An Approach of Flat Heterogeneous Modeling based on Heterogeneous Interface Components

Aimé Mokhoo Mbobi[1], Frédéric Boulanger[2], Mohamed Feredj[3]

Abstract *–Data processing applications are increasingly heterogeneous, mixing different technical domains such as telecommunications, digital and analog electronics, signal processing algorithms, etc. These domains have different methods of modeling and design that consider their components and relationships in different ways. To mix these different domains, modeling languages and platforms generally use a hierarchical approach where the hierarchy of the model is coupled to the change of model of computation. Moreover, heterogeneous components cannot be used and what happens when data crosses the boundary between two domains depends on the modeling environment.*
This paper presents a flat heterogeneous modeling approach that solves these issues by using Heterogeneous Interface Components, which gives more control to the designer on the semantics of the interactions between models of computation.

Keywords*: Heterogeneity, Embedded Systems, Modeling, Design, Model of Computation*

I. Introduction

In [1], modeling is defined as a formal representation of a given concept, system or subset whereas designing implies the implementation of successive models, where each one is the refinement of the previous. Hence, the first model may be viewed as the formal specification and the latest as its implementation. Thus, the goal of modeling is the exploration of models for final design whereas the goal of design is implementation. So, design and modeling are closely interdependent. On an abstract level, a model of a system can be regarded as a combination of different technical domains that may have different modeling and design methods. However, these domains do not consider their components and the interactions between them in the same manner. As a result, in each domain, interactions between components are controlled and governed by a specific set of physical laws or axioms called "*Model of Computation*" *(MoC)*. A comparative and detailed study of models of computation is presented in [2].
Currently, the modeling and design of complex systems naturally uses several MoCs that match diverse technical implementations; e.g. a third generation multi-media cell phone uses several technical domains such as algorithms for image and signal processing, software, physical interfaces, micro-waves and radio features, etc. The setup of such a system may imply the connection of subsystems that are not already using the same MoC. Such a system is called a "*Heterogeneous System*".

I.1. Heterogeneous Modeling Approaches

Heterogeneous modeling is simply modeling by using several MoCs. Since most systems are heterogeneous in nature, heterogeneous modeling provides more natural and more complete models. For instance, being able to use both state machines and Synchronous Data Flows allows the designer to describe explicitly the control in a digital signal processing model. If we were limited to the SDF MoC, control and data processing would have to be coded together and the model would be less expressive and much more difficult to maintain. In order to mix different MoCs, current heterogeneous modeling tools can use either an amorphous or a hierarchical approach of heterogeneity.

• *Amorphous Approach*: Many modeling and design environments support heterogeneous modeling, but they generally focus on a fixed set of MoCs that are generally continuous and discrete signals for electrical engineering or state machines and differential equations for hybrid systems. Since they use few MoCs that are known beforehand, they can define the union of these MoCs. Moreover, the complete knowledge of the interactions between these MoCs allows computing the behavior of a heterogeneous model. This approach is called the "Amorphous Approach". A digital to analogical signal converter with digital inputs and analog output is an example of such systems. SIMULINK and VHDL-AMS are modeling and design environments that rely on this approach.

• *Heterogeneous Approach*: Modeling and design tools that support an open set of MoCs cannot build the

Manuscript received January 2007, revised February 2007, accepted March 2007

union of the MoCs because they are too numerous and not known beforehand. These tools require that each component obeys only one MoC. Since two connected components obey the same MoC, all interconnected components must obey the same MoC. However, the hierarchical abstraction makes it possible to model a contained component using a MoC which is different from the one used for its container. Therefore, changes of MoC can only occur at the boundary of a component: this leads to the "hierarchical heterogeneous modeling" paradigm used by several modeling and design environments such as EL GRECO [3], PTOLEMY II [1], ROSETTA [4] etc. This hierarchical approach is obviously an efficient way of managing system complexity [5], [6], [7], [8]. This complexity can come from the structure or behavior of the system [3] what leads to structural and behavioral hierarchy exposed in [9], [10]. Structural hierarchy is used to identify self-contained subsystems whereas behavioral hierarchy is a way of considering a complex process as the result of sequential or concurrent single processes. In hierarchy, block diagrams support abstraction and refinement. Abstraction allows a block diagram to be compressed into a single block; refinement allows a block to be expanded into a block diagram [11]. Therefore, the fundamental question is not this hierarchical relationship between blocks; it is rather the impact of the coupling between changes of MoC and hierarchy on the modeling, design and maintainability of applications. This coupling reduces reusability, corrupts modularity and makes the maintainability of systems complex. We think that the hierarchy in a heterogeneous model should not depend on modeling tools. It should rather represent the compositional structure of a system according to its functional decomposability.

