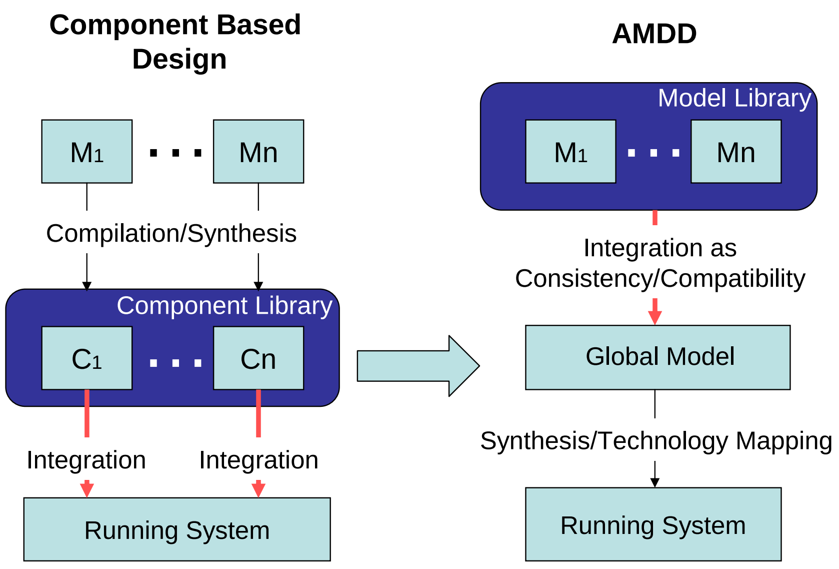

Fig. 1. The AMDD Process

Systems-on-a-Chip (SoC) to guarantee larger integration in both senses: physically, compacting complex systems on a single chip instead of on a board, but in particular also projectually, i.e. integrating the components well before the silicon level, namely at the design level: rather than combining chips (the classical way), hardware engineers start to combine directly the component's designs and to directly produce (synthesize) system-level solutions, which are homogeneous at the silicon level. Interestingly, they solve the problem of compatibility by moving it to a higher level of abstraction.

AMDD: Aggressive Model-Driven Development

At the larger scale of (embedded) system development, moving the problem of compatibility to a higher level of abstraction means moving it to the modelling level (see Fig. 1): rather than using the models, as usual in today's Component Based Development paradigm, just as a means of specification, which

– need to be compiled to become a 'real thing' (e.g., a component of a software library),
– must be updated (but typically are not), whenever the real thing changes
– typically only provide a local view of a portion or an aspect of a system,

models should be put into the center of the design activity, becoming *the* first class entities of the *global* system design process. In such an approach, as shown on the right side of Fig. 1,

– libraries should be established on the modelling level: building blocks should be (elementary) models rather than software components,
– systems should be specified by model combinations (composition, configuration, superposition, conjunction...), viewed as a set of constraints that the implementation needs to satisfy,

- global model combinations should be compiled (synthesized, e.g. by solving all the imposed constraints) into a homogeneous solution for a desired environment, which of course includes the realization of an adequate technology mapping,
- system changes (upgrades, customer-specific adaptations, new versions, etc.) should happen only (or at least primarily) at the modelling level, with a subsequent global recompilation (re-synthesis)
- optimizations should be kept distinct from design issues, in order to maintain the information on the structure and the design decisions independently of the considerations that lead to a particular optimized implementation.

With this *aggressive* style of *model-driven development* (AMDD), which strictly separates compatibility, migration, and optimization issues from model/functionality composition, it would be possible to overcome the problem of incompatibility between

- (global) models and (global) implementations, which is guaranteed and later-on maintained by (semi-) automatic compilation and synthesis, as well as between
- system components, paradigms, and hardware platforms: a dedicated compilation/synthesis of the considered *global* functionality for a specific platform architecture avoids the problems of incompatible design decisions for the individual components.

