Fostering the Reuse and Collaborative Development of Models in the AMS SoC Design Process

Torsten Mähne, Alain Vachoux, and Yusuf Leblebici
Laboratoire de Systèmes Microélectroniques (LSM), École Polytechnique Fédérale de Lausanne (EPFL)
EPFL/STI/IMM/LSM, Bâtiment ELD, Station 11, CH-1015 Lausanne, Switzerland
Phone: +41(21)69-36922, Fax: +41(21)69-36959, WWW: http://lsm.epfl.ch/, E-Mail: torsten.maehne@epfl.ch

***Abstract*—Systems-on-Chips (SoCs) integrate more and more heterogeneous components: analog/RF/digital circuits, sensors, actuators, software. For the design of these systems very different description formalisms, or Models of Computation (MoCs), and tools are used for the different subblocks and design stages, which often create interoperability problems. Additionally the verification of a complete SoC is difficult due to huge performance problems. The goal of this Ph.D. work is to develop an efficient modeling and simulation platform that supports the design of mixed-signal SoCs using component models written in different design languages and using different MoCs. One component of this work is the development of a web-based platform for collecting behavioral models and supporting the design of Analog and Mixed-Signal (AMS) SoCs. Its current state and an outlook on its further development is the focus of this paper.**

I. Introduction

The design of SoCs has currently to address a number of significant issues, namely: *increasing complexity* (computing and communication capabilities), *significant heterogeneity* (analog, RF and digital hardware, embedded software, sensors, actuators), *increasing environmental awareness* (energy saving, environmental monitoring and interaction), *increasing sensitivity to silicon technologies* (deep sub-micron technological processes), and *increasing reuse of subsystems* (ever shrinking time to market). One difficulty in solving these issues while designing a mixed-signal SoC is the usage of a diversity of specialized EDA tools, design languages and design formats that are usually efficiently supporting only one aspect, i.e., RF, analog, or digital, of the complete mixed-signal system. Furthermore, the design's heterogeneity requires a diversity of description formalisms, also called *Models of Computation* (MoCs), analysis and simulation methods. It is still difficult to handle all the different design aspects *simultaneously*. Designers are forced to bridge the gaps between tools and methodologies using manual conversion of models, proprietary tool couplings, and tool integrations. This makes the design process overly complicated, error-prone and time consuming. Another very important issue is the capability to perform *efficient overall system verification* in the early phases of the design process. System verification is based on the development of virtual prototypes [1] of the complete heterogeneous system. Its main goals are to support architecture exploration, performance estimations, validation of reused parts, verification of the interfaces between MEMS, RF, analog, and digital parts, the interoperability with other systems, and the assessment of the impacts of the future working environment and the used manufacturing technologies. Modeling tasks are therefore at the heart of the V-shaped SoC design process (Fig. 1). The management of the created models of a device, component, or the whole system on different abstraction levels is one important aspect since the model development itself is not easy for several reasons:

- The *block heterogeneity* requires specific knowledge about the involved physical domains.
- To *formalize* his intend into a model, the designer needs to know the design languages, methodologies, and tools.
- The model needs to be *adapted to tools for specific design task*, e.g., simulation, verification, optimization, synthesis.
- The model should contain *only* the *necessary details* to fulfill a design task within a *reasonable execution time*.

The reuse of models with possible adjustment to the new task is one important element of any effort to shorten the design time but is complicated by two important issues. First, designers are often not aware if and where a similar model is already existing. Second, the designer has to gain trust in the validity of the model for his specific task, which is more difficult to achieve for foreign models because of the so-called "Not invented here" syndrome. To overcome these problems, the models need to be documented regarding their interface, implementation, extend of covered effects, how they were verified, and other general properties. The designer needs to understand from this information the model structure and its functionality as well as to judge, if it is suitable for a given task. It has to be clear, which tools the model is expected to be compatible with. The ModelLib project aims to solve this through the development of a web-based platform to improve the access to and reuse of model collections. It shall allow to gather behavioral models, to validate collected models through collaborative review and development, to organize meta-model information, and to allow queries supporting the top-down and bottom-up design approaches. [2] ModelLib is realized as an open-source framework, but will implement appropriate IP protection mechanism for the stored documentation and models. This way, sensitive models may be kept from unauthorized use, while still keeping access to the related meta information. Its current state and an outlook on its further development is the focus of this paper.

