

System-Level Design Flow Based on a Functional Reference for HW and SW

Walter Tibboel, Victor Reyes, Martin Klompstra, Dennis Alders
NXP Semiconductors

High Tech Campus 37 - Eindhoven - The Netherlands

Walter.Tibboel@nxp.com

ABSTRACT

Heterogeneous MPSoC design where flexible programmable cores are combined with optimized HW co-processors is a quite complex and challenging task. In this paper, we present a system-level design flow that uses a single *functional reference* for modeling both HW and SW. The models follow an interface-centric design approach based on the TTL interface (Task Transaction Level). TTL models are applied at all three abstraction levels of the design flow: functional, architecture and implementation level. The TTL model at the functional level serves as the functional reference. HW implementations are generated from refined TTL models by behavioral synthesis tooling. Likewise, SW implementations are supported by source code transformations. Both the HW and SW implementations are verified against the functional reference. Details of the complete flow are presented in the paper through an MP3 case study.

Categories and Subject Descriptors

C.1.2 [Multiple Data Stream Architectures]: Interconnection architectures. D.1.3 [Concurrent Programming] Distributed programming. B.4.3 [Interconnections] Interfaces. B.5.2 [Design Aids] Automatic synthesis.

General Terms

Design, Performance.

Keywords

System-level design, Functional reference, Task Transaction Level, Platform interface, Code transformation, Behavioral synthesis

1 INTRODUCTION

Nowadays embedded systems have to cope with more complex and demanding applications, as well as with more stringent low-cost and low-power requirements. Such embedded systems are heterogeneous Multiprocessor System-on-Chip (MPSoC) where flexible programmable cores are combined with optimized HW

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2007, June 4–8, 2007, San Diego, California, USA.

Copyright 2007 ACM 978-1-59593-627-1/07/0006 ...\$5.00.

co-processors. Hence, heterogeneous MPSoC design is a quite complex and challenging task.

Traditionally, embedded SW and dedicated HW co-processors are developed by separate design teams. Ultimately SW and HW have to be integrated in order to verify the application functionality. Often this integration is performed very late in the design process using detailed RTL models, or even a sample of the chip. Error detection and correction of the RTL models is quite a time consuming task and usually causes inconsistencies between the RTL models and the models at higher abstraction levels, since the changes are not propagated up.

This paper presents a top-down system-level design flow that uses a single *functional reference* for modeling both HW and SW. The models follow an interface-centric design approach based on the TTL interface [1]. TTL models are applied at all three abstraction levels of our design flow: functional, architecture and implementation level. The TTL model at the functional level serves as *the functional reference*.

TTL functional models are seamlessly refined down to implementation via source code transformations for SW and high-level behavioral synthesis for HW. Additionally the interface synthesis capabilities of behavioral synthesis tooling is applied onto TTL HW models. TTL implementation models are directly connected to the more abstract TTL models of *the functional reference* to verify their functional correctness. In this way, models at different abstraction levels are kept consistent, which is a valuable contribution of this flow.

Section 2 discusses related work. Section 3 explains the TTL models at different abstraction levels. Sections 4 and 5 disclose details of the HW and SW implementation process. Section 6 presents an MP3 case study and, finally, conclusions are drawn in Section 7.

2 RELATED WORK

Several innovative design methods for multiprocessor systems have been developed [1][4][5][6]. The authors of [1] present a methodology that supports homogeneous programming styles for multiprocessor platforms composed out of heterogeneous HW and SW processing elements. Unlike the approach we use, one model for HW and SW, this methodology is based on two complementary programming models.

In contrast to methodologies using proprietary languages (e.g. SpecC [2], Polis [4]), our approach is based on SystemC, the de facto industry standard for Electronic System Level (ESL) design,

which eases the integration of commercial (behavioral) synthesis tooling. The usage of SystemC in behavioral synthesis for complex system development is demonstrated in [14].

Our flow has much in common with Ocapı [5] and the work of TİMA [6]. Likewise, we also use the Kienhuis Y-chart approach [4] to separate functional behavior from architecture details. A single model for HW and SW based on TTL, is the differentiating factor of our approach. TTL is based on previous Philips work [7] [8].