In essence, delaying the compilation/synthesis until all parameters are known (e.g. all compatibility constraints are available), may drastically simplify this task, as the individual parts can already be compiled/synthesized specifically for the current global context. In a good setup, this should not only simplify the integration issue (rather than having to be open for all eventualities, one can concentrate on precisely given circumstances), but also improve the efficiency of the compiled/synthesized implementations. In fact, AMDD has the potential to drastically reduce the long-term costs due to version incompatibility, system migration and upgrading, and lower risk factors like vendor and technology dependency. Thus it helps protecting the investment in the software infrastructure. We are therefore convinced that this aggressive style of model-driven development will become the development style at least for mass customized software in the future. In particular we believe that AMDD, even though being drastically different from state of the art industrial embedded system design, which is very much driven by the underlying hardware architecture right from the beginning, will change accordingly: technology moves so fast, and the varieties are so manifold that the classical platform-focussed development will find its limits very soon.

The Scope of AMDD

Of course, AMDD will never replace genuine software development, as it assumes techniques to be able to solve problems (like synthesis or technology mapping) which are undecidable in general. On the other hand, more than 90% of the software development costs arise worldwide for a rather primitive software development level, during routine application programming or software update, where there are no technological or design challenges. There, the major problem faced is software quantity rather than achievement of very high quality, and automation should be largely possible. AMDD is intended to address (a significant part of) this 90% 'niche'.

What does this mean? AMDD aims at making things that inherently *are* simple as simple as they should be. In particular this means that AMDD is (at least in the beginning) characterized by abstractions, neglecting interesting, but at a certain level of development unnecessary, details, like e.g. distribution of computation, methods of communication, synchronization, real time. General software development practices can be replaced here by a model and pattern-based approach, adequately restricted to make AMDD effective. The challenge for AMDD therefore is initially to characterize and then model specific scenarios where its effectiveness can be guaranteed. Typically, these will be application-specific scenarios, at the beginning rather restrictive, which will then be generalized and standardized in order to extend the scope of applicability.

Making AMDD work

In order to reach a practicable and powerful environment for AMDD there is still a long way to go:

- adequate modelling patterns need to be designed,
- new analysis and verification techniques need to be developed,
- new compilation/synthesis techniques need to be devised,
- automatic deployment procedures need to be implemented,
- systems and middleware need to be elaborated to support automatic deployment, and,
- at the meta-level, we need a theory for the adequate specification of the settings which support this style of development.

It should be noted, however, that there is an enormous bulk of work one can build upon. Thus there is room also for quick wins and early success: AMDD is a paradigm of system design, and as such, it inherently leaves a high degree of freedom in the design of adequate settings, which, as described in Section 3, can be successfully used already today.

In the following we will focus on the following main ingredients:

1. a *heterogeneous landscape of models*, to be able to capture all the particularities necessary for the subsequent adequate product synthesis. This concerns the system specification itself, the platforms it runs on together with their communication topology, the required programming style, exceptions, real time aspects, etc.

2. a rich collection of *flexible formal methods and tools*, to deal with the heterogeneous models, their consistency, and their validation, compilation, and testing.
3. *automatic deployment and maintenance support* that are integrated in the whole process and are able to provide 'intelligent' feedback in case of late problems or errors.

2 What we can build upon

2.1 Heterogeneous Landscape of Models

One of the major problems in software engineering is that software is multi-dimensional: it comprises a number of different (loosely related) dimensions, which typically need to be modelled in different styles in order to be treated adequately. Important for simplifying the software/application development is the reduction of the complexity of this multi-dimensional space, by placing it into some standard scenario. Such reductions are typically application-specific. Besides simplifying the application development they also provide a handle for the required automatic compilation and deployment procedures.

Typical among these dimensions, often also called **views**, are

- the *architectural view*, which expresses the static structure of the software (dependencies like nesting, inheritance, references). This should not be confused with the architectural view of the hardware platform, which may indeed be drastically different. - The charm of the OO-style was that it claimed to bridge this gap.
- the *process view*, which describes the dynamic behavior of the system. How does the system run under which circumstance (in the good case)
- the *exception view*, which addresses the system's behavior under malicious or even unforeseen circumstances
- the *timing view*, addressing real time aspects
- the various *thematic views* concerned with roles, specific requirements, ...

