An XML-based Meta-model for the Design of Multiprocessor Embedded Systems

W.O. Cesário, L. Gauthier, D. Lyonnard, G. Nicolescu and A.A. Jerraya
TIMA Laboratory
46 av. Félix Viallet - 38000 - Grenoble - France
Wander.Cesario@imag.fr

Abstract

The design of multiprocessor embedded systems requires new design paradigms. Most new paradigms are conceived around the idea of deploying pre-characterized software and hardware components. Making all these building blocks communicate in an efficient and error-proof manner is the most important challenge facing new design methodologies. Expectably, convenient modeling of these components and their communication at different abstraction levels is the central problem of any new system-level synthesis environment. In this paper we present Colif, a meta-model that was developed to be an object-oriented intermediate design model for system-level synthesis of multiprocessor embedded systems. Its design representation clearly distinguishes communication infrastructure and behavior, allowing independent refinement of these features. Colif objects are polymorphic, in the sense that they represent design information at different abstraction levels. At the highest abstraction level, it does not impose any specific computation model or communication semantics, it could be seen as a "meta-model". It will only have full execution semantics, so we could call it an "actual" design model, when working at lower abstraction levels.

1. Introduction

The design of state-of-the-art multiprocessor "System-on-a-Chip" (SOC) embedded systems while being technically achievable and a powerful market differential, is yet an overly difficult challenge in terms of design effort. New design paradigms are being created to increase designers' productivity to a level that will make SOC design a viable marketing alternative. They are mostly considering the (re-)utilization of pre-characterized embedded components both in software and in hardware. This principle was borrowed from the design using discrete components assembled in a printed-circuit board (PCB) where "standard" component parts are widespread. Unfortunately, some limitations of the discrete world are artificially finding their way into the SOC embedded world on certain proposed design methodologies. Mainly, for the intrachip communication network, they are relying on ultrahigh-speed wide data-buses, which is easier to manage than point-to-point connections. Often, system wide buses are the bottleneck of multiprocessor systems, embedded or not. In embedded systems, this bottleneck can be avoided by carefully tuning a point-to-point communication network. This approach is much more complex to manage because it needs more intelligence at both ends of the wires, so a design methodology centered at communication infrastructure is crucial to the deployment of this solution. While most new single-die multiprocessor chips are using a high-speed system bus, many people think that point-to-point connections are the right choice for future designs [1]. As with many other high-level architectural issues, the final decision must be taken by the designers. Consequently, good design methodologies for multiprocessor embedded systems must support very flexible communication models in particular, which asks for a very flexible design model in general.

Clearly, a design model must be judged by the flexibility of its communication models, but also by the adopted execution model. In embedded systems, basically we find two kinds of execution/task models: (i) in the *run-to-completion* model tasks will not stop until they finish execution, (ii) in the *interactive* model tasks may stop processing to wait for an I/O or they could relinquish control themselves. It is complex to use the first model because all internal system states must be removed by a front-end. In large systems, this could require a huge effort. The required features for design models targeted at synthesis of multiprocessor embedded SOCs are:

- It must concentrate on the communication infrastructure, leaving behavioral modeling to a front-end and detailed architecture representation to a back-end;
- It must enable refinement of intrachip's communication network more or less independently from the optimizations of the internal behavioral of system's components [2];

- It must encompass many abstraction levels, but for flexibility's sake, computation and communication models must not be imposed on the highest abstraction level.

The above requirements are difficult to fulfill because of constraints imposed by the choice of a restrained set of computation models or communication models or both. System specifications based on simulation engines such as SystemC [3] and SpecC [7] impose a certain computation/communication model from the very beginning of system's modeling. Other initiatives like Open Verilog International' Semantic Reference Manual (OVI-SRM) try to be independent of input languages but impose a fixed task/communication model. These representation constraints imposed on designers are the price to be paid when adopting a given modeling solution with predefined execution semantics. We propose a meta-model that is able to generate executable models after taking some crucial design decisions. While writing onto a meta-model, designers could keep their freedom of choosing computation/communication models and they get the equivalent executable models for "free" when the appropriate design decisions were made.