Interface synthesis (introduced in [10]) of behavioral synthesis tooling [12][13] is applied onto the TTL models to provide a smooth path from a *functional reference* to HW implementation.

3 DESIGN METHODOLOGY

Our design flow applies a top-down approach focus on the efficient implementation of media processing applications. Although this flow might be applied for complete platforms, it suits better for sub-systems design.

3.1 Functional level

The objective of the functional level is to capture the behavior of a target application in a TTL executable specification and verify its functional correctness via simulations; neither architecture nor time is taken into account at this level. Since only un-timed functional models are taken into account, high-speed simulations can be achieved.

TTL models are described as (hierarchical) process networks, which contain tasks executing concurrently. Inter-task communication is realized via unidirectional communication channels connected to task ports. TTL channels and its associated ports are strictly typed, i.e. one can only communicate one kind of data type. Thus, communication and parallelism are modeled explicitly. A simple TTL model example is shown in Figure 1.

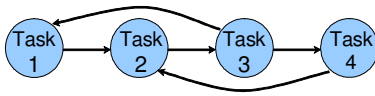


Figure 1 Functional model

The TTL specification offers a set of abstract interfaces for synchronization and data transfer operations. The set of interfaces support multiple paradigms for inter-task communication, which combined can provide efficient SW/SW, HW/HW or HW/SW communication.

The TTL model at this level serves as the *functional reference* for both the HW and SW tasks at the lower abstraction levels. Since TTL interfaces and semantics are consistent at all three abstraction levels, a TTL task of the *functional reference* can be seamlessly replaced with the corresponding refined task at the lower abstraction levels. This is key to enable the functional correctness verification of the refined task using the single functional reference.

To be more precise, tasks can be replaced as long as the number of ports of a task, and the associated data type of each of the ports is not modified. With this restriction it is e.g. still allowed to change the synchronization and communication grain size, since

these implementation details remain completely hidden within the TTL interfaces. By contrast, a change in the functionality of the process network should result in a modification of the functional reference as well, since apparently a different specification is required.

3.2 Architectural level

The objective of the architectural level is to enable performance analysis and design space exploration. That is, decisions about the HW/SW partitioning, communication and synchronization implementation, etc.

In order to obtain the TTL architectural model, the *functional reference* is mapped onto an abstract SystemC virtual prototype (VP) of the HW platform (see Figure 2). The mapping step first requires refinement of the functional model into HW and SW tasks. Such refinements may consist of, for instance, switching to a different TTL interface type, since some types are better suited for HW and others for SW. In the flow this is still a manual procedure. Although the amount of code changes required is kept to a minimum, through the high-level interfaces, still an error can be introduced during the refinement of the tasks. Therefore, before mapping them on the architectural model the refined tasks are verified against the *functional reference*.

After mapping, the performance is measured by tracing specific metrics like e.g. busload, communication buffer filling, processor load, and communication delays. Graphical overviews are generated showing clearly the contributions of different tasks/ports in the system. Details on performance analysis and visualization are out of the scope of the paper.

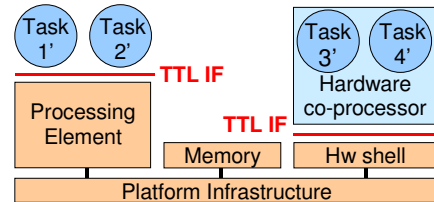


Figure 2 TTL model mapped on a platform

Figure 2 shows the refined SW task1' and task2'. These SW tasks are executed on an abstract SystemC processor model, named Processing Element (PE). A PE is a placeholder for multiple TTL tasks and provides multi-tasks scheduling services, interrupts and preemption features, as well as a configurable implementation for the TTL interface. At the architecture level, PEs are still abstract in the sense that their instruction set is not determined. Details of these abstract architectural models can be found in [9]. Refined Task3' and Task4' are mapped to a HW co-processor. To connect these refined TTL tasks to the platform a so-called HW shell is introduced. This HW shell, modeled in SystemC, implements the TTL interface and translates it into platform-optimized calls. In this way architecture specific details are separated from the HW co-processor functionality.