Of course, UML tries to address all these facets in a unifying way, but we all know that UML is currently rather a heterogeneous, expressive sample of languages, which lacks a clear notion of (conceptual) integration like consistency and the idea of global dynamic behavior. Such aspects are dealt with currently independently e.g. by means of concepts like *contracts* [1] (or more generally, and more complicatedly, via business-rules oriented programming like e.g. in [6]). The latter concepts are also not supported by systematic means for guaranteeing consistency. In contrast, AMDD views these heterogeneous specifications (consisting of essentially independent models) just as constraints which must be 'solved' during the compilation/synthesis phase (see also [13]).

Another recently very popular approach is Aspect Oriented Programming (AOP) [7, 2], which sounds convincing at first, but does not seem to scale for realistic systems. The programmer treats different aspects separately in the code, but has to understand precisely the weaving mechanism, which often is more complicated than programming all the system traditionally. In particular, the claimed modularity is only in the file structure but not on the conceptual side. In other words, in the good case one can write down the aspects separately, but understanding their mutual global impact requires a deep understanding of weaving, and, even worse, of the result of weaving, which very much reminds of an interleaving expansion of a highly distributed system.

2.2 Formal Methods and Tools

There are numerous formal methods and tools addressing validation, ranging from methods for correctness-by-construction/rule-based transformation, correctness calculi, model checkers, and constraint solvers to tools in practical use like PVS, Bandera, SLAM to name just a few. On the compiler side there are complex (optimizing) compiler suites, code generators, and controller synthesizers, and other methods to support technology mapping. A complete account of these methods would be far beyond the purpose of this paper. Here it is sufficient to note that there is already a high potential of technology waiting to be used.

2.3 Automatic Deployment and Maintenance Support

At the moment, this is the weakest point of the current practice: the deployment of complex systems on a heterogeneous, distributed platform is typically a nightmare, the required system-level testing is virtually unsupported, and the maintenance and upgrading very often turn out to be extremely time consuming and expensive, de facto responsible for the slogan "never change a running system".

Still, also in this area there is a lot of technology one can build upon: the development of Java and the JVM or the dotnet activities are well-accepted means to help getting models into operation, in particular, when heterogeneous hardware is concerned. Interoperability can be established using CORBA, RMI, RPC, Webservices, complex middleware etc, and there are tools for testing and version management. Unfortunately, using these tools requires a lot of expertise, time to detect undocumented anomalies and to develop patches, and this for every application to be deployed.

3 A Simple AMDD-Setting

The Application Building Center (ABC) developed at METAFrame Technologies in cooperation with the University of Dortmund is intended to promote the AMDD-style of development in order to move the application development for certain classes of applications towards the application expert. Even though the ABC should only be regarded as a first step of AMDD development, it already comprises some important AMDD-essentials (Fig. 2.3):

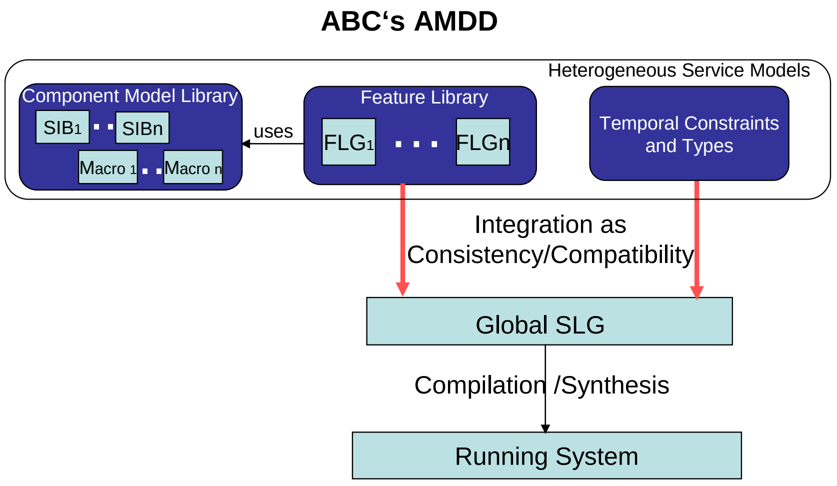

Fig. 2. The AMDD Process in the ABC

1. *Heterogeneous landscape of models*: the central model structure of the ABS are hierarchical Service Logic Graphs (SLGs)[14, 9]. SLGs are flow chart-like graphs. They model the application behavior in terms of the intended process flows, based on coarse granular building blocks called SIBs (Service-Independent Building blocks) which are intended to be understood directly by the application experts [14] – independently of the structure of the underlying code, which, in our case, is typically written in Java/C/C++. The component models (SIBs or hierarchical subservices called Macros), the feature-based service models called Feature Logic Graphs (FLGs), and the Global SLGs modelling applications are all hierarchical SLGs.
   Additionally, the ABC supports model specification in terms of
   (a) two modal logics, to abstractly and loosely characterize valid behaviors (see also [5]),
   (b) a classification scheme for building blocks and types, and
   (c) high level type specifications, used to specify compatibility between the building blocks of the SLGs.

