Pre-Simulation Elaboration of Heterogeneous Systems: The SystemC Multi-Disciplinary Virtual Prototyping Approach

Cédric Ben Aoun*, Liliana Andrade*, Torsten Maehne*, François Pêcheux*, Marie-Minerve Louërat*, Alain Vachoux†
* Sorbonne Universités, UPMC Univ Paris 06, CNRS UMR 7606, Paris, France
† École Polytechnique Fédérale de Lausanne (EPFL), LSM, Lausanne, Switzerland

***Abstract*—Designers of the upcoming digital-centric More-than-Moore systems are lacking a common design and simulation environment able to efficiently manage all the multi-disciplinary aspects of its components of various nature that closely interact with each other. A key to successful design and verification lies in a SystemC-based virtual prototyping environment that is able to simulate a complex heterogeneous system as a whole, for which each component is described and solved using the most appropriate Model of Computation (MoC).**

In this paper, we present a new generic MoC-independent elaboration scheme that aims at preparing a Virtual Prototype (VP) for simulation. It requires to check the correct composition of the system model through dimensional analysis, to explore the model structure to identify involved MoC and interfaces between MoCs, and to detect the underlying dependencies. Eventually, information extracted from the exploration allow the instantiation of MoC-specific solvers. To soundly handle the global model execution with a Discrete Event (DE) kernel as the main solver, synchronization mechanisms with master-slave semantics within the model structure are implicitly deduced.

I. Introduction

Design engineers, who want to build tomorrow's More-than-Moore embedded systems, must learn to think, create, and design differently than today. The embedded software plays an important role in these digital-centric systems and must be validated long before the physical circuit is available. In this context, virtual prototyping at a high level of abstraction as a mean to experiment innovative ideas, perform design exploration and early verification of both functional and non-functional aspects becomes a necessity. This is especially true for systems featuring a tight coordination between the computational and physical elements of different natures. Such complex heterogeneous systems, which mix digital and analog electronics as well as mechanical, thermal, optical, RF domains, and software, can not anymore be built as the late and hopefully right composition of intellectual properties.

As a consequence, Electronic Design Automation (EDA) tooling must accordingly evolve with these new requirements and even anticipate them. The challenges for simulation tools are especially numerous, complex to solve, and intertwined. These tools have to remain simple, robust, and accurate while offering a good simulation performance and attracting designers with various backgrounds and proficiency levels. Ultimately, the Holy Grail of heterogeneous simulation is a monolithic, but extensible environment that offers to any System on a Chip (SoC) designer an efficient way to simply connect and efficiently simulate models coming from heterogeneous places. They should not have the need for fiddling with simulation artefacts or the Models of Computation (MoCs) intrinsic parameters used to describe the different models. For these designers, whose goal is mainly the definition of an executable specification of the system to be implemented and who are at the very heart of the innovation process, simulation accuracy is less important than simulation speed and holistic consideration of all the system specifications. In other words, first-order results are sufficient, provided the embedded software can soundly run on the simulated system involving different physical domains and experience the environment's reactions with the time constants proper to these domains.

In essence, the simulation engine for such multi-disciplinary virtual prototyping must exhibit the following properties. It has to be digital-centric, because digital hardware and software are at the heart of today's and future heterogeneous systems. Its kernel has to be monolithic, for the sake of performance. It has to support the simulation of billions of digital cycles, the booting of a complete operating system, the transmission of wireless messages between RF transceivers, and manage non-functional properties at different time scales. For this reason, the overhead associated to the use of external tools and related synchronization issues with approaches like Functional Mockup Interface (FMI) [7] are often too high. The simulation engine has to deal with complex hierarchical models that reflect the system design and it has to check the system's correct composition of its potentially multi-disciplinary entities and the homogeneity of the resulting model equations. Furthermore, the complexity of the model equations and the variety of concerned physical domains require to put dimensional analysis at the center of the modeling process. Dimension checks have to lead to *meaningful* messages, which help locating compilation, compositional, and causality errors. The simulation engine also has to be flexible and ready for extensions, i.e., it should be easy to add new MoCs for specific application domains that integrate seamlessly with all

This work is supported by the European project CATRENE CA701 H-INCEPTION.

978-1-4673-7311-1/15/$31.00 ©2015 IEEE

other MoCs. This requires the definition of an Application Programming Interface (API) that allows the addition of new MoC without too much difficulty. It is based on a generic recursive elaboration mechanism that helps defining an explicit synchronization scheme between compatible MoCs. The simulation engine should also support the verification and validation methodologies, such as the Universal Verification Methodology (UVM) [1], that are more and more tightly integrated into the design process.

Inspired by the work of the Ptolemy II [6] and based on the proven SystemC [11] with its Analog/Mixed-Signal (AMS) extensions [2], this paper presents the steps taken to realize the pre-simulation elaboration, i.e., to build up the appropriate C++ structures that extend the SystemC simulation kernel to deal with the simulation of heterogeneous models.

The key idea of the presented elaboration scheme is to take as a starting point the system model (composed of a hierarchy of sub-models interconnected by ports and signals) and run through it to identify a finite set of homogeneous regions, each of which being handled by a specific model of computation. During this search, performed both in depth and breadth, the interfaces between these regions (identified by special ports called converter ports) are determined, and a hierarchy of interoperating MoCs is built up. This hierarchy is used to express unequivocally the interactions between the different MoCs and the associated solvers.