In this paper, the heterogeneity we are talking about is not from hardware and software partitioning such as defined in [12]. It is at the modeling and design levels. It is only related to the rules that govern the interactions between the components of a model of a system. An implementation of the system may use a single set of rules onto which different modeling approaches would be mapped. On the contrary, two systems modeled by using the same model may be implemented with different rules. For instance, the continuous time model may be used to model both a mechanical and an electrical system as shown in Fig. 1.

$$m\frac{d^2x_1}{dt^2} + a\frac{dx_1}{dt} + kx_1 = kx_2 \qquad (1)$$

$$R_1R_2C_1\frac{d^2v_1}{dt^2} + (R_1C_1 + R_1C_2 + R_2C_2)\frac{dv}{dt} + v = U \qquad (2)$$

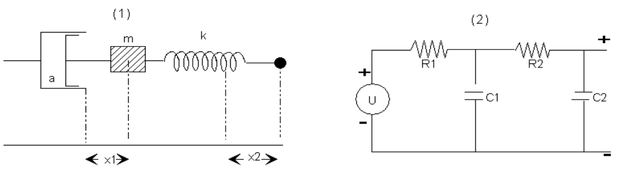


Fig. 1. Two different systems modeled by the same equations.

II. Hierarchical approach issues

Hierarchy manages the complexity of a system by hiding non pertinent internal details at a given level of modeling. When you look inside a component, you may see either a low level description of the component behavior when the component is atomic or primitive or a model of this component in the same modeling environment when it is composite. In both cases, the interface of the component hides the internal details of its behavior and insulates these internal details from the outside environment in which the component is used. So the inner and outer MoCs can be different. It is still necessary to define how those MoCs interact and how data is transformed when crossing a boundary [9].

Currently, most modeling tools use the hierarchical approach. But, this approach has some drawbacks leading to some issues: (1) model hierarchy is coupled with the changes of MoC and may not reflect the effective structure of the system, (2) components that have inputs or outputs that obey different MoCs cannot be used in a model, and (3) what happens when data crosses the boundary between two domains depends on the modeling environment.

II.1. Change of MoC implies hierarchy

Since changes of MoC may occur only when we change the level in the hierarchy of the model, ad hoc constructions arise at the boundary between two MoCs. This issue can be solved by either preserving or changing the semantic properties across MoCs.

According to the preservation of semantic properties across MoCs, only terminals that obey the same MoC can be connected, but a component may obey several MoCs. This way preserves the semantic properties along connections between terminals. Since components can obey several MoCs, we can use a third component dedicated to the change of semantics between two heterogeneous components in the same hierarchical level as shown in Fig. 2. Then, the heterogeneous behavior of the system can occur inside those components as part of their behavior and also as part of the behavior of the system.

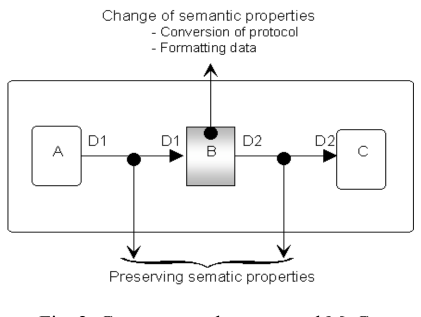


Fig. 2. Component obeys several MoCs.

As for the change of semantic properties across MoCs, here components obey only one MoC, but we can make connections between terminals across MoCs as shown in Fig. 3. To achieve this, the change of semantic properties mechanism between MoCs must be implemented in either the core or the ends of the connection. But, implementing the change of semantic properties mechanism between MoCs in the core of connections requires this connection to become active, to provide both the conversion of communication protocol and data formatting mechanisms. The challenge is that to be able to do so from one MoC to another, the connection must be able to fulfill the three conditions given in [13] : (1) to translate the common semantic properties, (2) to ignore the semantic properties in the first MoC that are not present in the target MoC, (3) to create the semantic properties in the target MoC that are not present in the first MoC

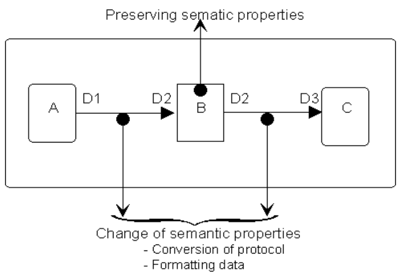


Fig. 3. Component obeys only one MoC.