   The granularity of the building blocks is essential here as it determines the level of abstraction of the whole reasoning: the verification tools directly consider the SLGs as formal models, the names of the (parameterized) building blocks as (parameterized) events, and the branching conditions as (atomic) propositions. Thus the ABC focusses on the level of *component composition* rather then on component construction: its compatibility, its type correctness, and its behavioral correctness are under formal methods control [9].
2. *Formal methods and tools*: the ABC comprises a high-level type checker, two model checkers, a model synthesizer, a compiler for SLGs, an interpreter, and a view generator. The model synthesizer, the model checkers and the type checker take care of the consistency and compatibility conditions expressed by the four kinds of constraints/models mentioned above.

3. *Automatic deployment and maintenance support*: an automated deployment process, system-level testing [10], regression testing, version control, and online monitoring [3] support the phases following the first deployment.
   In particular the automatic deployment service needs some meta-modelling in advance. In fact, this has been realized using the ABC itself. Also the testing services and the online monitoring are themselves strong formal methods-based [11] and have been realized via the ABC.

In this sense, the ABC can be regarded as a simple and restrictive but working AMDD framework. In fact, in the ABC, composition/coordination of components as well as their maintenance and version control happen exclusively at the modelling level, and the compilation to running source code (mostly Java and C++) and deployment of the resulting applications are fully automatic.

4 Conclusions and Perspectives

We have proposed an aggressive version of model-driven development (AMDD), which moves most of the recurring problems of compatibility and consistency from the coding and integration to the modelling level. Of course, AMDD requires a complex preparation of adequate settings, where the required compilation and synthesis techniques can be realized. Still, the effort to create these settings and their (application dependent) restrictions can be easily paid off by immense cost reductions in software mass construction and maintenance. In fact, besides reducing the costs, aggressive model-driven development will also lead (more or less automatically) to a kind of normed software development, making software engineering a true engineering activity.

This direction is also consistent with the perspective indicated by the joint GI-ITG position paper on *Organic Computing*[1] [12]: the blurring of borders between hardware and software (machines and programs) that initiated with embedded systems and with hardware/software codesign is going to reach a completely new dimension, where

– the systems are conceived, designed and implemented in terms of *services*,
– they are provided and used in a virtual space, and where
– the distinction on where (local, global, at which node, on which hardware) and how (hardware, software, network, ...) the services are available is relatively inessential information.

In particular, according to availability or convenience, the provider of services can be changed and the provision of services is not a permanent contract anymore.[2]

[1] GI, the Gesellschaft für Informatik and ITG, the Informationstechnikgesellschaft im VDE, the Verband der Elektrotechnik, Elektronik und Informationstechnik are the German counterparts of the ACM and IEEE, respectively.

[2] This is a scenario that concretizes the idea of Sentient Computing [4].

We are convinced that this aggressive style of model-driven development - which overcomes the problem of compatibility between model and implementation, as well as between system components, paradigms and hardware platforms - will become the development style for most of the applications in the future. AMDD is a paradigm of system design, and as such it inherently leaves a high degree of freedom in the design of adequate settings. In particular, we do not expect a single solution to emerge, but rather a collection of environments and settings optimized and tailored for this design paradigm in a number of relevant areas of application.