After a related work section that details the work of Ptolemy II, SystemC extensions, and the existence of powerful C++ libraries such as Boost Units, the main elaboration phases of the framework *SystemC Multi-Disciplinary Virtual Prototyping* (SystemC MDVP) are presented: dimensional analysis, MoC-oriented clustering, and recursive MoC-specific solver instantiation. An outlook on the simulation phase concludes the paper.

II. Related Work

One of the pioneering works in the field of heterogeneous systems simulation is the Ptolemy Classic [5] and its sequel, Ptolemy II [6]. Ptolemy II is a proof of concept simulator, which addresses the complex issue of modeling heterogeneity in a hierarchy of connected entities. Ptolemy II divides a complex model into a tree of nested sub-models. Each hierarchical level is locally homogeneous. Ptolemy II mainly relies on the concept of *Actor* to describe a computational component of the system. A *composite actor* can be defined as a netlist of sub-actors. Each local sub-model is managed by a specific and well-defined *Model of Computation*, which actually defines how computation and model solving are performed. MoCs are implemented by means of *Domains*: *Receivers*, which define the communication semantics, and *Directors*, which define the execution order of actors; together, they define the environment of actors. Directors are eventually responsible for the instantiation of domain-specific receivers. Since the whole environment is defined by the domain, actors represent abstract functionalities that are reusable in many domains. However, the implications of this flexibility can be tricky or overwhelming for the end user. In addition to building the netlist of components, the designer wishes to simulate, he also has to explicitly build the composite actors to encapsulate the subsystems, and he has to choose and instantiate the correct directors with respect to the created hierarchy and the simulated domains. These constraints raise two major issues. First, it forces the user to fully apprehend the director's internals and the underlying simulator semantics. Second, the explicit definition of the hierarchy results in a composite set of system models and simulation-specific artefacts.

Besides Ptolemy, several solutions for the simulation of heterogeneous systems have been presented during the last decade. They mainly rely on SystemC [11], a Discrete Event (DE) simulation kernel, which can be used to perform rapid system prototyping at several levels of abstraction. Based on SystemC, we can notice the important work of SystemC-H [14] and SystemC-A [4]. Both extend the SystemC kernel by modifying its internal structure, which limits portability and standard compliance. Another approach built upon SystemC is *HetSC* [10], which major drawback is to not being able to express continuous time. SystemC AMS extensions [2] have been specifically developed to allow the simulation of analog behaviors coupled with digital-centric systems. These extensions have originally been implemented in the Fraunhofer SystemC-AMS proof-of-concept simulator [8]. They have been successfully applied in communication, automotive, and consumer electronics use cases with good simulation performance and accuracy.

For the moment, though, it is rather difficult for design teams to extend the current SystemC-AMS simulator with other MoCs than the ones proposed. Neither does the SystemC AMS 2.0 standard [2] define an API for this purpose nor does the proof-of-concept simulator document its internal API. To our knowledge, only two attempts have been published, which add a Non-Linear Network (NLN) [16] and Bond Graph (BG) MoC [13], respectively, by authors with deep knowledge of the SystemC-AMS implementation. These MoCs rely on internal APIs to integrate themselves without modifying the latter. An analysis of the SystemC-AMS source code shows that each MoC is required to fully handle its elaboration once the SystemC port binding phase has been finished. SystemC-AMS only provides a minimum support for this task by providing a list of all instantiated modules belonging to a certain MoC. It provides one synchronization mechanism used by all the existing MoCs and no API is provided to define new synchronization schemes between MoCs.

At last, the modeling of heterogeneous systems can be very error prone from a physical perspective, since designers from different disciplines use different measurement units and scales. A major improvement towards heterogeneous simulation was made with the integration of dimensional analysis into SystemC AMS [12] through the use of Boost.Units library [15]. It allows to enhance models with the notion of physical quantities, which avoids compositional errors.

III. SystemC MDVP

Already today, Systems on Chip (SoCs) represent a composition of different models of computation. The Figure 1 describes a generic SoC according to the present models of computation that are used to describe and simulate the heterogeneous components. One can see that the SoC embeds digital components (processor, memory, peripherals, etc.) as well as components from other engineering domains: RF transceivers, Micro-Electro-Mechanical Systems (MEMS), optical and biological sensor. This figure shows our vision of embedded systems, which sets the digital part at the center of the system, with other domains gravitating around it. It clearly illustrates the hierarchical organization of the MoCs: the optical sensor modeled by means of an optical MoC is itself encapsulated in an analog model, that will in turn behave like a digital component. To perform the holistic simulation from the digital view point, the analog part must be simulated with an analog solver but controlled, in terms of synchronization and time-management, by the digital discrete event kernel. Similarly, the optical part must be simulated with an optical solver but this solver must execute under the supervision of the analog MoC. This transitive interfacing scheme prevails in the whole system and clearly explicits a MoC hierarchy with father-son or master-slave relationships.