Likewise, implementing the change of semantic properties mechanism between MoCs at the end of the connection requires this terminal to become active.

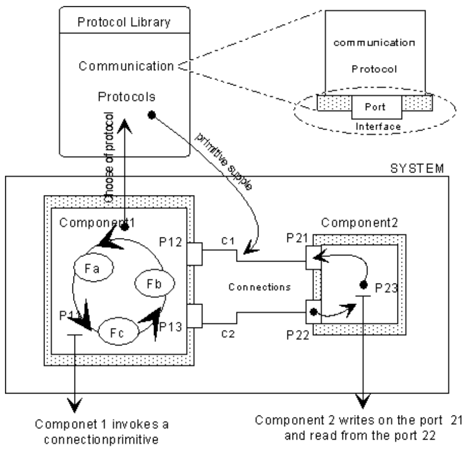


Fig. 4. The use of RPC.

To achieve this, some approaches use the Remote Procedure Call (RPC) mechanism where a component can call a communication primitive of a connection by reading from or writing to the interface port. As shown in Fig. 4, the component M1 chooses the communication protocol in the protocol library. Since it can have several implementations, the synthesis process chooses the appropriate protocol to the MoC used by the components M1 and M2 and provides the suitable primitive for the connection. In object-oriented philosophy, this concept of protocol library is often replaced by the power of the polymorphism technique. In PTOLEMY II for instance, this concept is replaced by the dual concept of polymorphism and hierarchical abstraction but on different hierarchical levels.

II.2. Banning of components using several MoCs

The second issue is the banning of components that have terminals which obey different MoCs in the same hierarchical level. For instance, an analog to digital converter could be modeled in a continuous time domain, the digital outputs being modeled with continuous signals with sharp changes. If such a model is close to the reality, it is not at the right abstraction level when one wants to consider the outputs as discrete sequences of values. On the contrary, the analogical to digital converter could be modeled using a discrete domain where the continuous inputs would be considered as sequences of discrete samples, turning the device into a resampler or a no-op. To solve this problem, a Heterogeneous Interface Component must be used.

II.3. Implicit transformation between MoCs

The third issue hides the transformations that occur at the boundary of two domains inside the "edge of the components". These transformations depend on the modeling tool and are therefore not explicitly stated in the model of a system. And the designer has neither a clear understanding nor the complete control of what happens when data crosses the boundary between two domains of the system. To solve this problem, two approaches may be used: (1) the first approach advocates to allow the designer to edit the edge of the components to specify how data is transformed when it goes through it. (2) The second approach advocates moving these transformations from the edge to the core of the components. This makes the component internal specification depend on the domain in which it is used, what impairs modularity and reusability.

II.4. Example and Goal

Consider the example shown in Fig. 5 of a signal rectifier to illustrate the issues of the hierarchical heterogeneous models. The top level uses flows of data samples, and the behavior of the detector is modeled using discrete events. When the flow of samples enters the detector, it is converted to a sequence of valued events. When an event is produced at the output, its value is used to build a data sample in the outer domain. This is an example of what may happen at the boundary of a component, and the important point is that these transformations depend on the modeling tool and are not specified in the model of the system. Since the data flow MoC which is used for that detector requires that a sample of data be produced on the output each time a sample is consumed on the input, the discrete event behavior of the detector must respect this condition. So even if the input signal does not change its sign and no event has to be produced, the detector must produce something on its output to obey the outer semantics.

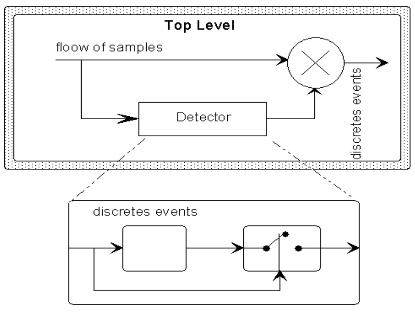


Fig. 5. Example of a hierarchical system.