This work has been funded by the Hasler Stiftung under project № 2161.

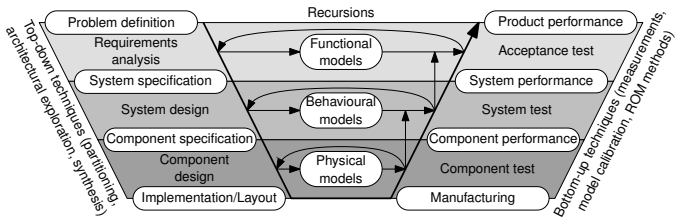

Fig. 1. V-model of the design process of an AMS SoC

II. Requirements to a Model Library

A model library can be set up on different organizational levels, e.g., within an project group, a company, or as a community portal on the Internet. The use cases (Fig. 2) for the AMS designer accessing the library server through a client include submitting, retrieving, and collaboratively developing the models over the Internet. However, the demands for security and required detail of access control rise with each level of broader access. The communication between client and server needs to be done through an encrypted channel. Users have to authenticate themselves so that the information stored in the library can be selectively made available. This is needed to protect the Intellectual Property (IP) of the authors and to support the conformance to their license terms, under which they are making their work available to the public or a restricted group of users. The users of the model library can be categorized into five basic roles with a different profile of allowed actions. The *guest* is anonymous and has thus the minimum rights. He can only browse for a limited collection of public models, send queries about their meta information, and retrieve their source code by checking it out from a public repository. By authenticating himself with a valid user name and password, he can become an *authorized user*, who gains further rights dependent on his membership in different groups. These groups *grant* or *deny* him access to the different parts of the library (the different models, test benches, and documents themselves, as well as the supplementary meta information about them). He can participate in the review process by discussing the models, test benches, and documents stored in the library. He can also contribute to their development by submitting new models, adding/editing of the meta information about them, and organizing them into different categories called *model classes*. *EDA tools* need also a direct access to the model library and are a special form of an authorized user. There are two privileged roles. The *content manager* configures accounts and access rights of the users, reviews their submitted models, test benches, and documents, and adds commonly used information about supported design languages and tools. The *system administrator* is responsible for maintenance and development of the library platform.

One way to access the models in the library is to directly browse through the collection of available models. For this they need to be sorted into a hierarchy of *model classes*. After selecting a model, the user is presented the meta information describing the properties of the model, which can be detailed

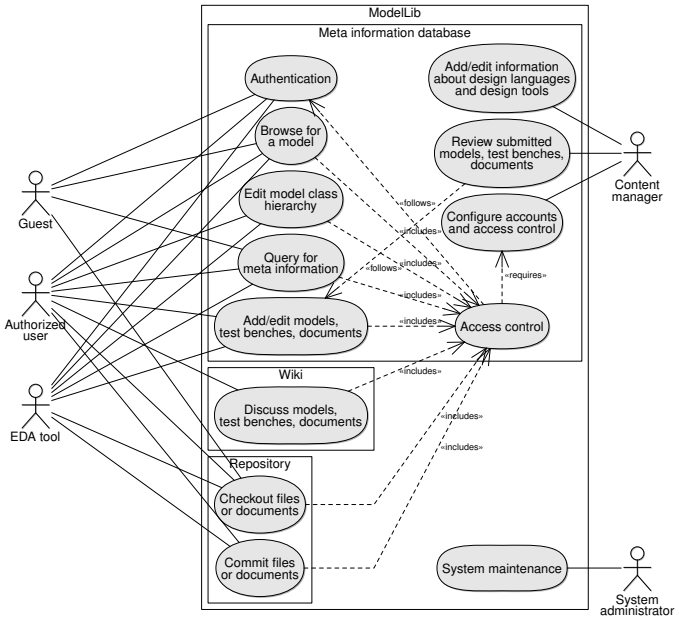

Fig. 2. Use cases of a model library

into *structural meta information* describing "How the model is built?" and *semantical meta information* describing "How the model can be used?". The *structural meta information* describes the following aspects:

- *Name and storage place* of the model within the model class hierarchy;
- *Interface*, including detailed information about all parameters and ports as well as the assertions associated to it;
- One or more *model implementations (architectures)* in the form of behavioral and/or structural description(s);
- *Design entities*, each one gathering the interface and one model implementation using a particular design language (e.g., VHDL-AMS) and tested against particular tools (e.g., simulators or synthesizers); and
- *Test benches* to validate design entities along with test data and expected results.