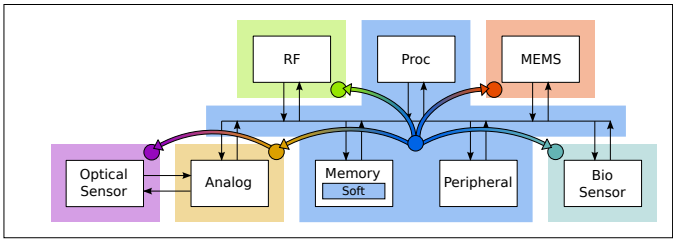

Figure 1. System on Chip as a set of interacting MoCs.

The key point of the SystemC MDVP approach is to define all the principles, tools, and API functions that allow a model sub-tree to behave as if it were a standard module of the MoC it is instantiated in. Interfacing two MoCs means performing time synchronization between the two solvers and translating signal values from one MoC to another. The time synchronization issues are part of the simulation and are addressed at the end of the paper. When it comes to model parts that are at the boundary of two MoCs, the designer should not be faced with questions that are deeply linked to the simulation infrastructure. Designers want to connect parts or third party models coming from different sources and not address issues such as the choice of the appropriate director or the setting of an obscure simulation parameter. For the simplicity of use, the translation of data values implies the use of converter ports. For him, the modeling process must appear as if one part of the port is in one MoC and the other one is in the other MoC.

When it comes to heterogeneous modeling, from the simulator architect view point, the interaction between two models of computation (heterogeneous entities) becomes more challenging. To make the SystemC MDVP framework flexible and extendible, we choose to only consider MoC interactions by master-slave semantics: in this relationship, a MoC commands and the other obeys. These semantics allow a simple definition of the interaction between MoCs: the master always imposes its view point on the slave. In the aim to perform a seamless interaction between different models of computation, it is required by the SystemC MDVP framework that the slave MoC provides all the interaction mechanisms with the master MoC, i.e., the slave adapts itself to comply with the master semantics. Whatever the complexity of the sub-models hierarchy, it has to appear as a single model from the master view point, as shown in Figure 2.

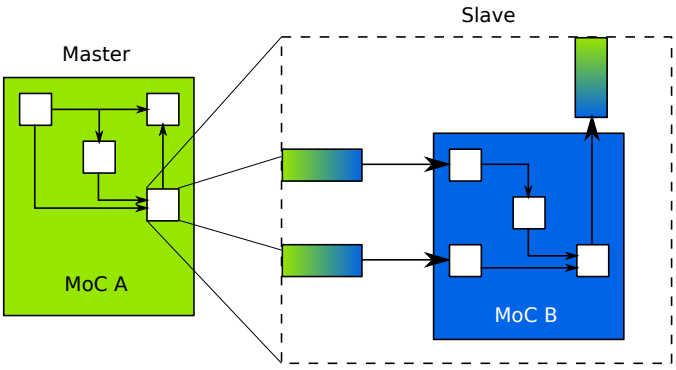

Figure 2. Interaction between two MoCs.

Each MoC has its own properties (abstraction of time, etc.), which are part of the MoC semantics. From a programming view point, this means that a slave MoC has to implement the complete set of properties/callbacks defined in a master MoC interface.

Being a master or a slave is not an exclusive state: a MoC can simultaneously be the master of another MoCs and the slave of a third MoC. While this approach may appear very restrictive (a MoC can be managed by only one master MoC), it allows to implicitly choose during the elaboration the right set of interfacing mechanisms. That is in contrast to Ptolemy II, where the designer has to explicitly add the right interfacing mechanism himself (Directors). Another benefit, compared to Ptolemy II, comes from the fact that it avoids the explicit description of the simulation hierarchy concurrently with the model, cancelling error-prone intermingling of simulation artefacts with the model description.

Those interactions must be defined within the MoC. Thus, the available interactions are statically defined and the model of computation will not be allowed to interact with a MoC that is not part of the available interaction. We chose this approach, because it allows a model of computation to be agnostic about the existence of slave MoCs.

Inside a set of connected modules, a given MoC can interact with only one master. It implies that in separated sets of connected modules, a given model of computation can interact with one master in one set and with another master in another set.

The Figure 3 illustrates the authorized master-slave relationships between MoCs within SystemC MDVP. These

interactions are valid for a set of connected modules. We see that a master model of computation can simultaneously interact with several slave models of computation since it is not aware of their existence. It is important to notice that a model of computation can not be its own master or slave, no matter if other MoCs are in-between.

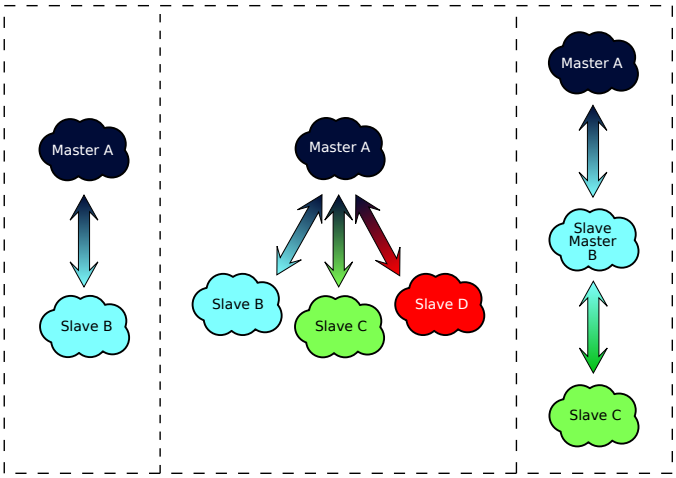

Figure 3. Authorized master-slave relations within the SystemC MDVP framework.

