Semi-Formal Refinement of Heterogeneous Embedded Systems by Foreign Model Integration

Seyed Hosein Attarzadeh Niaki*, Ingo Sander*
*Electronic Systems
School of Information and Communication Technology
KTH Royal Institute of Technology, Stockholm–Sweden
Email: {shan2,ingo}@kth.se

***Abstract*—There is a need for integration of external models in high-level system design flows. We introduce a set of partial refinement operations to implement models of heterogeneous embedded systems. The models are in form of process networks where each process belongs to a single model of computation. A semi-formal design flow has been introduced based on these operations to incrementally refine system specifications to their implementation. Wrapper processes, which allow co-simulation of a system model in the framework with external models and implementations are used to keep the intermediate system models after each refinement step verifiable. Additionally, this design flow has the advantage of integrating legacy code and IP cores. Using a simple example as the case study, we have shown how we can apply this design methodology to a simple system.**

I. Introduction

Continuous grow in the complexity of embedded systems in terms of size, functionality and heterogeneity requires incorporation of appropriate disciplines in the design process. Abstraction and IP-reuse have been used to tackle this problem and make the complexity manageable. The former has appeared as Electronic System Level (ESL) modeling/design solutions and languages with higher abstraction levels which are used both for hardware and software design. Also, there exist pre-designed and verified IP cores for software, hardware and even verification tasks in different abstraction levels which can be used as a sub-component of a larger system. However, a methodology which combines the benefits of both approaches and provides integration of external models in a well-defined design flow thats starts from abstract formal models, is missing.

In a typical heterogeneous system, external IP cores may need different external simulation tools in order to verify the correct functionality of the overall system. For example, an HDL simulator is used to simulate a hardware code, while at the same time the software should be run on an instruction-set simulator. In addition, refining and synthesizing a high level system specification down to its implementation involves the same problem after each refining step. It may be needed to refine a sub-part of a big system while keeping the rest of the system at high level specification and still be able to simulate the entire system in order to study each refinement task in isolation. It is advantageous to keep the same test bench (and probably enrich it) during the design steps so that we make sure it always conforms to the original verified high level specification.

In a recent work [1], we have suggested a general method to enable co-simulation of heterogeneous systems based on the notion of Models of Computation (MoCs). Different external tools are wrapped as a process in their respective model of computation as a process in the framework in a composable way. However, a usable discipline to exploit this ability in the context of a design methodology is still missing.

In this work, we will try to fill this gap by sketching an semi-formal and incremental solution to refine a high level system specification step by step using well-defined operations. Our chosen modeling framework captures system specifications as a hierarchical network of communicating processes. The introduced operations act as a toolbox for the designer to choose how he should refine a model, while wrappers enable verification of partially refined models and integrated IP-cores by simulation. The main theme here is to replace sub-parts of a system gradually with their implementation.

In summary, the contributions of this work are:

- define a set of semi-formal refinement operations to lower the abstraction level of a system model by replacement,
- study the requirements and propose a method for wrapping the intermediate refined components or IP cores in the system,
- propose a design flow to systematically apply the introduced operations to a high-level system specification in order to get the final implementation.

II. The Modeling Framework

ForSyDe (Formal System Design) [2] is the modeling framework which is chosen for the purpose of this work. However, the ideas presented here are general enough to be applicable to other similar formal heterogeneous modeling frameworks. In this section, we briefly describe how ForSyDe addresses the problem of formal heterogeneous embedded system design.

A. Models of Computation

Models of Computation (MoCs) [3] allow us to express models with different time abstractions, which is required for practical heterogeneous system design. MoCs define how different components of a system communicate and interact with each other—taking either physical time or an abstract

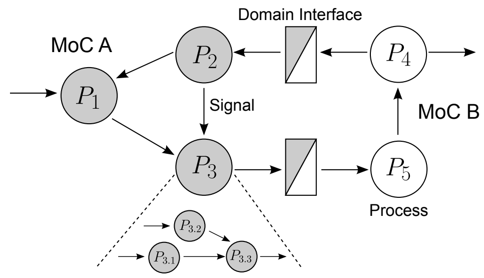


Fig. 1. A Hierarchical Heterogeneous Process Network (HPN) in ForSyDe.

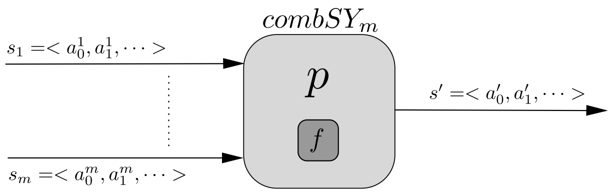

$combSY_m$
$s_1 =< a_0^1, a_1^1, \cdots >$
p
f
$s' =< a'_0, a'_1, \cdots >$
$s_m =< a_0^m, a_1^m, \cdots >$

Fig. 2. The process constructor $combSY_m$ creates a combinational synchronous process.

notion of time into account. Specific MoCs are suitable for modeling each component of a complex system or even different aspects of a single component. For example, in the synchronous MoC, which is mainly used for digital hardware modeling, the lifetime of a system is seen as instantaneous reactions (clock ticks), in which each component consumes exactly one data token from all of its inputs and produces exactly one output token on its outputs. On the other hand, to model an analog component of a system, the continuous-time (CT) MoC becomes handy which assumes a non-discrete consumption and production of tokens in time. A formal definition of some of the MoCs used in ForSyDe can be found in [4].