In particular, we envisage a coordinative design paradigm similar to the already successful paradigm of *feature-oriented design* [5]: in that setting, widely adopted in the telecommunication industry, systems are composed of a thin skeleton of basic functionality, enriched at need and on demand via additional features that deliver premium functionality (services) to the customers/end-users. A well studied example is the combination of POTS (Plain Old Telephone Service) functionality as a basic telephony service provided by a switch, enriched and virtualized by features like Call Forwarding, Conference Call, Collect Billing etc. In feature-oriented design, the structure that matters is not the technical structure (objects, classes) of and under the system, but rather the *structure of the application-domain* (what does the system do for me, when and under which conditions), together with the capability of mapping the what into the how and of changing the how on the fly. Indeed, the Intelligent Network standard is defined in this optic: it defines which features exist in that application domain and what they deliver to the user, and it says nothing about implementational issues, which are left free to the single vendors.

Even though it is only a very first step, we consider the ABC a kind of proof of concept motivating the design of more elaborate aggressive model-based development techniques. In fact, we have already reapt the benefits of this modelling style in one of our projects, in the Integrated Testing Environment projects (with Siemens ICN, Witten (D)). In an initial project phase we built a system-level test environment for complex Computer-Telephony Integrated applications that covered client-server third party application interoperating with telecommunication switches and communicating over a LAN [10]. In a second phase we were faced with the problem of the next generation of applications, that from the engineering point of view had a completely different, and much more complex, profile: we needed to capture internet-based applications that online, role-based, and remotely (over internet) reconfigure e.g. the complete routing and call management settings on a virtual switch implemented as a fault tolerant cluster of physical switches [8]. This meant a new quality of complexity along at least three dimensions: testing over the internet, testing virtual clusters, and testing a controlling system in a non-steady state (during reconfiguration). Thanks to our AMDD approach, this did not affect at all the *conceptual type* of the models we used in the ITE! Thus we were able to help the Siemens engineers to solve their new problem within the existing modelling framework, just by compatibly extending the libraries of models. In particular, they could reuse SIBs, features and SLGs from the previous project phase with no change.

References

1. L.F. Andrade, J.L. Fiadeiro: *Architecture Based Evolution of Software Systems*, http://www.atxsoftware.com/publications/SFM.pdf.
2. AspectJ Website: http://eclipse.org/aspectj/
3. A. Hagerer, H. Hungar, O. Niese, and B. Steffen: *Model Generation by Moderated Regular Extrapolation.* Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002), Grenoble (F), LNCS 2306, pp. 80-95.
4. A. Hopper: The Royal Society Clifford Paterson Lecture: *Sentient Computing*, 1999.
5. B. Jonsson, T. Margaria, G. Naeser, J. Nyström, and B. Steffen. Incremental requirement specification for evolving systems. Nordic Journal of Computing, vol. 8(1):65, Also in *Proc. of Feature Interactions in Telecommunications and Software Systems 2000*, 2001.
6. JRules, ILOG. http://www.ilog.com/
7. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J.Irwin: Aspect-Oriented Programming. Proc. of ECOOP, Springer-Verlag (1997).
8. T. Margaria, O. Niese, B. Steffen, A. Erochok: *System Level Testing of Virtual Switch (Re-)Configuration over IP*, Proc. IEEE European Test Workshop, Corfu (GR), May 2002, IEEE Society Press.
9. T. Margaria, B. Steffen: *Lightweight Coarse-grained Coordination: A Scalable System-Level Approach*, to appear in STTT, Int. Journal on Software Tools for Technology Transfer, Springer-Verlag, 2003.
10. O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. An automated testing environment for CTI systems using concepts for specification and verification of workflows. *Annual Review of Communication*, Int. Engineering Consortium Chicago (USA), Vol. 54, pp. 927-936, IEC, 2001.
11. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In H. Hußmann, editor, *Proc. FASE 2001*, LNCS 2029, pages 233–248. Springer Verlag, 2001.
12. *Organic Computing: Computer- und Systemarchitektur im Jahr 2010*, position paper of the VDE/ITG/GI. http://www.gi-ev.de/download/VDE-ITG-GI-Positionspapier
13. B. Steffen. Unifying models. In R. Reischuk and M. Morvan, editors, *Proc. STACS'97*, LNCS 1200, pages 1–20. Springer Verlag, 1997.
14. T. Margaria, B. Steffen: *METAFrame in Practice: Design of Intelligent Network Services*, in "Correct System Design - Issues, Methods and Perspectives", LNCS 1710, Springer-Verlag, 1999, pp. 390-415.