The master-slave semantic that we defined within SystemC MDVP to express the interaction between models of computation, is a strong concept in our approach to the heterogeneous system simulation. Stating that the slave model of computation has to comply with its master interface (seamlessly from the master view) allows to see these relations-interactions as an encapsulation. This approach is especially well-suited in a hierarchical environment.

IV. SystemC MDVP Elaboration

Any system model is built from primitive modules (which describe an atomic behavior) and instantiated modules, which are interconnected together and prepared for simulation during the elaboration phase. SystemC MDVP framework is built on top of SystemC and, as such, can benefit from the regular elaboration phase of the DE simulator kernel. To realize the elaboration with multiple models of computation, we identify three specific steps that need to be added to the standard SystemC elaboration. These steps are detailed in the Figure 4.

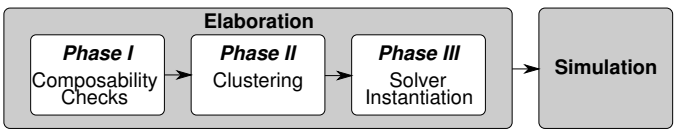

Figure 4. SystemC MDVP additional elaboration phases.

Strictly speaking, the *phase I* is not performed during the elaboration, as it is already realized during the compilation. However, the composability check perfectly fits the intent of the elaboration phase. This is why we decided to show this step as part of the elaboration process. Both, *phase II and III* are realized during the *end_of_elaboration()* callback of SystemC, which guarantees that the flattening of the system is complete (each primitive module has been identified and directly or indirectly linked to a MoC). The *phase II* describes the clustering step, where the system is being explored and analyzed to extract relevant information to perform the simulation. The *phase III* exploits the results of the previous phase to setup all the solvers required to simulate the system.

A. Composability and Dimensional Analysis

The composability of a heterogeneous model has to be checked as early as possible in the elaboration process so that later phases can rely on a sane structure. Thus, the user can be early informed about problems such as: connection of module ports to the wrong channel, e.g., continuous-time signals representing physical quantities are usually defined as values of type `double`, which make them too generic and can allow incorrect bindings of physical quantities belonging to different domains, e.g., optical and electrical domains. Other problems are: forwarding of parameter values to the wrong parameter (due to the absence of named arguments in C++), and inhomogeneous model equations. Most of these problems can be detected by the compiler through static type checking if the variables, ports, and channel types are more precisely defined in the model sources so that they express the data semantics associated to them.

The sound solution for integrating the therefore necessary quantity and unit types as well as dimensional analysis into SystemC AMS, which was presented in [12] can be also applied to our SystemC MDVP framework as it relies on the open Boost.Units library [15]. It is based on C++'s flexible type System, which allows to implement quantity and unit data types as template classes that support dimensional analysis at compile time through template meta-programming. The SystemC MDVP framework will ultimately provide full support for using Boost.Units quantity data types in user models and support the tracing of all kinds of signals. Authors of MoCs for describing multi-disciplinary behavior are encouraged to provide generic modeling primitives compatible with the new types. The quantity data types prevent the interconnection of ports and channels of different nature and ensure coherent model equations through compile-time dimensional analysis. The hard-to-read compiler error messages due to the complex template structure of the quantity data types can be efficiently simplified through a filter program.

B. Recursive Clustering

The clustering algorithm represents the second stage of the elaboration process (Figure 4, phase II). Its purpose is to explore the complete system to collect information and generate appropriate simulation data structures by means of the creation of a domain-based, hierarchical view of the system. In practice, the clustering phase consists of analyzing the system top-level model and its associated sub-models to identify a finite set of a homogeneous regions in the design, called *clusters*. Clustering aims at organizing these clusters in a cluster hierarchy and to identify for each cluster the specific MoC it is associated to and

its master. This way, the dependencies between the underlying MoCs are clearly established, based on the previously detailed master-slave semantics.

Clusters are built w.r.t the connectivity of the system provided by the designer. The notion of cluster encapsulates into a single object/abstraction entity components associated to the same MoC, which are actually connected together or connected through slave components. Eventually, a cluster can also encapsulate clusters belonging to other MoCs, provided all the master-slave semantics are respected within the whole hierarchy. This encapsulated cluster, which represents the abstraction of a slave MoC, is seen as a component of the master MoC, and is hence encapsulated into the master's cluster the same way as primitive modules. A cluster is complete when no more components (modules or sub-clusters) can be added to it by connection or instantiation. It means that all the connected modules and slave modules of the MoC associated to this cluster have been analyzed and organized to represent the subtree of the current cluster.

Applying the clustering algorithm w.r.t. the master-slave semantics allows the creation of a tree of clusters, where each cluster may only have one master (only one immediate higher node in the tree). On the opposite, a cluster can be the master of several sub-clusters (a node can have several immediate lower nodes in the tree). As SystemC MDVP is built upon the discrete event simulation kernel of SystemC, the root of this cluster tree is always a cluster associated to the DE MoC.