B. System Specification in ForSyDe

In the ForSyDe design methodology, a system is specified as a *Hierarchical (concurrent) Process Network* (HPN). Each process belongs to a single MoC. Processes communicate with each other only via *signals* and each process belongs to a specific MoC. Processes which belong to different MoCs are connected via special processes, called *Domain Interfaces* (DIs). Hierarchy is achieved by process composition. Processes in a lower level level of hierarchy must belong to the same MoC as their common parent, unless a DI is used explicitly. Fig.1 illustrates the structure of a hierarchical heterogeneous ForSyDe process network. Here, the process P_3 is made out of three processes $P_{3.1}$, $P_{3.2}$, and $P_{3.3}$.

Signals are used as a communication and synchronization mechanism between processes. In ForSyDe, we follow a similar approach to the tagged signal model [3], where a signal is an ordered set of events, and events are composed of tags (which could be implicit) and values. From the designer's point of view, signals can be thought of as streams of events conveying data tokens between processes.

A key concept in ForSyDe, which makes it different from many other modeling frameworks, is that the designer is restricted to use a set of pre-defined constructs in order to specify a system. In ForSyDe, functional processes are only created by means of *process constructors*. Process constructors are in charge of building processes out of user-defined functions and constants, which define the behavior of the resulting process. This restriction leads to a formal model which improves analyzability, since the usage of a specific process constructors implies how the process perform computation, how it communicates with other processes, and to which MoC it belongs.

Basic Process Constructors: A $combSY_m$ process constructor creates a combinational process which takes m input signals, generates one output signal and has no internal state as illustrated in Fig.2. Such a process will continuously get one value from each of its inputs and supplies them to a function to calculate the output. $combSY_m$ is a stateless process constructor, which takes the function f as argument and returns the process p. p in turn is a process with m input signals $(s_1, \cdots, s_j, \cdots, s_m)$ and one output signal (s'). The function f defines the functionality of the process. It takes one input event from signals s_j as arguments and produces one output event, so: $f(a_i^1, \cdots, a_i^m) = a'_i$. Thus, the process p is defined by f, which is repeatedly applied on parts of the input signals and produces parts of the output signal. We call an application of f an *activation cycle*, *firing cycle* or *evaluation cycle* of the process.

The $delaySY$ process constructor introduces state to a model by making processes that add a delay to the signals. It acts much like a register in digital hardware design. The produced process concatenates an initial value to the beginning of the input signal. More complex stateful process constructors—such as Moore and Mealy state machines—can be defined based on $delaySY$ and $combSY_m$ process constructors. For example, in order to create a process p that implements a Mealy finite-state machine in the synchronous model of computation, the designer uses the process constructor $mealySY$, and supplies the function f calculating the next state, the function g calculating the output, and the value v that gives the initial state of the finite state machine. Thus, a process P of type $mealySY$ can be created with

$$p = mealySY(f, g, v) \tag{1}$$

The formality and analyzability acquired by employing process constructors in ForSyDe has been used to develop a transformational refinement methodology [2], a compiler for hardware-synthesis [5], and formal methods to verify the correctness of local refinements [6].

The reference implementation of ForSyDe in the pure functional language Haskell [7] implements processes operating on signals as functions which recursively operating on (potentially infinite) lazy lists. However, the concepts used by ForSyDe are language independent. For example, the SystemC implementation of ForSyDe uses FIFOs with blocking reads

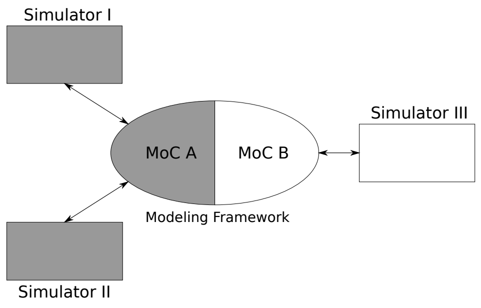


Fig. 3. Heterogeneous co-simulation using a heterogeneous modeling framework.

and writes as signals in order to implement the same behavior on top of the SystemC scheduler.

III. Wrapping External Models

The challenge is to simulate a partially refined heterogeneous system with components that are modeled and executed in different tools and languages. It is important to know how shall the simulators and the main simulation engine be synchronize together. In this section we will introduce a recent extension to ForSyDe which allows construction of wrapper processes that are used for co-simulation [1]. This formal extension not only allows heterogeneous co-simulation of a partially refined system, but also eases reasoning about the co-simulation models and makes room for automating tasks such as interface and wrapper generation.