3.3 Implementation level

The objective of this level is to come to an efficient implementation for the TTL tasks. RTL code is generated for the

HW co-processors and embedded SW created for the tasks running on the PE.

Further refined TTL HW tasks serve as input for a behavioral synthesis tool, which generates HDL RTL and a Cycle-Accurate (CA) SystemC model. The result is shown in Figure 3 where the abstract TTL HW tasks are replaced by a CA model of the HW co-processor. The generated co-processor communicates via a TTL interface at signal-level while the abstract HW shell uses TTL IMCs (Interface Method Calls). Both use the same TTL interface types, only the abstraction level differs. A transactor is introduced to bridge the abstractions levels during simulations.

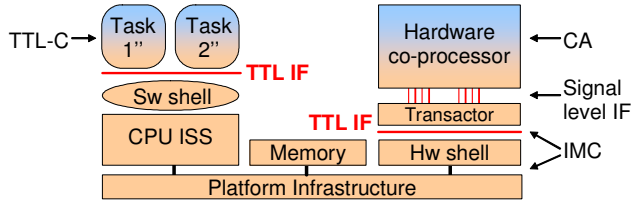


Figure 3 Implemented TTL running on a platform

Similarly, a transactor can also connect the CA co-processor directly to task1 and task2 of the functional reference, to verify the functional correctness of the HW implementation. Such a seamless mix of implementation and functional reference models distinguishes this approach from other solutions.

Further, an Instruction Set Simulator (ISS) model replaces the PE on the VP to execute the SW tasks (see Figure 3). This requires a SW shell, similar to a HW shell, which implements the TTL IMCs. Such a SW shell can include a lightweight multithreaded kernel or can be implemented on top of a RTOS. Both the SW tasks and the SW shell are cross-compiled to the embedded processor. Most compilers for embedded systems only accept C code, whereas the TTL model is C++ code. The SCATE tooling (see Section 5) is able to transform a TTL C++ model into an equivalent TTL C model.

The SW and HW shells are typical library components, which can be reused when the platform is applied for new applications.

4 HARDWARE SYNTHESIS

This section explains the integration of behavioral synthesis in our system-level design flow. We demonstrate that the TTL models used in the functional simulations also can be synthesized using a behavioral synthesis tool. This section also describes how the HW implementations are kept consistent with the functional reference.

Two behavioral synthesis tools [12] and [13] have been evaluated. Section 4.1 describes the behavioral synthesis source input and the design output. Section 4.2 gives detailed information about the interface synthesis technology, by explaining a TTL port example. Interface synthesis is a typical methodology for behavioral synthesis tooling to convert IMC interfaces into signal-level interfaces.

4.1 Behavioral synthesis input and output

The behavioral synthesis tooling is used to transform HW tasks at architecture level into RTL at the implementation level (see

Figure 4). At the architecture level the co-processor is described in TTL C++ with IMC interfaces. This model serves as the input model for the behavioral synthesis tool. The output of the tool is a co-processor at RT level with signal-level interfaces.

As shown in Figure 5, behavioral synthesis tools support the separation of task functionality from implementation details (like e.g. the tool pragmas), which fits perfectly in our methodology.

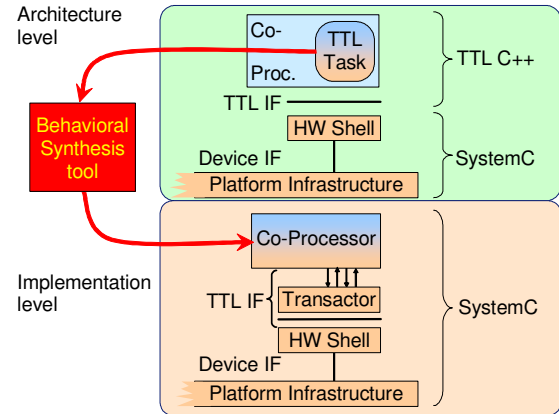


Figure 4 Position of behavioral synthesis

Manual refinements of the task functionality could be e.g. rewriting loop structures and adding labels that refer to tool specific optimization pragmas. The refined task is co-simulated with the other tasks of the functional reference to secure the functional correctness.