The clustering mechanism is very similar to the Shift-Reduce process in LR grammars [3]. During the exploration of the system's hierarchy, as the module (resp. the sub-cluster) encountered belongs (resp. integrates itself) to the MoC in the process of abstraction, a shift operation is performed. When the cluster making is complete, a reduce operation is accomplished to integrate the subtree as if it were a standard module of the higher hierarchical level.

To facilitate the generation of this tree, all SystemC MDVP modules derive from a global C++ class **scm_moc_if** (scm stands for SystemC MDVP), and a list of all the primitive modules instantiated in the system is built. Based on the content of this list, a set of primitive clusters is created with each primitive module being encapsulated into its own cluster. This set of primitive clusters represents the initial set of clusters to be analyzed (hereafter referred as set_cl). Two other structures are needed for port attributes, one indicating the *visited* status of a port and the other defining its type (regular or converter port). Algorithm 1 details the recursive clustering function, which creates the cluster tree that represents the hierarchical view of the system.

The `contains_cluster()` function (Algorithm 1, line 9) is used to break the recursion, as it avoids to process again clusters that have already been processed. If the analyzed cluster is eligible for processing, it is removed from the set of clusters (Algorithm 1, line 12). Then, each port of the currently processed cluster is considered as a starting port to find the boundaries of the cluster being built. Each processed port is considered visited, to avoid endless loops (Algorithm 1, lines 15-18). If the currently processed port happens to be a converter port (Algorithm 1, line 19), the master cluster, to which the port belongs to, is compared to the current master of the cluster new_cl (Algorithm 1, line 21). A failing comparison means that the cluster new_cl has two different master MoCs, implying a clustering error as the master-slave semantic rules are not respected. This situation leads to abort the elaboration, providing to the end user relevant information about the malformed design he tries to simulate. Otherwise,

```
1  structure Cluster
2      String moc;
3      String master_moc;
4      List<Module> moc_ifs;
5      List<Cluster> cls;
6      Solver s;
7  end
8  Function process_cluster(set_cl, cl, new_cl)
       Data: set_cl, Set of Cluster to be analyzed.
       Data: cl, Cluster to be analyzed.
       Data: new_cl, Cluster to be built.
9      if not set_cl.contains_cluster(cl) then
10         return;
11     else
12         set_cl.remove_cluster(cl);
13     end if
14     foreach port p in the port list of cl do
15         if is_visited(p) then
16             continue;
17         end if
18         set_visited(p);
19         if is_converter_port(p) then
20             master = get_master(p);
21             check_master(new_cl, master);
22             set_master(new_cl, master);
23             new_cl.add_port(p);
24             continue;
25         end if
26         foreach port p_p connected to p do
27             if is_visited(p_p) then
28                 continue;
29             end if
30             set_visited(p_p);
31             next_cl =
               get_cluster_from_port(p_p);
32             if is_converter_port(p_p) then
33                 sub_cl =
                   create_cluster(next_cl);
34                 set_cl.add_cluster(sub_cl);
35                 reset_attributes(p_p);
36             else
37                 process_cluster(next_cl,
                   new_cl);
38                 continue;
39             end if
40         end foreach
41     end foreach
42     new_cl.add_sub_cluster(cl);
43 end
```

Algorithm 1: Recursive function responsible for the clustering.

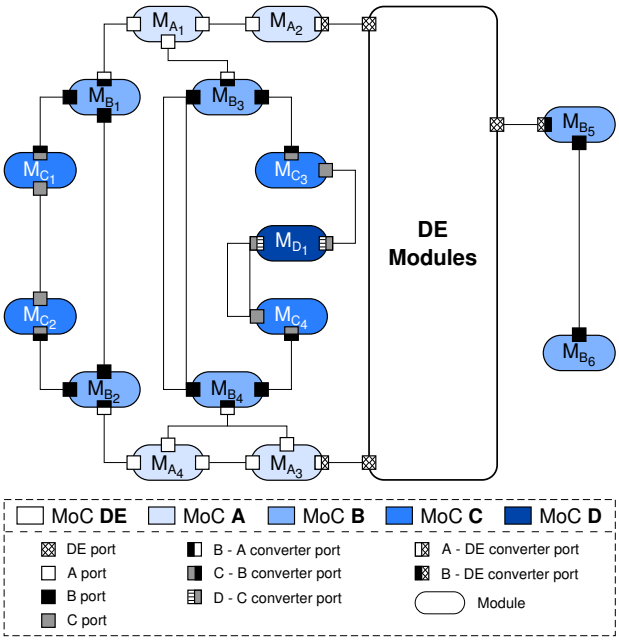

Figure 5. Example of a complex heterogeneous system involving five hierarchical levels.