A. The Co-simulation Architecture

In order to realize and ease the co-simulation of a main model with other models run by external tools, we let the tools communicate with the main modeling framework using their natural semantics which is best abstracted using a model of computation. In other words, the first step is to understand the MoC which best describes the level of abstraction that a specific language/modeling tool assumes. Then, a framework which can execute models with multiple MoCs will be chosen and extended to communicate with other tools and co-simulate/co-execute them together. Fig.3 represents this idea pictorially. Here, MoC A best describes the abstraction level of simulators I and II and MoC B is more suitable to capture the semantics of simulator III. They communicate with the main framework with (almost) the semantics of the MoC that they belong to, and the modeling framework handles the communication between the MoCs internally—using domain interfaces. Note that a simulator which is best described by a specific MoC (e.g., SY) can still be freely chosen from different abstraction-levels and accuracies (e.g., cycle-accurate, instruction-accurate, etc.).

Based on what has been described in Sec.II, ForSyDe is perfect candidate for such a framework. The core idea is to introduce a new class of processes to ForSyDe which "wrap" an external model inside a selected MoC. These *wrappers* are back-doors in the framework that communicate data and synchronize with external simulators, but appear as ordinary processes in the ForSyDe process network—complying with the semantics of the MoCs to which they belong. This implies that all the communication between models should be done using the ForSyDe framework and no access to global values are permitted.

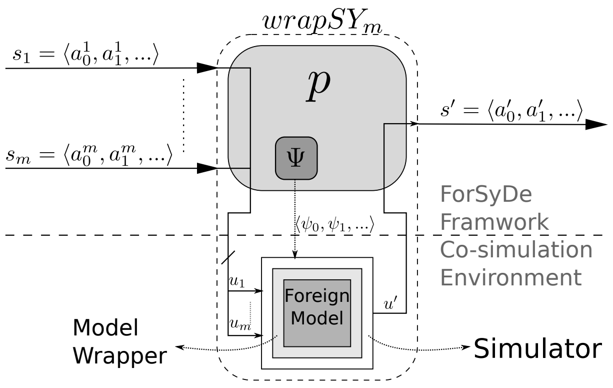

$wrapSY_m$
$s_1 = \langle a_0^1, a_1^1, ...\rangle$
$s_m = \langle a_0^m, a_1^m, ...\rangle$
p
Ψ
$s' = \langle a_0', a_1', ...\rangle$
$\langle\psi_0, \psi_1, ...\rangle$
ForSyDe Framwork
Co-simulation Environment
u_1
u_m
Foreign Model
u'
Model Wrapper
Simulator

Fig. 4. The process constructor $wrapSY_m$ creates a wrapper process with m inputs and one output in the synchronous MoC, plus the communication and synchronization streams between the simulator and the framework.

B. Communication and Synchronization

In Sec.II we have seen that ForSyDe processes have no means of synchronization or communication except the data passing through signals.

As the first option, we have exploited the same idea in order to implement the interface between ForSyDe and each simulator. A FIFO-like communication medium with blocking read and write semantics is sufficient for this purpose.

However, not all simulators can be controlled in this way. We may need to explicitly control the target simulator engine for each execution step in order to advance the simulation. We do this by issuing a set (or a stream) of commands to the simulator in addition to communicating the input and output data.

C. The wrapper process constructor

The wrapper process constructor is defined formally in [1] and is illustrated in Fig.4.:

The process constructor accepts Ψ, a sequence of simulation functions $\langle\psi_0, \psi_1, ...\rangle$ and produces a process p. $\psi_i s$ may be implemented as simulator control commands. But, they need to be semantically equivalent to function application to inputs in order to obtain the simulator outputs. The process p, in addition to its normal input and output signals, has an input and a set of output streams which are used by the process for communication with the simulator. The communication streams u_j and u' are equivalent to the partitioned input and output signals and could be implemented as TCP queues, named pipes, shared memory communication, etc. The final shape of Ψ after implementation is context dependent and can be a set of commands for controlling the simulator engine or even totally absent (in case of synchronization via blocking communication).

In the scenario shown in Fig.4, process p is called the *wrapper process*, the foreign model is called the *wrapped model* and the adaptation layer around it is called the *model wrapper*.

D. Model Wrappers

Model wrappers are adaptation layers around the actual simulated models and interact with the ForSyDe framework by exchanging data and synchronizing with it. Model wrappers depend on the target language/tool which is being simulated and can vary based on the number of input and output data streams, the simulation commands, and the communication/synchronization medium which is chosen for implementation. Fortunately, a generic model wrapper template can be introduced for each simulator and it can be even generated automatically based on the information obtained from the $wrapSY_m$-based process. In [1], three popular classes of model wrappers are introduced: a GDB wrapper for wrapping compiled software, an HDL wrapper for digital hardware models, and a Simulink wrapper for high-level models. We have restricted this work to the synchronous model of computation. We briefly review the essentials of the first two wrappers in the following.