2. Semantic models for system-level design

A first trend in system-level modeling was to create new languages or to extend existing languages to match the requirements of modeling SOCs. N2C [4], SpecC [7] and SystemC [3] goal is to leverage C/C++ designer knowledge for system specification using language/library extensions that provide the necessary modeling concepts, namely: timing, netlists and concurrency. Simple language compilation provides a fast executable model for verification but task/communication models are strictly bound to the simulation policy. Rosetta [8] is a completely new language from the System Level Design Language (SLDL) committee, essentially it builds a formal constraint model while combining heterogeneous semantic domains. Efforts to produce an executable model from Rosetta have taken the direction of formal verification environments so it is foreseeable that strict interaction rules between semantic domains will restrict communication model flexibility. The most recent release of SystemC (version 1.1 beta) is a step towards making modeling more flexible by introducing higher-level communication models (called "abstract protocols") and higher-level functional abstractions. Notwithstanding the added flexibility, the task model used to simulate these new abstraction levels and abstract protocols is the Remote Procedure Call (RPC). RPC is inappropriate for describing distributed systems because of the implied serialization.

More recently, there seems to be a trend towards language-independent semantic models and meta-models. Open Verilog International's Architectural Language Committee (OVI-ALC), came up with a "Semantic Reference Manual" (SRM) marking a switch from the search of a specific system-specification language. However, OVI-SRM assumes that tasks run-to-completion; the communication model and the abstraction level are fixed, interconnections are invariably mapped onto a shared-memory peer-to-peer network. Virtual Socket Interface Alliance' (VSIA) system level development design working group proposes an "abstract model" where functionality and accuracy is assured by standards: System-Level Interface (SLIF) for communication models, Performance Modeling Standards for design constraints, Data-type Standard for model integration. While very flexible, this model imposes a complex refinement process where all properties that are refined need to be back verified against the related properties in the more abstract models [5]. A more interesting view was brought by the semantics group of the Gigascale Silicon Research Center (GSRC) [9]. For them, system modeling has four aspects: "abstract syntax", "syntactic transformations", "concrete syntax" and "semantics". Besides being "concrete syntax" (language) independent, the "abstract syntax" is also "semantic" independent. We could think of it as a meta-model since it must be able to describe at least: netlists, state transition diagrams, block diagrams, object models and graph structures. Now that SLDL and OVI are both part of the Accellera initiative, they are trying to collaborate with VSIA and GSRC to promote the development of a unified SRM.

Colif is a meta-model that is able to use mixed communication models at different abstraction levels and that adopts a computation model based on interactive tasks. Being a meta-model, it will be possible to generate executable co-simulation models for design verification from system-level behaviour models to cycle-accurate RTL models and mixed-abstraction level models as design decisions are being made. Colif is targeted to a communication-based design flow; the next section introduces communication models and their corresponding abstraction levels.

3. Abstraction levels in communication

Table 1 summarizes abstraction levels with regard to communication, note that what is labeled "Driver level", "Message level" and "Service level" constitute what is normally denoted "system level". The reason for differentiating levels according to communication abstractions is a matter of specification and design process. We use three features to characterize communication abstractions: the media, the data type and the behaviour. The media is the infrastructure for carrying data of a certain type while performing some actions that transforms this data, these transformations are called behaviour (more details will be presented in section 4.2). A very high-level specification

can be described in terms of services supplied to the environment, and a breakdown of such a description would naturally be the services and their constituting requests in the composition of tasks or processes. After analyzing the requirements and their fulfillment, the next step is designing a high-level architecture for the system. Such architecture consists of tasks and processes from the specification, but more in detail. Yet the design and analysis of the communication will be the most complex task, since the properties of this communication scheme will severely impact the performance of the whole system. In order to handle this, we have to concentrate on the communication mechanisms, primitives and abstractions.