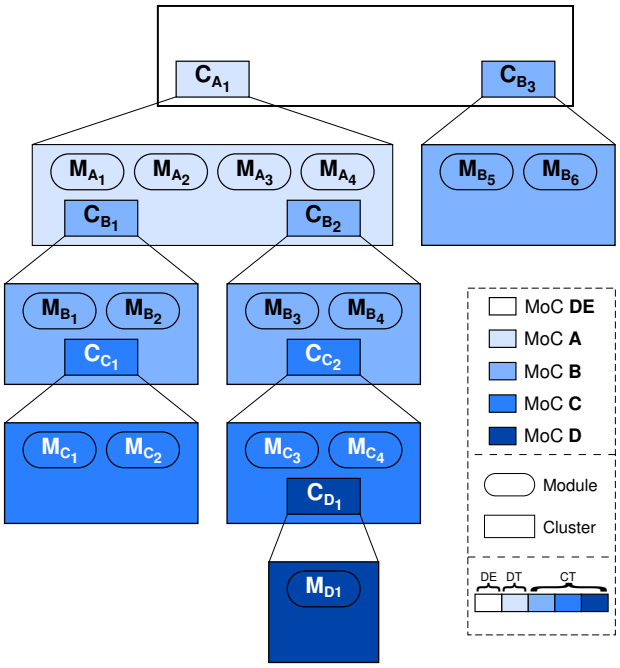

Figure 6. Clustering tree matching the complex heterogeneous system example in Figure 5.

the valid converter port is added to the cluster new_cl.

If the port is not a converter port, a depth first traversal is performed, i.e., all the ports connected to p are analyzed in turn. One can notice that the clustering algorithm does not involve the communication channel: SystemC MDVP ports provide mechanisms to directly access to the set of ports connected to the same channel. It allows to completely bypass them in the algorithm.

The process performed on each connected port is similar to the previous one (Algorithm 1, lines 27-30). This time, the traversal is used to identify the boundaries (converter port) of another slave MoC. For that purpose, a new empty cluster is created and the `process_cluster()` function is applied on it. When the built sub-cluster is complete (Algorithm 1, line 33), it has to be added into set_cl (Algorithm 1, line 34) to be processed later. To be encapsulated correctly, it must be considered as a component of the current MoC. The purpose of the `reset_attributes()` function (Algorithm 1, line 35) is to reset the ports traversal attributes to false and to re-trigger the analysis of the converter port, which has to be considered this time as a regular port of the master MoC. If the port does not belong to a boundary, a recursive call to analyze the cluster corresponding to this port is done. When the exploration of a cluster is complete, it is added as a sub-cluster to the new cluster being created (Algorithm 1, line 42).

As an example, Figure 5 describes an abstract heterogeneous system involving five different MoCs: **A**, **B**, **C**, **D**, and **DE**. The master-slave relationships between them are defined as follows: DE_{DE}>B_{CT} and DE_{DE}>A_{DT}>B_{CT}>C_{CT}>D_{CT} (with master>slave). The MoC **B** provides communication mechanisms with MoC **A** and MoC **DE** through its converter ports, so respectively do MoC **C** with MoC **B** and MoC **D** with MoC **C**.

Applying the clustering algorithm to Figure 5 yields Figure 6. The root of the tree represents the encapsulation into SystemC (DE). In each cluster, one can notice a list of primitive modules in the first row and a list of sub-clusters on the second row. Cluster $\mathbf{C_{B_1}}$ is composed of two primitive modules ($\mathbf{M_{B_1}}$ and $\mathbf{M_{B_2}}$ associated to MoC **B**) and one sub-cluster $\mathbf{C_{C_1}}$ associated to the MoC **C**. From the MoC **A** view point, the sub-cluster $\mathbf{C_{B_1}}$ must appear as a MoC-**A**-compliant module, i.e., it has to respect the semantics of MoC **A**.

This representation imposes to each level to respect the semantics of the immediate parent level. Finally, this leads to a simulatable system ruled by SystemC semantics and which fits all the requirements for digital-centric systems.

C. Solver Instantiation and Simulation setup

This phase of the elaboration is performed once the clusters hierarchy is built and once a pair of models of computation (moc and $master\_moc$) is associated to each cluster. Its purpose is to finalize the creation of internal data structures required to support the semantics of simulation. In this phase, as two types of objects (*modules* and *solvers*) are manipulated using common functions, an abstract base class **scm_core::scm_moc_if** is defined that allows the encapsulation of these functions. This abstract class, as is shown in Figure 7, can represent either modules or solvers. *Modules* are the primitive blocks instantiated by the user, which belong to a MoC **X**, and which implement a set of function prototypes defined by the class **scm_X::scm_module**. *Solvers* are **scm_X::scm_Y_solver** objects, automatically instantiated in a cluster during this phase. These solvers perform the synchronization operations of the cluster elements (primitive modules and sub-clusters)