Co-simulation of hardware and software is essential for most of the embedded system design projects. In the context of this work, we are interested in developing a generic model wrapper for compiled embedded software. The software might be running on the designer's host machine at early development stages where testing the functional correctness is desired, be simulated in an Instruction-Set Simulator (ISS), or even run on a real hardware platform and co-executed with high-level models in a Hardware-In-Loop (HIL) fashion. The GNU Debugger (GDB) [8] is the debugger provided by the GCC tool-chain which supports a wide range of processors. GDB allows the programmer to run a software, stop it at specified breakpoints, inspect its progress and even modify its environment. Many instruction-set simulators also provide GDB support, which makes it a good candidate for the purpose of this work. No matter if we are running the software on the host machine, simulating it in an ISS, or running it on actual hardware, we call the GDB instance itself the simulator. The wrapper process in ForSyDe can connect to the GDB instance via a TCP connection or a serial line, synchronize, and communicate data with it using GDB commands. The model wrapper in this case includes synchronization breakpoints at the beginning and at the end of a loop where the data is written to and read from memory locations corresponding to input and output variables, using GDB commands. Thus, in this case communication is performed by manipulating the memory locations in the software's address space. The body of the loop only invokes the wrapped software function(s).

The intention here is to wrap a model in the synchronous subset of hardware description languages. The synthesizable subset of VHDL and Verilog HDL fall into this category. The assumption here is that the model is simulated using an HDL simulator. The model wrapper appears as a top-level module which instantiated the wrapped model and communicates input and output data from/to the ForSyDe framework to/from it. A simple design of the wrapper synchronizes the external tool with the framework using the input and output streams without explicit control over the simulation engine. It can be simply implemented using file I/O and Unix named pipes without any specific tool support.. Using standard facilities in HDLs such as the `TEXTIO` package in VHDL, the model wrapper can write to and read from Unix named pipes. One more issue to be taken care of is generation of clock and reset inputs for the wrapped model. In the synchronous MoC of ForSyDe, no explicit clock exists. If the wrapped model contains registers, then the model wrapper is needed to generate these signals and feed the HDL model with them. A valid periodic clock must be generated in the model wrapper which will be used both by the core process of the wrapper and also fed to the wrapped model.

IV. Refinement-by-Replacement

Wrappers can be used to follow a bottom-up design for heterogeneous systems, where the implementation of individual modules are known and we need to compose and examine them as a co-simulation model in a heterogeneous design framework.

Alternatively, if a co-simulation model is obtained by replacing a process in a simulation model with a wrapper process, which wraps a model with a lower abstraction level compared to its original counterpart, a top-down design approach has been followed. We call this *refinement-by-replacement* and is the focus of this work. Thanks to the new extension, an iterative and partial refinement of a model down to its implementation while keeping the model executable and testable with the original test-benches in each step is possible.

In this section we will see how the refinement-by-replacement design methodology uses the wrappers to refine and implement a ForSyDe model. A set of operations are defined to refine and transform sub-parts of a model. Then a design flow will be sketched using these operations.

A. The Partial Refinement Operations

The goal is to transform an abstract model which captures the essential functionality of a system to a desired implementation of it incrementally. In each step we need to perform a partial refinement operation which lowers the abstraction level for one sub-part of a system locally, leaving the rest of the system unaffected. We will try to define these operation carefully so that we don't lose the benefits of the formality and verifiability of the original system at once. Fig.5 illustrates the evolution of (probably sub-part of) a ForSyDe model down to its implementation by successive application of refinement operations.

1) The Replacement Operation: This is the the most fundamental operation for refinement of an abstract model. A process is chosen to be replaced by a wrapper process which exchanges data and synchronizes with its implementation, or an externally simulated model of it. Such a process is called