Here, we have put a sampler that uses the value of the last emitted event to produce an output each time an input sample is consumed. We have to put this sampler in the internal model of the detector because of the external semantics. So the implementation of the detector depends on the context in which it is used, what impairs modularity and reuse.

The goal of this paper is not to banish the hierarchy, but to propose a new heterogeneous approach called ”*Flat heterogeneous Modeling*” built on hierarchical heterogeneous modeling. Its first goal is to dissociate the changes of MoC from the hierarchy by using components that have heterogeneous inputs or outputs; from where arises the use of several MoCs at the same hierarchical level. Secondly, to allow the explicit specification of the heterogeneous behavior in the HICs, from where comes the explicit specification of what happens at the boundary between different MoCs by the system designer.

III. Flat heterogeneous approach features

To model the flat heterogeneous approach, we choose actor-oriented methodology [14], [15,] [16], [17]. We also choose to allow the connection only of terminals that obey the same MoC (as seen in section II.1) because its preserves the semantic properties across the connection. Additionally it has an advantage to support Heterogeneous Interface Components (HICs) that include the change of semantics between heterogeneous MoCs as a part of their behavior. The HICs naturally appear in the models; therefore, they raise the question of making them obey several MoCs knowing in advance neither the MoCs nor their number. Since a HIC is heterogeneous, it has ports that obey different MoCs. When it interprets an input, it translates its meaning in the associated MoC into its internal semantic. When it produces an output, it translates the data from its input and its internal state into the semantics of this output according to its MoC. So, the behavior of a HIC can be decomposed into as many secondary behaviors as there are MoCs in its interface, and these secondary behaviors are coupled by the internal semantic of the HIC. As a result, the behavior of the HIC according to a MoC can influence its behavior according to another MoC. Each one of its secondary behaviors is viewed as “a bridge” between a MoC and its total behavior. Thus, the specification of the behavior of a HIC leads to an internal representation of the semantics of the inputs according to their respective MoCs, and the translation of this internal representation into outputs. Therefore, a HIC can use as many domains as necessary. HICs require a heterogeneous model of execution for the interpretation of the flat model in the hierarchical approach.

III.1. First attempt

When considering a HIC having a behavior at the boundary of only two heterogeneous MoCs, we see that it must be represented in both MoCs. Thus, the heterogeneous execution model divides the system at the border of both MoCs and creates two subsystems : Ω_1 and Ω_2 controlled by their MoCs. Unfortunately this configuration is physically impossible because a component cannot be simultaneously in Ω_1 and Ω_2.

III.2. Flat approach by hierarchical abstraction

Because of this physical constraint, we have intuitively outsourced the HIC from Ω_1 and Ω_2 and used hierarchical abstraction. That gives to Ω_1 and Ω_2 the respective ports Ω_1^{Out} and Ω_2^{In}. Inside each subsystem, these ports are connected to the homogeneous components, and outside, they are connected to the ports Hic^{In} and Hic^{Out} of HIC. This configuration is obviously realizable, and has been implemented. Unfortunately, it is hardly realistic because it makes the heterogeneous execution model depend on the MoCs used by Ω_1 and Ω_2. Also, it is not compatible with an open set of MoCs.

III.3. Flat approach by non-hierarchical abstraction

Let us replace this previous concept by that of “non-Hierarchical Abstraction” [9] [8] in which the interface of a subsystem is deprived of communication ports. So there is no communication channel connecting the subsystems to each other. This approach relies on the following:

• Projection of HIC: For each MoC used by a HIC, we create a component that represents this HIC in a subsystem that obeys the MoC. We call it the “Projection” of the HIC onto the subsystem as shown in Fig. 6. This component is homogeneous because it has only the ports of the HIC which obey the local MoC, other ports being masked during the projection process. This concept introduces two types of particular channels: Heterogeneous and Homogeneous Abstract channels

• *Heterogeneous Abstract channels* are channels between the projections of a HIC on various subsystems. Such channels cannot be represented by direct connections because the communication occurs inside the HIC. As shown on Fig. 7, after HIC projection, the original channel between the actors A_1 and A_2 disappears since it cannot be handled by the homogeneous MoC. It appears as an abstract heterogeneous channel between A_1 and A_2 which contains both the channel c_1, the projection Hic_{Tx}, the HIC, the projection Hic_{Rx} and the channel c_2. We call it an "Abstract heterogeneous channel". Such a channel obeys the rules of the heterogeneous domain.