Table 1 - Abstraction levels for communication

Abstraction level	Communication			Encapsulation	Description	Typical communication primitive
	Media	Data type	Behaviour			
Service	Type-resolved dynamic net	namespaces + concrete and algebraic data types	Routing	Classes (objects), Packages	Specification languages	Request(print, device, file)
Message	Active channels with complex data structures	Concrete generic data types	Protocol conversion	Dynamic process blocks	SDL, MSC	Send(data, disk)
Driver	Logical interconnections	Fixed enumerated data types	Driver-level protocol	Static process blocks, modules	Cossap, CSP, StateCharts, SystemC1.1	Write(data, port) Wait until x=y
RT	Binary signals	Fixed bit-vector data representation	Transmission	Modules, entities	VHDL,Verilog, SystemC0.9-1.0	Set(value, port) Wait (clock)

To allow reuse, selection and integration of different modules, the communication and the computation will be separated and encapsulated. Through the presented abstraction levels, a system will be modeled as an ensemble of communicating hierarchical modules. Each module is defined by its interface and its content where the interface is composed of a set of ports on which external or internal operations can be performed. The module content may be composed of other module instances, or it may be composed of tasks or processes (more details will be given in the next section). Each abstraction level is defined by specific concepts that encapsulate lower abstraction level concepts, and they are themselves also encapsulated in levels that are more abstract.

3.1. Register transfer level

In the register-transfer level (RTL), combinatory logic controls the registers and any address decoding or interrupt management is explicitly defined and described. Values are represented by signals between the modules, and the register to be loaded may be chosen with a compound structure carrying the address for the register. The main focus on this level is the handling of the buses, and how the different modules can share the logical and operational space defined by these structures (e.g. shared bus, point-to-point communication, etc.). These structures are not real communication abstractions since this level demands a fixed bit-vector representation for any data type. Setting values on physical/logical wires will produce an immediate reaction with respect to values and time; the only delay is due to the physical character of the wires, there is no delay caused by any underlying transmission protocol. RT-level communication abstractions are shown in the lowest row in Table 1. Most commercial synthesis tools are still at this level of abstraction. They offer a more unbound way of describing the inner content of processes—not limited to a specific finite-state machine description, the communication, however, is still bound by the same restrictions and lack of abstractions.

3.2. Driver level

The system is modeled as interconnected modules communicating through logical connections exchanging fixed, enumerated data types (e.g. integers, reals, etc.) conforming to driver-level protocols. Communication time is non-zero but predictable, because size and structure of data, and the transmission protocol are well known. Driver-level typical communication abstractions are master-slave buses and *rendez-vous* or FIFO based point-to-point communications. For a bus implementation, protocol choices at this level will exactly define how the modules' access and privileges regarding the buses will take place. However, more detailed information about the interface between the buses and the modules will not be described at this abstraction level.

3.3. Message level

At the message level it is useful to be able to describe how processes communicate in the respect of concurrency, synchronization, and channel behaviour. The communication will be modeled through active channels capable of interconnecting modules independent of underlying communication protocols. Data are terms,

and do not necessary have a predetermined size, and the communication works solely on the level of such terms. Communication time is thus non-zero and in addition not predictable. It is a simple model, yet it can, by changing the underlying semantics and channel behaviour, describe diverse communication schemes. Refining active channels into logical interconnections will normally entail some module describing the channel behaviour acting as communication controller. SDL, the Specification and Description Language [10], uses a process and channel-modeling basis with the basic send/receive semantics in addition to infinite queuing for channels. Some projects have researched using this level for functional specification and synthesis, e.g. [11], but they impose restrictions on the descriptions and communications in particular.

3.4. Service level

The highest abstraction level with specified communication semantics is reached when communication is seen as the combination of requests and services. A process can request a service from another process. In this model, the underlying protocols, connection structures, and essential timing issues are completely abstracted away. This communication abstraction can support several time models based on the concurrency structure and local time capabilities of the processes themselves. Details of this level are presented in [12]. CORBA, the Common Object Request Broker Architecture [13], is a good example of the request-service model in the software domain. Programs or libraries register their services through descriptions in an Interface Definition Language (IDL) and one or more ORBs (Object Request Brokers) perform the actual communication routing between a mutual request-service pair.

4. Colif meta-model

Colif is intended to be an object-oriented intermediate design model for system-level synthesis of multiprocessor SOC embedded systems. The design representation is based on an object model that clearly distinguishes between communication infrastructure and task behavior so that they can be independently refined.

Colif's basic objects are explicit renderings of the design structure: hierarchical modules and communication networks. They are polymorphic, in the sense that they represent design information at different abstraction levels, from the system-level to the RT-level. Colif is a meta-model because, at the highest abstraction level, no specific computation model or communication semantics are imposed.