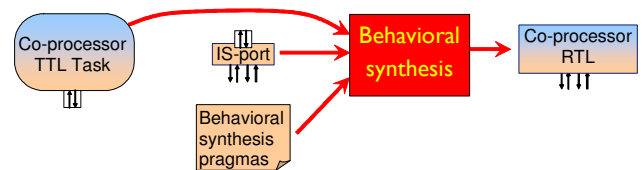


Figure 5 Behavioral synthesis input and output

The separated implementation details in Figure 5 contain tool specific pragmas to optimize the behavioral synthesis results. Optimization examples are e.g.: loop unrolling, pipelining, controlling the amount parallelism in the data path, mapping arrays into memory or into registers, and mapping multiple arrays in one array.

The second part of the implementation details is the Interface Synthesis port (IS-port). The IS-port converts the IMC interface of the functional model, into a signal-level interface for the RT level model (see Figure 5). Both sides of the IS-port have the same TTL interface; only the abstraction level differs (it can be seen as the counterpart of the transactor explained in Section 3.3). The IS-port is tool specific and TTL specific. The platform design team has to create and support the IS-ports.

4.2 TTL interface synthesis example

In this section we present in detail a TTL interface at two abstraction levels. We chose as example the TTL RB (Relative Blocking) interface type, which separates the synchronization and

data transfers, using blocking semantics for the synchronization operations, and has scalar and vector support. More information about the RB type can be found in [1]. Section 4.2.1 discusses the IMC interface, Section 4.2.2 discusses the corresponding signal interface and Section 4.2.3 explains the IS-port that is required to bridge these two interfaces.

4.2.1 RB-in modeled with IMC

The RB-in interface has the following IMC primitives:

- $reAcquireData(p, n)$ Re-acquire n full, acquired or unacquired tokens from channel connected to port p .
- $load(p, d, v)$ Copy the value of the token with distance d to the oldest acquired token into value v .
- $load(p, d, v, n)$ Copy the value of the tokens with distance $d, \dots, d+n-1$ to the oldest acquired token into vector v of length n .
- $releaseRoom(p, n)$ Release the n oldest acquired tokens to the channel connected to port p .

Before a consumer task can obtain data ($load$) from the channel, it first has to acquire the data tokens. Typically the consumer performs multiple $loads$ per acquired data block. The reserved data space is released after the $releaseRoom$ call.

The order of these TTL calls is relevant, e.g. the $reAcquireData$ has to occur before the $load$. In general a behavioral synthesis tool can change the order of operations to improve the performance results. The tool is not aware of a relation between $reAcquireData$ and $load$ and might change their order, which leads to erroneous behavior. With some additional effort, both tools evaluated are able to keep the order of the operations.

4.2.2 RB-in modeled with signals

At the implementation level an IMC has to be converted into a bit-true and cycle-true protocol. Table 1 shows the mapping of the IMC arguments to the hardware signals.

Table 1 RB-in from IMC to signals

TTL RB-in IMC	Hardware signals			
	type (out)	tokens (out)	distance (out)	data (in)
$reAcquireData(p, n)$	0	n	-	-
$load(p, d, v)$	1	1	d	v
$load(p, d, v, n)$	2	n	d	$v[0] .. v[n-1]$
$releaseRoom(p, n)$	3	n	-	-

The $type$ signal informs which RB-in IMC is currently active. The $tokens$ signal carries the n argument and the $distance$ signal carries the d argument. The p argument of an IMC is the specific port instance used by the IMC (a task can have multiple RB-in port instances). At the RT level this is managed by wires connected to a specific port instance. The $data$ signal is only used by the $load$ calls. It can carry a single token value (scalar $load$) or

a vector of tokens, one at the time (vector $load$). The direction (in, out) of the signals is from the task's perspective.

Figure 6 shows two examples of the clock cycle true protocol. The figure shows two scalar $load$ transactions (type=1, tokens=1). The value in the $data$ waveform refers to the data token of the specific distance.