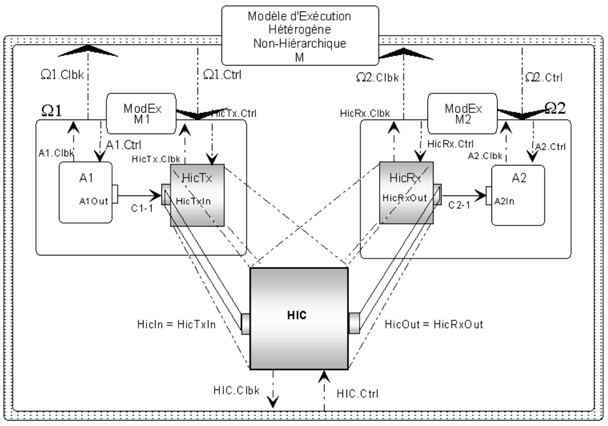


Fig. 6 a HIC projected into two domains.

• *Homogeneous Abstract channels:* let us consider the example on Fig. 7. Because of the non-hierarchical abstraction, a channel that connects two ports of A_1 and A_3 that do not belong to the same subsystem but use the same MoC is implemented by using an abstract channel. This channel includes the channels from A_1 to A_3 and two components: the "Relays Tx" transmitter which transmits data to a corresponding "Relays Rx" receiver. We call such a channel a "*Homogeneous Abstract channel*".

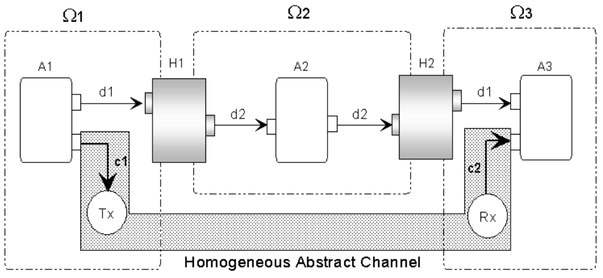


Fig. 7. Homogeneous Abstract Channel.

Data available on the input of Tx are also available on the output of Rx, and the scheduler of the subsystem will make sure that the behavior of Ω_1 is computed before the behavior of Ω_3 so that Tx can transmit the value to Rx before the output of Rx is used.

IV. Modeling of a HIC

IV.1. Structure of a HIC before the system partitioning

Let's take a HIC having only one input and one output. It has a set of variables noted HIC.X = {HicIn, HicOut, hicParameter, hicState} where HicIn is the input port from which the HIC reads data, HicOut is the output port by which the it sends data, hicParameter and hicState are the parameters and current state of the HIC.

Operations of the HIC before system partitioning

A HIC has heterogeneous inputs and outputs allowing it to communicate with two or several heterogeneous components. In addition, it must be able to provide a heterogeneous behavior at the borders of both MoCs that it uses. Then, a HIC has a set of data flow operations and a set of control operations.

• *Data flow operations of a HIC*: the behavior and communication of a HIC are determined by two sets : Hic.Comp and Hic.Comm respectively, set of its computational operations and set of its communication operations. Hic.Comp determines the way a HIC computes its behavior at the border of adjacent MoCs and Hic.Comm determines the way it sends the data towards a consumer actor. *Hic.Comm=Hic.Read* ∪ Hic.Write={**`existData(),read(),isFull(),`** **`write()`**} *and* Hic.Comp={**`computeBehavior()`**}. Hic.Comp enables it to compute its heterogeneous behavior. So, this is where the designer would specify the heterogeneous behavior of the system (interpretation of data when passing from a MoC towards another).

• *Control flow operations* : A HIC is controlled by Hic.Ctrl and Hic.Clbk respectively call back operations sent to him by the MoC and recall operations that it sends to the MoC. Hic.Ctrl = {**`initialization(),`** **`preCondition(),trigger(),postCondition()`**} and Hic.Clbk = {**`finish()`**}, operation by which it notifies the end of its activities to the MoC.

• *Set of execution operation of a HIC* : Hic.Oper is the union of all the operations that a HIC can perform: Hic.Oper = Hic.Comm ∪ Hic.Comp ∪ Hic.Clbk = {**`existData(),read(),isFull(),write(),`** **`computeBehavior(),finish()`**}. We give a summary of HIC operations in table 1 below.