4.1. XML-based model

Colif is a markup language; it is implemented using XML (eXtensible Markup Language). We could have directly used XML and associated data-structures for implementing Colif but, for diverse reasons, we decided to write Colif in an intermediate meta-language: MIDDLE-ML, which is also a markup language written in XML. First, with MIDDLE-ML we have a fine-tuned parser and in-memory data-structure for efficiently handling very large designs. Secondly, MIDDLE-ML is used as our standard language for the description of other tool-specific languages and libraries; all of them use the same in-memory data-structure as Colif. Thirdly, besides the parser, a complete Application Programming Interface (API) can be generated automatically for all languages based on MIDDLE-ML.

4.2. Object model

As a very general view, Colif is a model that sees the system as a set of subsystems, represented as *modules*, that exchange data. Subsystems are contained in hierarchical modules and the communication network is composed of *nets* and *ports*, as depicted in Figure 1. A fundamental decision in Colif's design was to clearly separate behavior from communication. Subsystems are contained in hierarchical modules and the communication network is composed of generalized nets and ports. The behavior of each subsystem could be modeled as a set of concurrent interactive tasks or hidden from synthesis by instantiating a blackbox. Colif' system model is a very abstract view of a system. The only assumption made is that there is no dynamic creation of objects. An executable model can be obtained by mapping the different parts of the system to a suitable language. This may be used to generate a co-simulation model at different stages of the design flow.

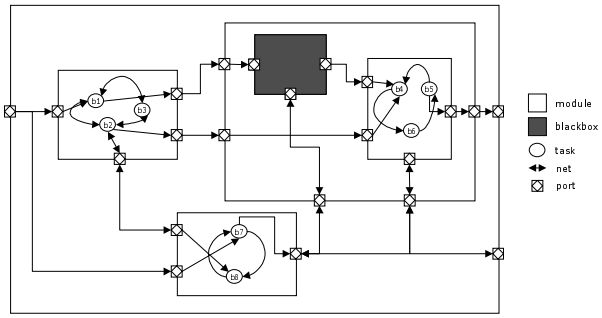


Figure 1 - Colif object model

Modules' interface is composed of *ports* and the set of possible operations they could realize. This object view of the port is a generalization of the concept of a hardware I/O port. Port operations (*methods*) can be further decomposed into two sets: (i) operations allowed to be

performed by the module that holds the port (*internal methods*), and (ii) operations that could be called from the rest of the system (*external methods*). The content of a module could be of two kinds: (i) a network of sub-modules or (ii) a network of interactive tasks. A module only describes the hierarchy of the system. A *task* is a module that is a leaf in the hierarchy of modules; its content is a reference to an externally defined behavior. A *blackbox* is a special module that has no specified content.

Colif's main strength is to model communication at different abstraction levels, assuring structural coherence while mutating from a system-level specification into an RT-level implementation. Synchronization is defined directly by the protocol used in a *per connection* fashion, there is no predefined synchronization mechanism. A *net* must be considered as an abstraction of a communication channel because no specific protocol is implied anyway. Consequently, it represents only the fact that two entities in the system could exchange data in some unspecified way in space and time. Nets will get much better defined semantics when the model is refined and specific communication protocols attached to them, what we could call *late binding* of communication semantics. Net semantics at different abstraction levels with respect to the media type, the net behavior and the format of the data transmitted can be summarized like this:

- At the service level, module interfaces are at the highest abstraction level. They are composed of ports that give access to an abstract network. These ports provide services of a certain type. Communication is modeled as an abstract network that routes service requests. These concepts hide all other communication details.
- At the message level, module interfaces are composed of access ports to active channels. These ports provide high-level procedure calls to access the network (like send and receive). Active channels that constitute the communication network can process protocol conversions on generic data types.
- At the driver level, access ports are detailed to the logical level and work with fixed data types. Communication media is modeled as abstract wires that use a driver-level protocol and can hide details like interrupt handling and address decoding. Managing communication details can lead to the addition of specific modules at this level; they are called *communication controllers*.
- At the RT-level, the system is described in terms of registers, combinatorial circuits and control circuits. All communication protocols are detailed to the level of timing diagrams for the hardware ports concerned in each data transfer. Interrupt handling/address decoding are explicitly described at this level. Communication media are physical wires and data are manipulated as bit-vectors.