To each of these aspects, an arbitrary number of references to external documents can be given. Fig. 4 from the ModelLib prototype shows one way of presenting the structural meta information to the user.

The *semantical meta information* further characterizes a model regarding its fidelity, performance, and usage as described for example in [3] and includes, e.g.:

- *Model refinement level: Pins* (named interface but no internal features), *Static* (time-invariant, steady-state internal behavior), *Dynamic* (time-varying behavior), *Precision* (including significant amount of second order effects), *Vector* (model with directional or spatial interface).
- *Model of Computation* (MoC): on which the model is based, e.g., Discrete-Event (DE), Finite State Machine (FSM), signal flow, conservative network, bond graph.
- *Feature properties:* describe to which *extent* (e.g., static, dynamic) a certain aspect of the component behavior

is captured by a model, on which *physical effect* (e.g., self-heating, noise) it is based, and how it influences the component *performance criteria* (e.g., gain, bandwidth).
- *Model validity:* describing under which assumptions and operating conditions a model is valid.
- *Suitability for design tasks:* e.g., architecture exploration, area/power estimation, bottom-up verification.
- *Keywords:* to index a model.
- *Execution capabilities:* Depending on its implementation, a model is suitable for different types of execution, notably *simulation* and *synthesis*. The results obtained from the model execution need to be characterized. A model can support for *simulation* different *analysis types* (e.g., DC, transient, small signal, and stability analysis). For each analysis type, the model can give different results (e.g., current, voltage, power, failure). Each result can be in a different form like *Flag*, *Message*, *Scalar*, *Waveform*, or *Relation*. *Synthesis* transforms a model through an algorithm into another model. During top-down design, details are added in each synthesis step. During bottom-up design, models can be simplified through reduced-order modeling methods to make them suitable for simulation on higher levels of abstraction.

It is possible to include all this supplementary information into free-form description fields, but for large model collections it is better to structure them as far as possible and to store them in an adapted data structure.

III. Implementation of the ModelLib Prototype

A running prototype of ModelLib [2], [4] implements basic features of a model library. Fig. 3 shows its architecture relying on several open-source tools (PostgreSQL, Apache 2, Subversion/WebSVN, and YaWiki) and how its components interact. The lower part of the figure shows the different user created documents managed by the ModelLib server. The file revisions of the documents are stored in the *Subversion repository*. The meta information about models and accompanying documents are stored in the *meta information database*. Informal texts, like discussions and HOWTOs, are stored in the *wiki database*. Both databases are managed by a *PostgreSQL* server. The user interfaces provide the access to the data storages and are implemented on an Apache 2 web server using PHP. This has the advantage that users can access ModelLib over the Internet using a standard web browser. *WebSVN* allows the comfortable browsing of the Subversion repository. On the user side, the file revisions of the models and documents are managed by the Subversion client. It also provides the possibility to commit them to and update them from the repository. *YaWiki* serves as a platform to discuss and jointly develop the documentation of the models. The developed ModelLib web interface (Fig. 4) allows the browsing for a model through a hierarchy of model classes. The meta information about models, test benches, design languages, design tools, and documents can be displayed. New models can be added by committing them into the repository and adding/editing their meta information using the web interface.