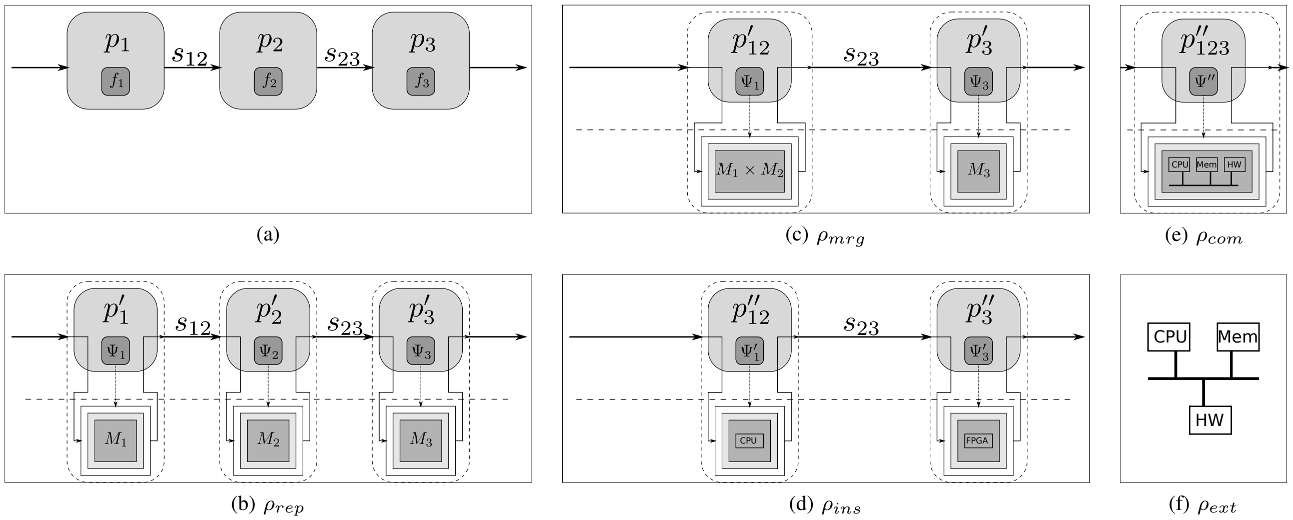


Fig. 5. Processes p_1, p_2, and p_3 in the original model (a) are replaced by their implementation in the refined model using replacement operations (b). A merge operation combines processes p'_1 and p'_2 to a single process p'_{12} (c). After application of platform instantiation operations, processes p'_{12} and p'_3 are elaborated to p''_{12} and p''_3 in (d) which now include the target implementation architectures. Usin a communication synthesis operation, processes p''_{12} and p''_3 are combined to a single process p''' (e) with proper communication established between their implementations. By applying the extraction operation, process p''' which includes the final implementation of the system, is extracted from framework and prepared for deployment (f).

a *replaced process*. Thus, this operation can be represented as a function $\rho_{rep} : P \rightarrow P$ where P is the set of all processes. The original process can be either a basic process, which is directly produced using a process constructor, or a composite process, which resides in higher levels of hierarchy and is formed by combining other processes. Fig.5b illustrates this scenario. In this case, p_1, p_2, and p_3 are replaced by processes which wrap their implementations M_1, M_2, and M_3. The communication signals are identical before and after application of the operation.

Note that in case of a ρ_{rep}, the separation between communication and computation is still preserved. The wrapped implementation only performs computation and it's the wrapper process which takes care of the communication with other processes in the respective MoC. Thus the implementation could be a single function in software, a piece of hardware without any specific protocol interface, etc.

A valid ρ_{rep}, by definition, is a one which results to a wrapped implementation and acts almost like the original process in the model. This means that both processes should have the same number of inputs/outputs and provided with the same events in their input signals, with a tolerable approximation, both processes must produce the same events in their output signals.

2) The Merge Operation: The merge operation simply combines two or more replaced processes together and it is of type $\rho_{mrg} : P^M \rightarrow P; M \geq 2$. This operation is used in situations where we need several similarly-implemented processes to share the same resource. We capture the similarity by requiring that the input replaced processes should have the same simulation functions (Ψs). For example, suppose that the two replaced processes p'_1 and p'_2 in Fig.5b which belong to the same MoC, are implemented as software functions and wrapped using the same wrapper (probably the GDB wrapper), which means: $\Psi_1 = \Psi_2$. If we know from platform constraints that both of the wrapped software functions in p'_1 and p'_2 will be executed on the same processor, it is reasonable to combine them together and wrap the result as a single process p'_{12} that integrates both functionalities ($M_1 \times M_2$) with the same kind of wrapper or simulation function (Ψ_1). Fig.5c shows the idea pictorially.

From the modeling framework point of view, merging two wrapper processes is done using similar parallel and sequential combinators ($||$ and $\circ$) which are defined in [4]. Note that just like the previous operation, the observed behavior of the resulting process, should be same as behavior of individual processes together before applying ρ_{mrg}. Care should be taken in case of using $||$, where it can introduce invalid constructs such as combinational loops in the synchronous MoC.

3) The Platform Instantiation Operation: Instantiation of the platform for a replaced process is the operation by which the information about the real architecture on which the implementation will be executed is added to the model. We call such a process an *instantiated process*. This operation has simply the type $\rho_{ins} : P \rightarrow P$. In case of a digital hardware implementation, platform instantiation is equivalent to synthesis of an RTL model to the netlist of target technology gates/blocks. In the software case, it is the act of running the compiled software on an instruction-set simulator (ISS) or a full architecture simulator instead of the host machine. Fig.5d illustrates an example of platform instantiation. Here, the software functions which implemented $M_1 \times M_2$ are compiled for a target CPU and simulated using ISS, and the RTL code that implemented the model M_3 are synthesized to the netlist of an FPGA target.