The three basic concepts of Colif's model: the hierarchical structure of modules, the generalized port and the late binding of communication semantics characterize it as a highly flexible model. This is by large the most suitable feature for the design of embedded systems and was one of the main guidelines in Colif's design. Of course, to have this flexibility we must pay the price of the lack of formal analysis based on well-defined execution semantics. Our choice for flexibility is justified by the nature of problem we want to solve that is the synthesis and design of complex heterogeneous embedded systems using multiprocessor SOC architectures. Usually, the specification of such a system is decomposed into multiple parts that are designed by different teams in a modular design flow.

5. Colif models at different abstraction levels

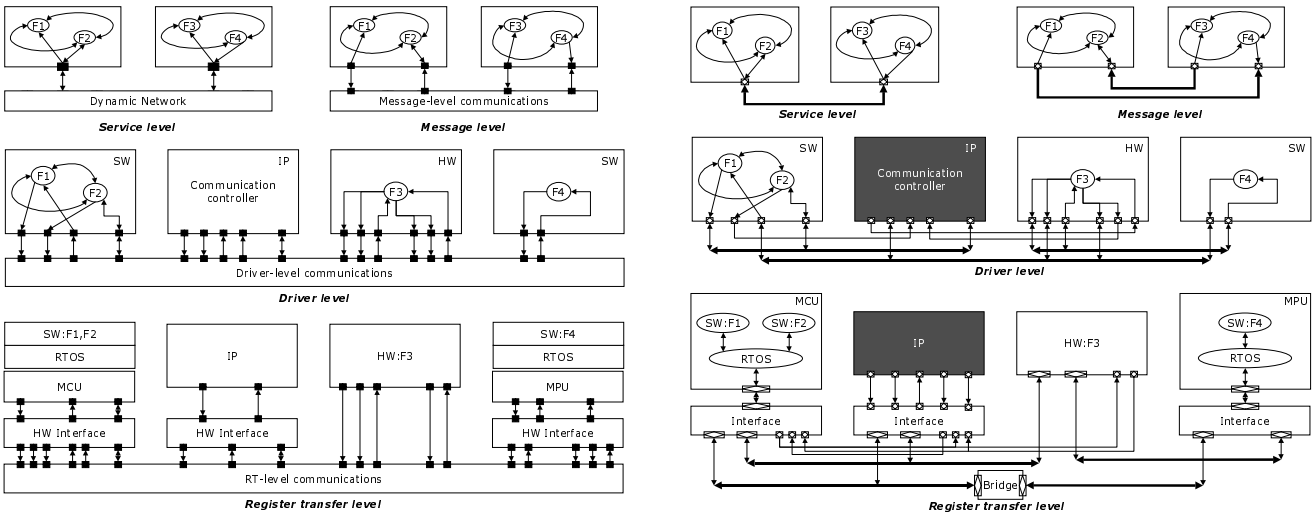
Figure 2 illustrates the deployment of Colif at different abstraction levels. The four abstraction levels used for system specification are shown conceptually by example abstract architectures in Figure 2(a). On the right side, their corresponding Colif models are shown in Figure 2(b).

The example specification at the service level uses a *dynamic routing network* [12] accessed by outgoing *request ports*, and connects those requests to *resolved service ports*. All processes and tasks in a hierarchical module can access the requested ports of its encapsulating module. In Colif (see Figure 2(b)), request ports are modeled as ports offering internal methods that allow tasks to request a service. Service ports offer services as external methods, so a Colif port could act as a request port and service port at the same time. Nets connect request ports to all service ports that could possibly execute the requested service or list of services. Service resolution (ORB) can be modeled as net behaviors.

At the message level, our example specification consists of point-to-point active channels between two hierarchical modules. Internal processes can send messages containing abstract data types. Active channels may specify any conversion according to high-level protocols and abstract data types. In the corresponding Colif model (see Figure 2(b)) nets have behaviours that are linked to an external protocol library that contains detailed behavioral descriptions of any given high-level protocol used in active channels. Ports provide internal methods like send/receive of abstract data types so tasks dispose of a high-level interface to active channels.