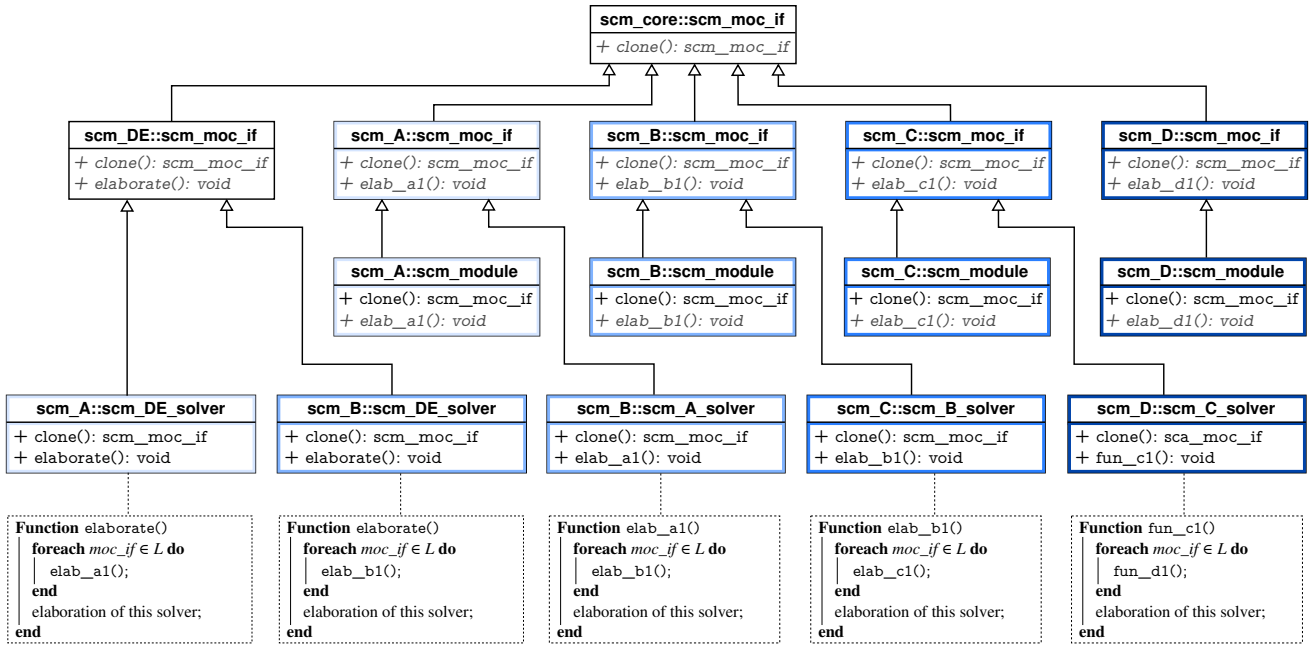

Figure 7. Minimal class hierarchy defined for the example shown in Figure 6.

belonging to MoC **X**, according to the semantic rules imposed by MoC **Y**.

As the simulation is supposed to be initiated by a DE kernel, the **scm_DE::scm_moc_if** represents the base interface to be respected during the addition of other MoCs. Three steps are required in order to add a new MoC **X**. First, the base class **scm_X::scm_moc_if** has to be defined, which shall be respected by the modules belonging to a MoC **X**, and shall be also implemented by the solvers that wish to be executed within the context of MoC **X**. It has to inherit from the abstract class **scm_core::scm_moc_if**. Second, the base class **scm_X::scm_module** has to be defined, from which derive the primitive blocks belonging to the MoC **X** and which are defined by the user. This base class has to inherit from the class **scm_X::scm_moc_if**. Third, the solver **scm_X::scm_Y_solver** has to be defined, which derives from the base class **scm_Y::scm_moc_if** of MoC **Y** with which it wishes to communicate. When a new MoC is added, it can interact with one or more of existing MoCs. Hence, several solvers may be defined in the third step. An example is shown in Figure 7, where two solvers are instantiated for MoC **B**: **scm_B::scm_DE_solver** and **scm_B::scm_A_solver**, performing the synchronization interactions with MoCs **DE** and **A**, respectively.

Using the proposed encapsulation of functions and the hierarchy of clusters provided by the second phase of the elaboration, two stages are performed: the instantiation of solvers and the simulation setup (Figure 4, phase III). In the first stage, based on the Prototype Creational Design Pattern [9], a factory is implemented for the creation of the new solver objects. This factory stores a dictionary of prototypes that should be cloned for a cluster. The dictionary of prototypes corresponds to a map containing a pair of MoCs **<X,Y>** and the **scm_X::scm_Y_solver** that should be cloned and assigned to a cluster belonging to MoC **X**, which will be executed within the context of MoC **Y**. For the clusters hierarchy shown in Figure 6, the dictionary contains the information summarized in Table I.

Table I
DICTIONARY OF SOLVERS AVAILABLE FOR THE CLUSTERS HIERARCHY SHOWN IN FIGURE 6.

<X,Y>	**Prototype to Clone**
<A,DE>	scm_A::scm_DE_solver
<B,DE>	scm_B::scm_DE_solver
<B,A>	scm_B::scm_A_solver
<C,B>	scm_C::scm_B_solver
<D,C>	scm_D::scm_C_solver

The recursive, bottom-up function proposed for the automatic instantiation of solvers is shown in Algorithm 2. The creation of these new solver objects during the algorithm execution is done by calling the `find_and_clone()` function (Algorithm 2, line 9). This function is provided by the factory and receives as parameters the MoC, the master MoC, and the list of **scm_core::scm_moc_if** (modules and solvers) associated to a cluster. The function is responsible for searching in the dictionary of prototypes the pair of MoCs received, for identifying the prototype to be cloned, and for calling the `clone()` function that is implemented in the previously identified prototype class. The `clone()` function creates a new instance calling the constructor of the prototype class, which also stores the list of modules/sub-solvers received as

```
1  Function instantiate_solver(cl)
       Data: Cluster cl.
       Result: Solver s instantiated for cl.
2      foreach sub_cl in the clusters list of cl do
3          m = instantiate_solver(sub_cl);
4          add m in the moc_ifs list of the sub_cl;
5      end foreach
6      if master_moc of cl is NULL then
7          return NULL;
8      end if
9      s =
       find_and_clone(moc,master_moc,moc_ifs);
10     set the solver s in cl;
11     return s;
12 end
```

Algorithm 2: Recursive function proposed for the solver instantiation.

third argument in the `find_and_clone()` function.