Platform instantiation can change the type of the simulator used to co-simulate the model and hence, the type of the wrapper process (or talking more abstract, the simulation functions). For example, a GDB wrapper might be used to wrap a software implementation of a process after a ρ_{rep}. But, as a result of a ρ_{ins}, an RTL model of bus based System-on-Chip is selected to execute the software which needs an HDL simulator to be executed.

4) The Communication Synthesis Operation: The merge operation is used to combine replaced processes in case they are homogeneous and will be implemented on the same resource. In such cases the implementations of the original processes can be combined and optimized to a single component in the final system. But, if the processes are heterogeneous, or they are decided to be implemented on different resources, we need to establish proper communication between them during integration. The Communication Synthesis Operation or $\rho_{com} : P^M \rightarrow P; M \geq 2$, combines two or more instantiated processes into a single process which includes the implementations of the original processes plus the appropriate communication among them. Fig.5e shows an example for communication synthesis.

Here, p''_{12} which wraps a CPU model that executes the software implementation of its functionality and P''_3 which includes a hardware implementation of an accelerator are chosen for combination using ρ_{com}. The resulting process, includes both implementations which communicate using a bus-based shared-memory protocol.

5) The Extraction Operation: After proper application of the previous operations, the designer will come up with a single wrapper process which includes a full implementation of the desired system. The Extraction Operation $\rho_{ext} : P \rightarrow \emptyset$ is simply the act of extracting the final implementation from the design framework and make it ready for a standalone deployment. This usually involves final refinement and synthesis operations, removing extra debugging facilities used for co-simulation/co-execution of the model, connecting the inputs and outputs of the implementation to proper interfaces instead of the testbenches, and performing final relevant optimizations. Fig.5f is an illustrative representation of this operation.

Definition 1: The set of all available refinement operations $\rho : P^M \rightarrow P^N$ is denoted by R and is defined as:

$$\rho \in R = \{\rho_{rep}, \rho_{mrg}, \rho_{ins}, \rho_{com}, \rho_{ext}\} \tag{2}$$

B. The Refinement-by-Replacement Design Flow

Starting from a valid ForSyDe model as a process network, a design flow consists of successive application of partial refinement operations until the final implementation is achieved. As stated before, the system can be simulated after each partial refinement in order to verify the functionality of the resulting model and/or estimating the cost of a design decision. Although there is not a strict order in which the designer should apply these operation on the processes in a system model, some of the refinement operations depend on each other. For example, a ρ_{mrg} is defined to combine two replaced processes, which means it should be preceded by two ρ_{rep}s on the candidate processes. Fig.6 summarizes the dependencies among the partial refinement operations. An arrow from operation a to operation b means that candidate process(es) for a to operate on, must have been previously refined by b.

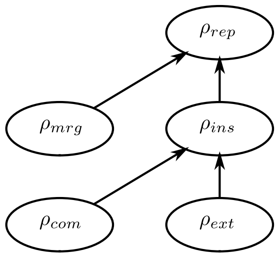

$$\begin{tikzcd} & \rho_{rep} \\ \rho_{mrg} \arrow[ur] & \rho_{ins} \arrow[u] \\ \rho_{com} \arrow[ur] & \rho_{ext} \arrow[u] \end{tikzcd}$$

Fig. 6. The dependencies among partial refinement operations.

```
Require: A valid process network $PN$
 1: repeat
 2:     choose $ps = \{p_1, \ldots, p_M\}; p_i \in PN$ and a $\rho : P^M \rightarrow P^N$ so that $ps$ is $\rho$-ready
 3:     replace $ps$ with $ps' = \rho(ps)$ in $PN$
 4:     validate $PN$ by simulation
 5:     while (constraints violated) or ($PN$ invalid) do
 6:         choose $ps'' = \{p'_1, \ldots, p'_{M'}\}; p'_i \in PN$ and a $\varrho : P^{M'} \rightarrow P^{N'}$ so that $ps'$ is $\varrho$-ready
 7:         replace $ps''$ with $ps''' = \varrho(ps'')$ in $PN$
 8:     end while
 9: until $|PN| = 1$ and $p \in PN$ is $\rho_{ext}$-ready
10: apply $\rho_{ext}(p \in PN)$ and exit
```

Fig. 7. A design flow using the defined refinement operations.

Definition 2: Consider a process network PN, a set of processes $ps = \{p_1, \ldots, p_M\}; p_i \in PN$, and a refinement operation $\rho : P^M \rightarrow P^N \in R$. ps is said to be ρ-ready if all elements of ps satisfy the dependency relations for application of refinement operation ρ to ps.

After application of one or more refinement operations, it is possible that the designer observe a design constraint violation or would like to check other refinement possibilities. Using a *rollback* it is possible to cancel a refinement operations, yielding a more abstract model.