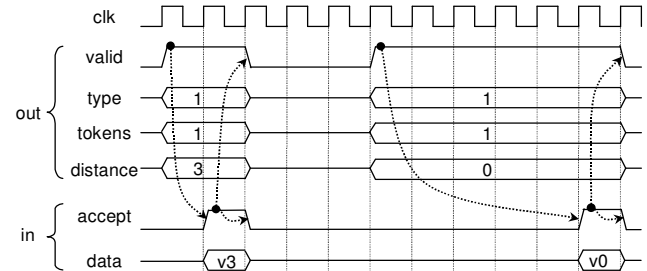


Figure 6 RT level protocol of the signals

A $valid/accept$ handshake pair controls the synchronization of the value transfers on the request and data signals, see Figure 6. The $valid$ pulse indicates that the $type$, $tokens$ and $distance$ signals contain proper values. The $accept$ pulse indicates that the requested token value is on the data signal. In the example the signals grouped by out (in) are driven by the HW co-processor (HW shell). On the left (right) hand side a fast (slow) response of the HW shell is depicted.

As mentioned above, before a load operation can be performed data has to be available in the channel. Testing the availability of the data is done by the $reAcquireData$ IMC. If data is not available this IMC will block. Using the IMC, this means that the function call does not return until the requested amount of tokens is in the channel. At the signal-level, the $accept$ pulse indicates the data availability. During the period the $reAcquireData$ IMC is blocked, only the interface between the co-processor and the HW shell is blocked; the platform infrastructure (bus) remains available for other devices.

4.2.3 IS-port

In the abstract TTL models, each TTL port declaration is templated by the token data type (see upper part of Figure 7).

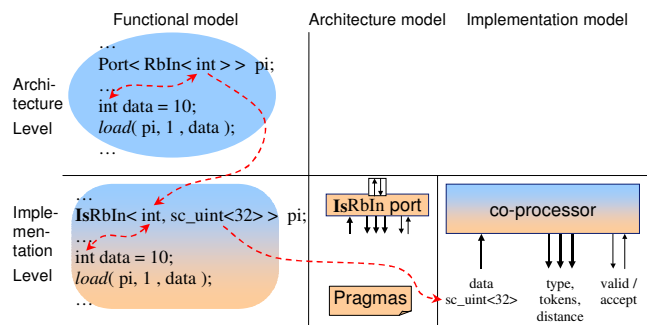


Figure 7 Interface synthesis

In case the TTL model is used as behavioral synthesis input the port declaration is replaced by the IS-port declaration. The IS-port

has two template arguments: a data type used inside the task, and a corresponding type of the data signal used outside the task. The IS-port performs the type casting if the two types differ. The latter is a SystemC bit restricted data type. The former can be a SystemC type or a simple C data type. The evaluated behavioral synthesis tools are capable of handling IS-ports with template arguments.

As discussed above, the port declaration in the TTL task at architecture level differs from the IS-port declaration at implementation level (see the *pi* declarations in Figure 7). Still the TTL calls in the model are exactly the same (see the *load IMC* in Figure 7). Therefore the amount of details that has to be added in the code is minimized, which reduces inconsistencies between the models of different abstraction levels.

5 SOFTWARE SYNTHESIS

The purpose of software synthesis is to perform source code transformations to support the system designer during the refinement of SW tasks towards an efficient implementation.

The use of tooling can automate repetitive and error prone steps in the design flow. In [15] tooling has been described to support source code transformations for multiprocessor architectures optimizing the cost of memory usage, synchronization cycles, data transfer cycles, and address generation cycles. In this article we use the same tooling for software synthesis. The name of this tool is SCATE (Source Code Analysis and Transformation Environment) [15].

Implementation models in general do not often use hierarchy for efficiency reasons. (Partial) flattening of a process network model at source code level can be labor intensive. For this reason this structural transformation has been implemented in SCATE.

As stated above at the functional level TTL C++ is being used because of its cleaner descriptions, better encapsulation and its natural fit with SystemC. Unfortunately, we have to use C to link to software compilers for embedded processors and hardware synthesizers when needed, since in most cases C++ is not fully supported as input language. SCATE assists in the transformation from TTL C++ to the C language, though, it is not the goal to implement a fully functional C++ to C conversion for every construct of the language. Instead, only the functionality that is needed in the TTL C++ to C translation context has been implemented. We support the conversion of the TTL infrastructure, while it is assumed that the functional code of the processes is plain C. Code manipulations performed are:

- Conversion of a class definition to a struct definition.
- Introduction and propagation of the 'this' pointer.
- Moving declarations to the beginning of a scope.
- Reduction of scoped declarations and expressions.
- Static member variables, member functions declarations/implementations placed within the class definition are made globals.