TABLE I
HIC OPERATIONS BEFORE PARTITIONING

Data Flow		Control Flow
COMPUTATION	COMMUNICATION	
How to compute data (Behavior)	How to get and how to send data	When to compute and when to communicate

Since a HIC is projected in various subsystems, it is fair to wonder about the way it will share its internal variables and operations with its projections.

IV.2. Structural partitioning of HIC

During the partitioning process, the projection of a HIC is simply the same HIC placed in a given subsystem. However, since this projection has removed its ports that do not obey the MoC of the subsystem onto it is projected, it looks like a homogeneous actor of this subsystem. The global behavior of the HIC is at the border of several MoCs, and can be performed only by a component not belonging to any subsystem. This is why the internal variables will be managed by the HIC itself. After system partitioning, a projection is connected to a channel via its port as shown on Fig. 8. This is why, all the interface variables of the projections will be shared with the interface variables of the HIC that is the source of this projection.

IV.3. Operational partitioning of a HIC

The partitioning is made according to communication and behavior. Thus, some operations are performed by the projections of the HIC and others by the HIC itself. In the same way, from the point of view of control, the partitioning is elaborated in such a way that the heterogeneous execution model performs some controls and delegates execution to the regular MoCs.

• *Communication operations* : According to the resulting structure of the HIC, the projections are connected to the different communication channels in the subsystems. As for the communication operations, **`existData()`** and **`read()`** will be performed by the input ports of HIC that is shared with the corresponding ports of its projections. **`isFull()`** and **`write()`** will be performed by the output ports of HIC that is shared with the corresponding ports of its projections.

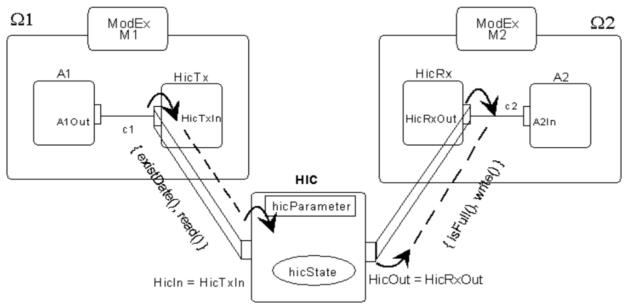


Fig. 8. Structure and operations partitioning of HIC.

Behavior Computational Operation: Because of the heterogeneity, the behavior computation operation is assigned to a HIC and **`computeBehavior()`** is invoked by its **`trigger()`** which is itself invoked by the **`trigger()`** of its projections. The execution of a HIC is included in the execution of its projection.

• *Control operations*: The heterogeneous execution model is relieved of data transfer between two heterogeneous actors. It is not informed of the different MoCs used by the various subsystems that it controls. Thus, it has to delegate all the local control of the subsystems to their respective local MoC. Thus, the control of each projection is exclusively delegated to the MoC that governs its subsystem. Its MoC ensures its triggering and detects the end of its activities.

• A projection is controlled by its MoC by the operations **`initialize()`**, **`preCondition()`**, **`postCondition()`**, **`trigger()`**. It notifies the end of its activities to its MoC by **`finish()`**. Its call-backs **`getModEx() and requestBehaviorComputing()`** request respectively the nature of its MoC and the triggering of its original HIC by the flat heterogeneous execution model in order to compute its behavior.

• A HIC is directly controlled by the flat heterogeneous execution model by its **`initialize()`**, **`preCondition()`**,**`trigger()`**,**`postCondition()`**. By **`getModEx()`**, it will request the nature of its MoC and it notifies the heterogeneous execution model of the end of its activities by **`finish()`**.

IV.4. Specification of the behavior of a HIC

When a projection of a HIC onto a subsystem is activated by the local MoC that governs this subsystem, a HIC must process inputs, produce outputs or update its internal state. The scheduling algorithm of the heterogeneous domain ensures that all the projections of a HIC that take input data are activated before any projection that must produce outputs is activated. However, when we design a HIC, we do not know in which order its inputs will be available because we do not know how it will be projected on the domains it uses. It is not possible to specify the behavior of the HIC for each of its domains, because it may happen that several terminals that use the same MoC be projected onto different subsystems. Therefore, the only solution to specify the behavior of a HIC is to specify how its state is updated for each possible set of known inputs, and to compute its outputs from the known inputs and the current state. This makes programming HICs less simple than regular components because the code must check which inputs are known before processing them.