Definition 3: Having $\rho : P^M \rightarrow P^N$ as a refinement operation and $p \in P$ a ForSyDe process, a rollback $\varrho : P^N \rightarrow P^M$ is defined to be the reverse of a refinement operation as:

$$\text{if } \rho(p) = p' \text{then } \varrho(p') = p \tag{3}$$

Using rollbacks, we can go back in history of the design flow and hence, have a form of traceability.

By successive application of refinement operations and rollbacks, it is also possible to explore the design space. Fig.7 is the sketch of a possible design flow using the refinement operations defined above. In the body of a loop, first a refinement operation and a set of candidate processes are identified. Then, the refinement operation is applied and the process network is updated. The new model is verified using new wrapper processes to see if any violations has happened. If the results of the refinement operation is not satisfactory, one or more rollback operations are performed to go back to an original model. At the end, if only one process is remaining and it is possible to extract it from the framework, which means it is an instantiated process, the models is extracted and deployed.

V. Case Study

We have chosen the equalizer example from [2] as a small, representative example of an embedded system in order to walk through the steps of the proposed design flow.

The main task of the equalizer system is to adjust an audio signal according to the user input. In addition, the bass level must not exceed a pre-defined threshold to avoid damage to the speakers. The `ButtonControl` subsystem monitors the button inputs and the override signal from the subsystem `DistortionControl` and adjusts the current bass and treble levels. This information is passed to the subsystem `AudioFilter` , which receives the audio input, filters and amplifies the audio signal according to the current bass and treble levels. This signal, the output signal of the equalizer, is analyzed by the `AudioAnalyzer` subsystem, which determines, whether the bass exceeds a pre-defined threshold. The result of this analysis is passed to the subsystem `DistortionControl`, which decides, if a minor or major violation is encountered and issues the necessary commands to the `ButtonControl` subsystem. Fig.8a illustrates the top level of the equalizer system.

A high-level model of this system is available as a hierarchical process network which is executable and verifiable using a test-bench. First,the four top-level processes are replaced with their wrapped implementations. The `AudioFilter` process will be implemented in hardware, while other three processes will be mapped as software to an embedded processor. The replaced implementation of `AudioFilter` internally uses three instances of the Altera FIR compiler IP core, plus multiplier and adder blocks from the library. In order to realize the DFT which is used internally by the `AudioAnalyzer` process, an open-source software IP-block is used. All other software processes are manually generated as C-code which can also be compiled on the host machine. The above four processes are wrapped using HDL and GDB wrappers and co-simulated after replacement operations in order to make sure the replaced blocks correctly implement the original model. Since the hardware blocks use an 8-bit representation instead of the floating point representation of floating-point numbers, some mismatches will occur. A revision of the test-bench which tolerates this variation from the original specification makes sure the implementation is still correct.

In the next step, because `AudioAnalyzer`, `ButtonControl` and `DistortionControl` are going to be implemented on the same CPU, a merge operation combines them all into a single process (Fig.8b).

By applying the platform instantiation operation to the two remaining processes, the target architecture details are pinned down. The wrapped software is compiled for an Altera Nios II soft processor which can be simulated in the Nios II ISS and still wrapped using the GDB wrapper. As for the `AudioFilter` process, performing synthesis and fitting for the Cyclone II FPGA followed by running the EDA netlist writer in Quartus II yields a simulation-ready netlist of final logic blocks. Again, simulation of the complete system with the same wrappers examines the validity of the implementation against the original model so far.

For communication synthesis, we introduce the interface between the software and hardware processes by adding input and output FIFOs to the hardware block which bridge between the Avalon memory-mapped interface of the CPU and the Avalon streaming interface of the hardware block.

The final extraction operation consists of re-compiling the software and re-synthesizing hardware with higher optimization efforts, removing the Altera JTAG debug core from the system, and connecting the inputs and outputs of the system to the audio A/D and D/A converters of the Altera DE2 board.

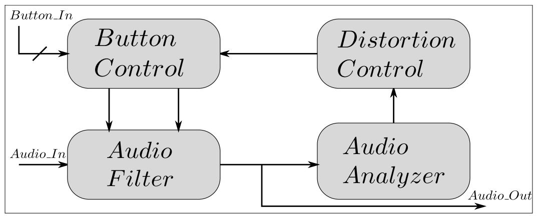
VI. Related Work

There exist other modeling frameworks that support heterogeneous system modeling based on the concept of models of computation. A notable example is Ptolemy [9] which supports a rich set of MoCs, but takes a different, actor-based implementation approach. Unlike ForSyDe, heterogeneity in Ptolemy is restricted to hierarchical composition in favor of a more disciplined approach and actors in the same level of hierarchy should belong to the same MoC. This restriction sometimes become annoying, especially for top-level models where heterogeneous components need to be composed. HetSC [10] is based on System-C and implements some MoCs on top of the SystemC discrete-event simulation engine. In contrast to ForSyDe, HetSC does not claim a solid formal basis.