Some constructs are simply eliminated from the source code since a full-fledged solution is too expensive,

- C++ library files, like iostream.

- The using directive.
- The reference (&) operators.
- The const function qualifiers.

In the future we intend to automate also source code transformations to ease the connection to behavioral synthesis, as discussed in the Section 4.

6 MP3 case study

Our methodology is demonstrated by implementing an MP3 application on an audio sub-system. This application consists of a stereo MP3 decoder and two Sample Rate Converters (SRC). The sample rate conversion is needed in the system because the output should have a 44.1 kHz sample rate, but the input can be of any sample rate specified by the MP3 standard. Like discussed, the product design starts with *the functional reference*, see Figure 8. This model is executed in a fast and un-timed simulation environment to obtain reference data.

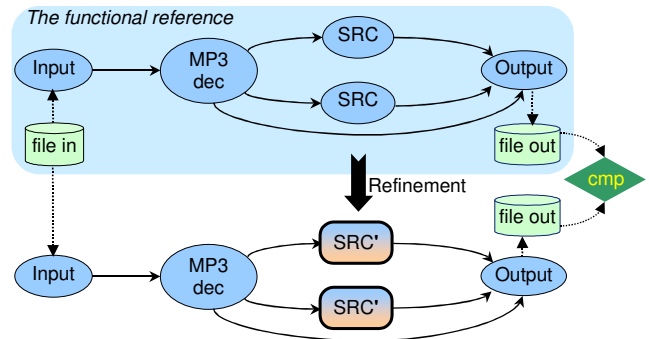


Figure 8 MP3 case at functional level

The input task reads MP3 data from a file, the output task stores the computed PCM data at in the responses file. In reality the input task is storage retrieval, the output task would be replaced by post processing task, like volume, balance etc.

The MP3 decoding is mapped onto an abstract processor model and the SRC's are mapped to dedicated hardware. At this stage, architectural exploration is done, like e.g. changing the TTL interface type, synchronization granularity and buffer sizes.

At the implementation level the MP3 TTL C++ code is automatically translated to TTL C (see Section 5). The MP3 TTL C model together with a SW shell is cross-compiled to an ARM processor. The channels between the MP3 decoder and SRC connect the software with the hardware co-processor. The channel buffers are located in the shared memory.

The SRC task has been refined into valid input for a behavioral synthesis tool. Next, the refined SRC task replaces the SRC task in *the functional reference* of Figure 8. In this way the SRC HW implementation is kept consistent with *the functional reference*, which prevents time-consuming integration problems at the end phase of the MPSoC design.

The SRC was synthesized using behavioral synthesis tooling [12]. The used chip technology is CMOS12. Table 2 shows a result summary report, generated by the tool. The output sample rate

could easily fulfill the 44.1 kHz specification; therefore the design was not optimized for speed, but for smaller area. The design runs at 133 MHz clock frequency.

Table 2 SRC synthesis results summary

Value	Result
Total area (square μm)	51,914
Register bits	541
Cycle time (ns)	7.5

After synthesis both a CA SystemC model and a RTL Verilog model are generated. These models have exactly the same timing behavior, but the former is optimized to fast SystemC simulations. The CA SystemC model is integrated in the platform, as shown in Figure 9. The simulation model also contains the cross-compiled MP3 decoder running on an ARM ISS. The implementation-level simulation generated the same bit-true reference data (file out) as the *functional reference* (see Figure 8).