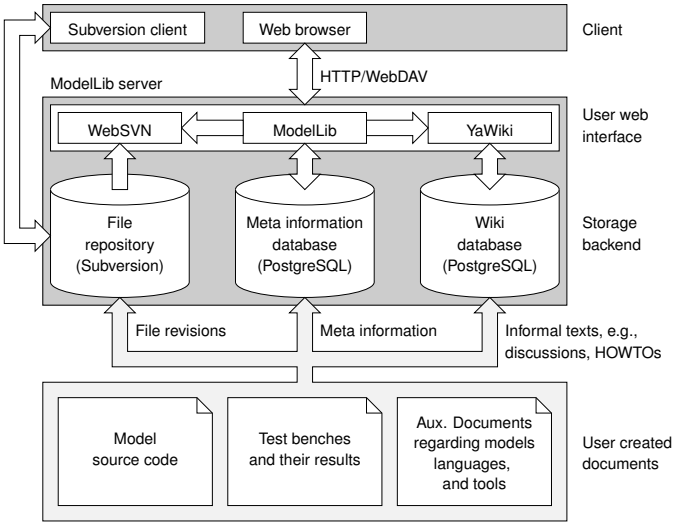

Fig. 3. Architecture of the ModelLib prototype

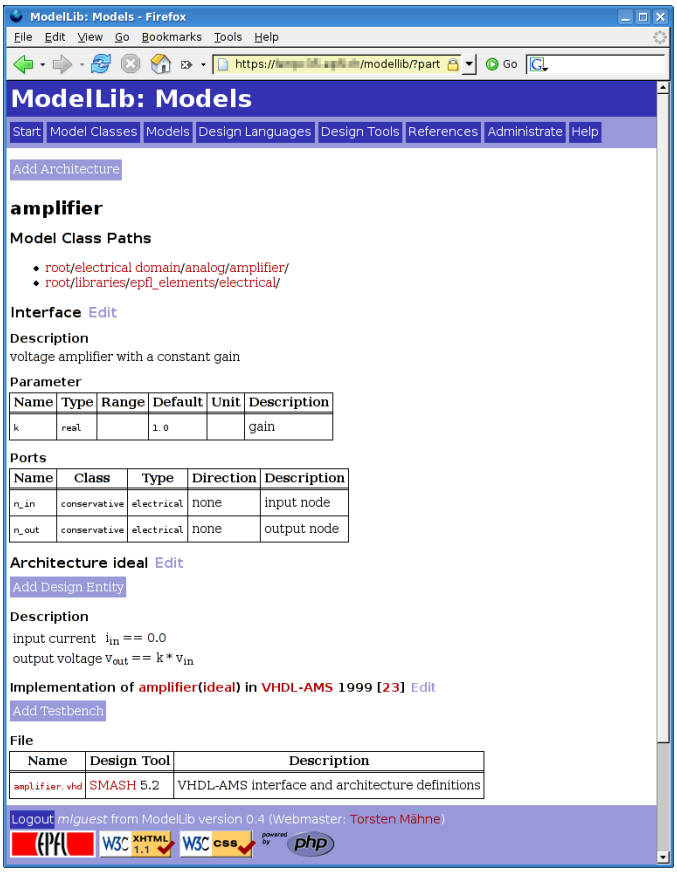

Fig. 4. User interface of the ModelLib prototype

The meta information describing the properties of the models is stored in a relational database. It currently considers the model class hierarchy, the information about referenced external documents, the information about available design language and tool versions, and the meta information about interface, architectures, assertions, design entities, test benches,

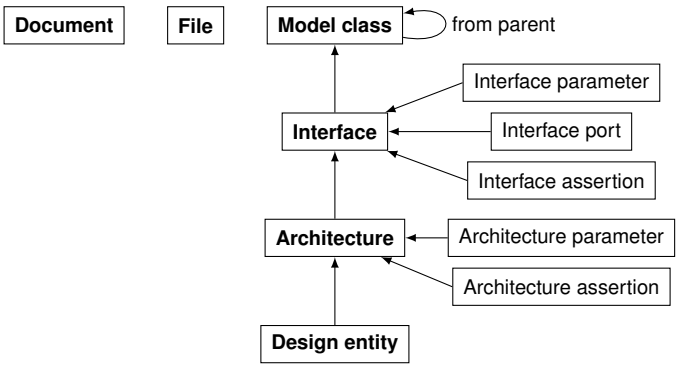

Fig. 5. Access rights inheritance between the tables storing the model meta information. Tables written in bold have an own ACL, which is merged with the inherited rights definitions. Documents and Files are not inheriting from other instances because they can be shared between several models.

and files of the models. This fully covers the structural meta information described in Section II. The structural meta information is not yet implemented in the database structure and can be stored, currently, only as free-form descriptions.