Once the solvers have been instantiated, the second stage is performed. In order to elaborate and configure each module and solver for simulation, the `elaborate()` function is called for each sub-cluster contained in the root of the clusters hierarchy. It is guaranteed that all the solvers instantiated to those sub-clusters respect the context imposed by the **scm_DE::scm_moc_if** base class. The call to the elaboration functions defined for each MoC (`elab_a1()`, `elab_b1()`, `elab_c1()` and `elab_d1()`) is automatically performed in cascade thanks to the implementation mechanism detailed in Figure 7. It is worth noting that during the elaboration of a solver, each of its modules and sub-solvers are elaborated before it is itself elaborated. At the end, all the structures, modules and solvers are ready for simulation.

V. Outlook on Simulation

Clustering and MoC solver instantiation are bottom-up processes that recursively build the hierarchical simulation infrastructure and define the general simulator behavior from the aggregation of MoC solver leaves. The simulation process itself relies on a top-down prefix-order traversal of the same infrastructure. Starting with the root solver, node-related local simulation contexts (some input stimuli, a floor timestamp, and a temporal horizon) are recursively propagated to each node of the hierarchy. This depth-first traversal generally implies shifting from a coarse representation of time to a more continuous one, according to the MoCs present in the hierarchy. Provided a node representing solver **Y** has received from its master node solver **X** a local simulation context, solver **Y** is allowed to perform its local simulation using its associated solver and to generate appropriate local simulation contexts for its subtree. The floor time stamp is the lower time boundary **Y** starts simulating at, and the temporal horizon represents the upper time limit. Each solver **Y** manages its own time scale and advances at its own pace according to the temporal horizon constraints imposed by its master solver **X**. Solver **Y** produces simulation results that are directly converted and integrated into solver **X** as solver **X** quantities.

VI. Conclusion

This paper addressed the requirements for building an efficient simulation engine for multi-disciplinary virtual prototyping, as materialized in the SystemC Multi-Disciplinary Virtual Prototyping (MDVP) framework. More precisely, the paper focused on the elaboration task that consists in creating the necessary information and C++ data structures for simulation. The concept of master-slave to describe MoCs relationships, allows to define a generic clustering, follow by a generic solver instantiation. Eventually the simulation setup is performed. All this elaboration process is performed in a MoC-independent way.

The elaboration task is also important for properly detecting inconsistencies in the composition of possibly heterogeneous components, namely in the dimensions and units associated to elements representing physical quantities and in the expected master-slave MoC relations.

Hence, the SystemC MDVP framework can be enriched with new MoCs without the need for modifying the elaboration phase. Future work will be dedicated to the extension of the simulation phase from the outlook provided in the Section V to complete the development of the framework.

References

[1] Accellera System Initiative. Standard universal verification methodology, http://accellera.org/downloads/standards/uvm.

[2] Accellera SystemC AMS Working Group. *Standard SystemC AMS extensions 2.0 Language Reference Manual*. Accellera Systems Initiative (ASI), March 2013.

[3] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[4] H. Al-Junaid and T. Kazmierski. An Analogue and Mixed-Signal Extension to SystemC. *The Institution of Electrical Engineers Proceedings Circuits Devices Systems*, 152(6):1–10, 2005.

[5] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. 1994.

[6] J. Eker et al. Taming Heterogeneity - The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[7] FMI development group. Functional Mock-up Interface (FMI). https://www.fmi-standard.org/.

[8] Fraunhofer IIS/EAS. SystemC-AMS proof-of-concept (PoC) implementation. http://systemc-ams.eas.iis.fraunhofer.de/.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[10] F. Herrera and E. Villar. A framework for embedded system specification under different models of computation in SystemC. In *Proceedings of the 43rd (ACM/IEEE) Design Automation Conference (DAC)-2006*, pages 911–914. ACM/IEEE, 2006.

[11] IEEE Computer Society. *1666-2011 IEEE Standard SystemC Language Reference Manual*. IEEE, January 2012.

[12] T. Maehne and A. Vachoux. Supporting dimensional analysis in SystemC-AMS. In *Proceedings of the 2009 IEEE International Behavioral Modeling and Simulation (BMAS) Workshop*, pages 108–113. IEEE, 2009.

[13] T. Maehne and A. Vachoux. Bond graph support in SystemC AMS. In *Proceedings of the 10th International Conference on Bond Graph Modeling and Simulation (ICBGM) 2012*, pages 159–166. SCS, 2012.

[14] H. Patel and S. K. Shukla. *SystemC Kernel Extension for Heterogeneous System Modeling*. Kluwer Academic Publishers, 2004.

[15] M. C. Schabel and S. Watanabe. *Boost.Units 1.1.0. http://www.boost.org/doc/html/boost_units.html*.

[16] T. Uhle and K. Einwich. A SystemC AMS extension for the simulation of non-linear circuits. In *Proceedings of the 23 IEEE International SoC Conference (SOCC) 2010*, pages 193–198. IEEE, 2010.