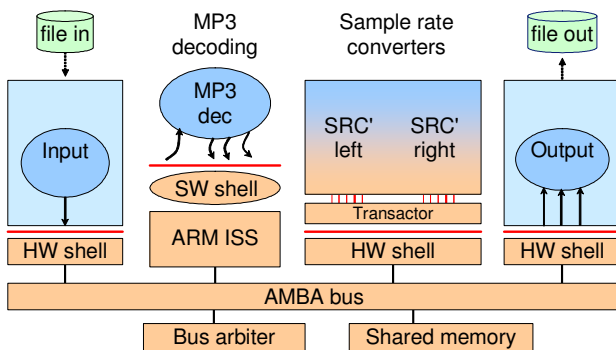


Figure 9 MP3 case at implementation level

The simulation time at implementation-level is about 200 times longer compared to functional level. The cross-compiled MP3 decoder running on the ARM CCM (Cycle Callable Model) is the main cause for the slow simulation speed at the implementation level. The simulation time at the functional level holds for both the *functional reference* and refined TTL tasks (see Figure 8). In order to improve the productivity, the validation of the functionality should be done in the fast functional simulations, and only move down to the slow implementation-level simulation to verify the performance and to do the final functional verification.

7 CONCLUSIONS

This work targets the increasing HW/SW integration problem, by means of a single *functional reference* model for both HW and SW. This model implements task-level interfaces, based on the TTL specification, which are consistently used along the three abstraction levels of the presented methodology. Both HW and SW designers verify their TTL implementations against the *functional reference*. This assures models at different abstraction layers are kept consistent.

Interface synthesis of behavioral synthesis tooling is applied onto the TTL models to provide a smooth path from a *functional reference* to HW implementation. TTL SW implementation is covered by means of innovative source code transformation techniques.

8 ACKNOWLEDGMENTS

We thank Steve Anderson from Forte Design Systems and Thomas Bollaert from Mentor Graphics. This work is partially sponsored by the European Commission in the ITEA 04006 MARTES project.

9 REFERENCES

- [1] P. van der Wolf et al., Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach. In *Proc. of CODES+ISSS'04*, 2004 [<http://portal.acm.org/citation.cfm?id=1016720.1016771>]
- [2] P. Paulin et al., Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In *Proc. of CODES+ISSS'04*, 2004 [<http://portal.acm.org/citation.cfm?id=1016735>]
- [3] D. D. Gajski et al., SpecC: Specification Language and Methodology. Kluwer Academic Publishers, 2000.
- [4] F. Balarin et al., Hardware-Software Co-Design of Embedded Systems, the POLIS Approach. *Kluwer*, 1997 [<http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=271072>]
- [5] B. Kienhuis et al., A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach. In the *LNCS series Vol. 2268, SAMOS*, 2001. [<http://portal.acm.org/citation.cfm?id=691571>]
- [6] G. Vanmeerbeeck et al., Hardware/Software partitioning for embedded systems in OCAPI-x1. In *Proc. of CODES+ISSS '01*, 2001 [<http://portal.acm.org/citation.cfm?id=371665&coll=portal&dl=ACM>]
- [7] A. A. Jerraya et al., Programming models and HW-SW Interfaces Abstraction for Multi-Processor SoC. In *Proc. 43th DAC*, 2006 [<http://portal.acm.org/citation.cfm?id=1146981&dl=GUIDE&coll=&CFID=15151515&CFTOKEN=6184618>]
- [8] E.A. De Kock et al., YAPI: Application modeling for Signal Processing Systems. In *Proc. 37th DAC*, 2000 [<http://portal.acm.org/citation.cfm?id=337292.337511>]
- [9] M.J. Rutten et al., Eclipse: Heterogeneous Multiprocessor Architecture for Flexible Media Processing. In *Proc of IPDP2002*, 2002 [http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1016517]
- [10] V. Reyes et al., A unified system-level modeling and simulation environment for MPSoC design: MPEG-4 decoder case study. In *Proc. DATE 06*, 2006 [<http://portal.acm.org/citation.cfm?id=1131608>]
- [11] J.A. Rowson et al., Interface-Based Design. In *Proc. 34th DAC*, 1997 [<http://portal.acm.org/citation.cfm?id=266060>]
- [12] <http://www.forteds.com>
- [13] http://www.mentor.com/products/c-based_design
- [14] A. Portero et al., HW-SW design methodologies used for a MPEG video coprocessor synthesis. In *Proc. of 16th International Conference on Microelectronics*, 2004
- [15] SCATE: <http://www.sourceforge.net/projects/scate>