IV. Fine-Grained Hierarchical Access Control Mechanism to the Meta Information

One important aspect for the success of a model library is the management of IP represented by the models. A fine-grained access control mechanism is thus necessary to disclose selectively information to different user groups. For example, the public might only have access to high-level models of a component or only the full interface information plus a compiled/encrypted model, whereas a customer of the design company has full read access to the documentation and the model sources, and the model developers themselves have full read/write access to all information. Subversion and YaWiki already implement mechanisms to limit the access to files/wiki pages with respect to the user and its group memberships using ACLs. But the meta information about the models stored in the relational database is the real key to the models in the repository and the documents in the wiki and may contain sensitive information. The Relational Database Management System (RDBMS) PostgreSQL provides authentication and access control mechanisms through its role concept and the grant/deny of privileges on tables, views and function execution. Each role can be member of other roles permitting the common definition of privileges. But the definition of privileges on the table level is too coarse in the case of the model library, since common information to all models, e.g., about the interface, are stored in the same table. That is why a tuple-wise (per table row) access control mechanism has been implemented additionally inside the database using only the features provided by the RDBMS [5].

In the chosen approach, the Select, Update, Delete, and Insert rights on the tables are granted only to the administrators (members of role mladmin). The authorized users and guests (members of mluser and mlguest) are granted only the Select right on views to which the rows from the underlying tables are propagated selectively according to the entries in an additional ACL table for the roles the database user is member of. The authorized users have also the Update, Delete, and Insert rights granted on the views. The modification of the view data is propagated back to the underlying tables using rules [6], which drop modifications not allowed by the entries of the attached ACL. The rights administration would be too tedious, if the rights were specified for each single tuple. That is why a rights inheritance scheme (Fig. 5) has been implemented into the view queries and rules allowing to reuse the rights defined on entities higher in the hierarchy and permitting their overriding through the local ACL.

The implementation on the database layer has the advantage that it simplifies the development of the higher (application logic and presentation) layers and that it allows the reuse of the access control implementation for different interfaces (e.g., web interface and EDA tool links).

V. Conclusions and Outlook

The developed ModelLib prototype implements basic use cases of a model library to foster the reuse and collaborative development of models in the AMS SoC design process. A fine-grained access control mechanism has been implemented recently on the database layer. The next steps in the ongoing development are the redesign and reimplementation of the business logic and presentation tiers using Java Platform, Enterprise Edition (Java EE) to improve the modularity of the data storage, application logic, and presentation layers and to realize direct EDA tool access through an API. Another important task is to support the structured storage of the semantical meta information, which characterize the fidelity, usage, performance, and other properties of the model. Elaborated query schemes need to be implemented into the web interface to guide the designer in the selection of a suitable model for his current task. Further goals are completion of the web interface functionality, better integration of the different software components, and automatization of the model import.

References

[1] T. Mähne, K. Kehr, A. Franke, J. Hauer, and B. Schmidt, "Creating virtual prototypes of complex MEMS transducers using reduced-order modelling methods and VHDL-AMS," in *Applications of Specification and Design Languages for SoCs: Selected papers from FDL 2005*, ser. The ChDL series. Springer, Oct. 2006, pp. 135–153.

[2] T. Mähne and A. Vachoux, "ModelLib: A web-based platform for collecting behavioural models and supporting the design of AMS systems," in *Forum on Specification and Design Languages (FDL) 2006*. Darmstadt University, Germany: ECSI, Sep. 19–22 2006, pp. 91–97. [Online]. Available: http://infoscience.epfl.ch/search.py?recid=88137

[3] SAE Electronic Design Automation Standards Committee, *Model Specification Process Standard*, SAE: The Engineering Society For Advancing Mobility Land Sea Air and Space International Std. J2546, 2002.

[4] T. Mähne. (2006, Jul. 13) ModelLib prototype. Laboratoire de Systèmes Microélectroniques (LSM), École Polytechnique Fédérale de Lausanne (EPFL). [Online]. Available: https://lsmpc4.epfl.ch/modellib/

[5] T. Böhm, "Development of a web-based application for collecting models and supporting the design of AMS systems," Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik (FIN), Postfach 4120, D-39016 Magdeburg, Jan. 2007.

[6] *PostgreSQL 8.1.8 Documentation*, The PostgreSQL Global Development Group, 1996–2005. [Online]. Available: http://www.postgresql.org/docs/8.1/interactive/index.html