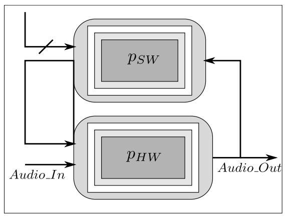
A considerable amount of work has been dedicated to hardware-software co-simulation. In [11, Sec.3] a survey of the well-known techniques is given. There, the main attempt is to boost the co-simulation speed via ad-hoc methods rather than providing a general solution for heterogeneous modeling.

Transaction-Level Modeling (TLM) is also a popular approach for system-level modeling and simulation nowadays. TLM is not a well defined, single abstraction level. Instead it is usually defined as abstraction levels above RTL. Depending on the amount of timing details in the model, TLM is divided into Programmer's View (PV), Programmer's View-Timed (PVT), etc. But, models in none of these abstraction levels have clearly defined semantics. Therefore, the main usage of TLM is restricted only to achieving faster simulation speeds for prototyping.

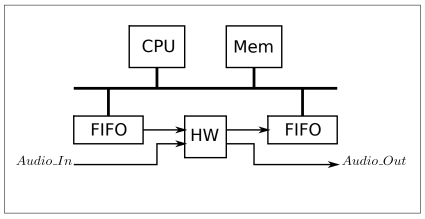
Commercial tools such as MATLAB/Simulink support integration with external tools and even implemented models using features such as Processor-in-Loop or Software-in-Loop. The drawback is that although Simulink is used in many application areas, it is not by itself a heterogeneous modeling environment. Also, we were unable to find a formal description of the semantics of Simulink models.



(a)



(b)



(c)

Fig. 8. The top-level block-diagram of the equalizer example (a). After replacement and merging software processes (b). Final implementation after extraction (c).

Some design frameworks such as Metropolis [12] try to formalize the design flow of typical heterogeneous system, but they typically avoid to deal directly with the fact that several IP blocks with different formal basis exist and need to be integrated into the design flow.

VII. Conclusion

Based on the idea of wrappers, which enables the ForSyDe modeling framework to integrate external models and legacy codes for simulation, we have presented a semi-formal top-down approach for refinement of ForSyDe models to their implementations. We have defined a set of partial refinement operations which are used in the design flow to incrementally map a process network to a target architecture. In addition to producing co-simulatable models, the introduced refinement operations act locally, which means that they do not affect the rest of model. This property has several interesting benefits. It helps the designers to lower the abstraction level of a sub-part of the system, while keeping the rest of model in high level, leading to faster verifiable-by-simulation models. Alternatively, it is possible to use the same test-benches to verify the functionality of a system description after each partial refinement during the design flow, and keep it compliant with the original specification.

Using the example of an equalizer system, we have shown how the presented flow can be used in practice to implement a systems for which we have the implementation of sub-modules already as IP-Blocks.

The proposed design method is not and can not be fully formal since it deals with an intrinsically informal problem. We have tried to avoid over-formalization of this work in order not to keep it general enough.

As a result, the workflow presented in Fig.7 does not suggest which refinement and rollback operations to be used in each step in a clever way. Instead, some sort of a full search is actually being performed. This partly due to the fact that a successful design decision typically requires the human guidance. But, it could be improved in the future by combining it with more elaborate design-space exploration techniques and also provide a better tool support for it.

Acknowledgment

This work is partially performed in the framework of the European ARTEMIS supported project SYSMODEL [13].

References

[1] S. H. Attarzadeh Niaki and I. Sander, "Co-simulation of embedded systems in a heterogeneous moc-based modeling framework," in *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, june 2011, pp. 238 –247.

[2] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, January 2004.

[3] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, December 1998.

[4] A. Jantsch, *Modeling Embedded Systems and SoCs*. Morgan Kaufmann, 2004.

[5] I. Sander, A. Acosta, and A. Jantsch, "Hardware design and synthesis in ForSyDe," Workshop on Hardware Design using Functional languages (HFL 09), 2009. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.151.8697

[6] T. Raudvere, I. Sander, and A. Jantsch, "Application and verification of local nonsemantic-preserving transformations in system design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 6, pp. 1091 –1103, june 2008.

[7] The Haskell programming language home page. [Online]. Available: http://www.haskell.org

[8] GDB: The GNU project debugger home page. [Online]. Available: http://www.gnu.org/software/gdb/

[9] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[10] F. Herrera and E. Villar, "A framework for heterogeneous specification and design of electronic embedded systems in SystemC," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–31, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1255456.1255459

[11] M. Loghi, T. Margaria, G. Pravadelli, and B. Steffen, "Dynamic and formal verification of embedded systems: A comparative survey," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 585–611, 2005. [Online]. Available: http://www.springerlink.com/content/f6gt2010185n6260/

[12] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, pp. 45–52, 2003.

[13] The SYSMODEL project home page. [Online]. Available: http://www.sysmodel.eu