On the driver level, communication goes through logical buses, and interfaces realize the protocols and stores the data while protocols' instructions are executed. Hence, a communication controller may be needed to control access to the bus and alleviate the protocols. Our example (see Figure 2(a)) shows an abstract architecture made of four modules interconnected through logical buses. Each module has been mapped on a processor in this abstract architecture. This may be a software

processor (e.g. DSP or a microcontroller executing software), a specific hardware processor or an existing IP core (global memory, peripheral, bus controller, etc.). Logical buses are abstract channels that transfer fixed data types (e.g. integer, real) and may encapsulate driver-level protocols (e.g., handshake or memory mapped I/O). In the corresponding Colif model (see Figure 2(b)), ports offer internal methods like send/receive but, at this abstraction level, they work only with fixed data types, and net behaviours are now linked to lower level communication protocols. The IP core acting as a communication controller is represented as a blackbox.



(a) Abstract architectures **(b) Colif models**

Figure 2 - System models at different abstraction levels

The last model shows a typical RT-level architecture where software modules are mapped onto specific processors running some kind of real-time operating system (RTOS). Communication between the different blocks is made through physical wires using the protocols chosen. Modules are connected to bus' signals through hardware interfaces where necessary, and they may include controllers and buffering. An IP core was encapsulated within an interface in order to accommodate different communication protocols. The corresponding Colif model uses nets with behaviours linked to cycle-accurate protocols to represent the physical buses. Hierarchical ports (the larger ones in Figure 2(b)) are used to model logically related ports, e.g., multi-bit address/data lines. Software tasks execute I/O operations through RTOS system calls, which are represented as nets with behaviours linked to the remote procedure call protocol. Finally, buses operating at different clock frequencies may require a *bridge* module to realize more efficient data transfers.

6. Evaluation

Table 2 compares the main characteristics of Colif meta-model and today's most advanced models/languages for system-level design. Simulation-based languages like SystemC [3] are good frameworks for system validation, but too much details are required to build an executable model. The main disadvantage is that design specification must start at a relative low level. The newest SystemC version enables higher levels of modeling abstraction by offering a new untimed functional model and abstract communication protocols. Old versions provided only the traditional discrete-event simulation paradigm where every task holds its internal state between events. A remote procedure call task model was added to support the simulation of the highest abstraction level. Yet, we will be ever tied with C++ syntax when using SystemC.

With Rosetta [8], designers are still tied to a new specific syntax but they get the freedom of defining new task models by writing new *facets* or combining existing ones. Communication models' flexibility is restricted because they must follow the rules that assure facet composition and interoperability. The main abstraction focus in Rosetta is the management of design constraints while the designer operates with facets composition rules. There is still no parser or simulator for Rosetta since the language specification is not yet finished.

OVI-SRM, as the name says, is a semantic model for embedded system design that is independent of any particular specification syntax. The main abstraction focus is the scheduling of run-to-completion tasks into embedded processors. The only communication model accepted is the point-to-point shared-memory interconnection so it is overly restrictive. Furthermore, communication modeling is restricted to a single abstraction level.

GSRC abstract syntax [6] is a meta-model aimed at tool integration, it can represent diverse design models: netlists, state transition diagrams, block diagrams, object models and graph structures. For system-level modeling it could be used to represent different "Models of

Computation" (MoC) or semantic domains that could be deployed in a composite simulation of the system. As with Rosetta, communication models must respect composite simulation semantics and are very restrictive in general. MoML [14] is an XML derivative that honors this abstract syntax. It is intended to be a generic modeling-markup language aimed at tool interoperability. Without semantics, MoML could not be compared with Colif because it would be meaningless. In this comparison, we used Ptolemy-II [15] semantics. There are many other models based on MoC composition: Abstract Codesign Finite-state Machine (ACFSM) [2] models abstract communication using infinite-size queues; El Greco uses a Cyclo-static Dataflow (CSDF) [16] and low-level fixed-size FIFOs for communications; System Properties Intervals (SPI) [17] is an abstract process network model with system properties annotated as intervals, its main focus is on performance estimation, and it also uses FIFO buffers for communications.

Table 2 - Comparing design languages/models

Model characteristics	Design language/model				
	SystemC1.1	**SLDL-Rosetta**	**OVI-SRM**	**GSRC**	**Colif**
Task model	Internal state/RPC	Definable facets	Run-to-completion	MoC composition	Interactive
Communication model	Multiple/ multi-level	Restricted by facets interoperability	Point-to-point (shared memory) single-level	Restricted by MoCs interoperability	Multiple/ multi-level
Abstraction focus	Communication protocols	Constraints management	Task scheduling	Abstract syntax	Communication semantics

Colif's interactive task model is an effective way of making task schedule more or less independently from the interaction with the communication models. As stated in [2], separating communication and task behavior is essential to dominate system design complexity. Multiple communication models are supported given that their behavioral could be described in an external communication library at multiple abstraction levels.

7. Conclusion and future work

The main contribution of this paper is a new meta-model focused on communication semantics at multiple abstraction levels and based on an interactive task model. Colif is one of the first system models to separate clearly task behaviour from communication behaviour. Being a meta-model allows the utilization of simulation-based languages like SystemC as a back-end tool for design verification. The automatic generation of an API for Colif guarantees that it could be easily extended to support other aspects of system modeling, e.g., constraints.

Currently, Colif has been used in a variety of synthesis tasks within an embedded multiprocessor design flow, specially interface synthesis, RTOS synthesis and co-simulation of hardware-software system models. In the future, it will be used in mixed-level executable model generation, mixed-level communication synthesis and synthesis of distributed shared memory architectures.

8. Bibliography

[1] W. Wade, "Embedded chips diverge on multiprocessing path," EE Times, Issue 1118, June 2000.

[2] M. Sgroi, L. Lavagno and A.S. Vincentelli, "Formal Models for Embedded System Design," IEEE Design & Test of Computers, vol.17 no.12, April-June 2000.

[3] J. Gerlach, W. Rosenstiel, "System Level Design Using the SystemC Modeling Platform," SDL 2000.

[4] D. Verkest, K. Van Rompaey, I. Bolsens, H. De man, "CoWare - A Design Environment for Heterogeneous Hardware/Software Systems," Design Automation for Embedded Systems, vol. 1, n°. 4, pp. 357-386, 1996.

[5] C. K. Lennard, "System-level models explore terrain," EE Times, June 13, 2000.

[6] A. Ferrari and A. Sangiovanni-Vincentelli, "System Design: Traditional Concepts and New Paradigms," Proc. of the ICCD, Austin, TX, USA, pp. 1-12, October 1999.

[7] D.D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC Specification Language and Methodology," Kluwer Academic Publishers, Boston, MA, ISBN 0-7923-7822-9, March 2000.

[8] P. Alexander, R. Kamath, D.Barton, "System Specification in Rosetta," Proc. of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2000.

[9] Keutzer, K.; Newton, A.R, "The MARCO/DARPA Gigascale Silicon Research Center," Proc. of ICCD, 1999.

[10] International Telecommunication Union, CCITT - Specification and Description Language (SDL), Mars, 1993. Recommendation Z.100.

[11] W. Glunz, T. Kruse, T. Rossel, and D. Monjau, "Integrating SDL and VHDL for system level specification," in Proc. Of the Conference on Hardware Description Languages, April 1993.

[12] K. Svarstad, N. Ben-Fredj, G. Nicolescu and A.A. Jerraya, "A Higher Level System Communication Model for Object-Oriented Specification and Design of Embedded Systems," submitted to ASP-DAC 2000.

[13] Object Management Group, "CORBAservices: Common Object Services Specification," Dec 1998. Available at http://www.omg.org/.

[14] E.A. Lee and S. Neuendorffer, "MoML—A Modeling Markup Language in XML—Version 0.4," Tech. Rep. UCB/ERL M00/12, March 14, 2000.

[15] E.A. Lee et al., "Ptolemy-II: Heterogeneuos Concurrent Modeling and Design in Java," Tech. Rep. UCB/ERL No. M99/40, CA 94720, July 19, 1999.

[16] J. Buck and R. Vaidyanatham, "Heterogeneous Modeling and Simulation of Embedded Systems in El Greco," CODES, San Diego, CA, 2000.

[17] R. Ernst, D. Ziegenbein, K. Richter, L. Teich, "Hardware/Software Co-Design of Embedded Systems - The SPI Workbench," Proc. IEEE Workshop on VLSI'99, pp. 9-17, Orlando, 1999.