V. Modeling of Flat Heterogeneous Execution Model

The simulation of such a flat heterogeneous system requires a heterogeneous execution model that is able to interpret the flat model in the hierarchical approach. This execution model ensures the management of HICs, the partitioning of the system into homogeneous subsystems, the delegation of the computation of the behavior of the subsystems to their MoC, the static scheduling of the subsystems and the coordination of the communications between subsystems. We designed this execution model to operate into three phases: partitioning of the system into subsystems, scheduling of the subsystems and execution.

V.1. Partitioning the system

During the initialization phase, the execution model divides the system at the border of the MoCs, and creates homogenous subsystems. The HICs are projected onto each subsystem to which some of their ports belong, and the other actors are transferred to their associated subsystems. Then the execution model copies the connections from the original system to the subsystems and generates virtual dependencies between the projections of a HIC on the same subsystems.

The partitioning algorithm minimizes the use of abstract homogeneous channels by first performing a topological sort on the actors. Then, for each actor, in an order which is compatible with the topological sort, it looks for a subsystem that uses the MoC of the actor. If such a subsystem exists, it puts the actor there if the dependencies allow it; else a new subsystem is created to host the actor. The condition on the dependencies must be respected so that the subsystems can be scheduled. To put an actor A_i in a subsystem S_j, the following conditions must hold:

- A_i must use the same MoC as S_j
- There is no path from A_i to any actor of S_j that goes through a HIC.

These conditions ensure that there won't be cross dependencies between subsystems. For instance, on Fig. 9, if we put A_{d1} and C_{d1} in the same Ω_1 subsystem (they both obey the MoC d_1), and B_{d2} and D_{d2} in Ω_2 subsystem (they both obey the MoC d_2), we cannot schedule the two subsystems because the projection of the HIC in Ω_1 must be activated after B_{d2} which is in Ω_2, and the projection of the HIC in Ω_2 must be activated after A_{d1} which is in Ω_1, so there is no possible schedule of Ω_1 and Ω_2. In this case, the algorithm will build four subsystems, each one containing an actor and a projection of the HIC.

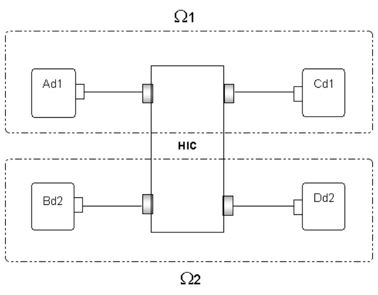


Fig. 9. Ω1 and Ω2 cannot be scheduled.

The above conditions may hold for more than one subsystem for a given actor, in which case we choose to put the actor in the subsystem that already contains a projection of the HIC which belongs to the same segment as the actor if any, or we will put it in the most recently created compatible subsystem.

V.2. Scheduling of the subsystems

Our flat heterogeneous execution model relies on the schedulers of the subsystem MoCs to schedule the actors inside, so that it does not depend on the semantics of their MoC. The precedence between subsystems is induced by the abstract homogenous channels (between actor relays) and the abstract heterogeneous channels (between HIC projections). A subsystem Ω_1 precedes another subsystem Ω_2, and we note $\Omega_1 < \Omega_2$ if either Ω_1 contains an output relay and Ω_2 contains the matching input relay or Ω_1 contains a projection of a HIC which has inputs and Ω_2 contains a projection of the same HIC which has outputs. The scheduling of the subsystems should be done according to the precedence induced by the HICs and the relays used to preserve homogenous communication channels across subsystems. However, because of the reasons that lead to the creation of relays, the precedence induced by relays on subsystems is always also induced by the HICs. Therefore, it is sufficient to take only the precedence induced by the HICs into account for scheduling the subsystems. After partitioning the system, the heterogeneous execution model builds a skeleton of the partitioned system that contains only the projections of the HICs and their dependencies. A topological sort of this skeleton is then used to determine the precedence relation on subsystems, and any order which is compatible with this relation of precedence is a possible scheduling of the subsystems.

V.3. Example of partitioning and scheduling

Consider the flat heterogeneous system of Fig. 10.

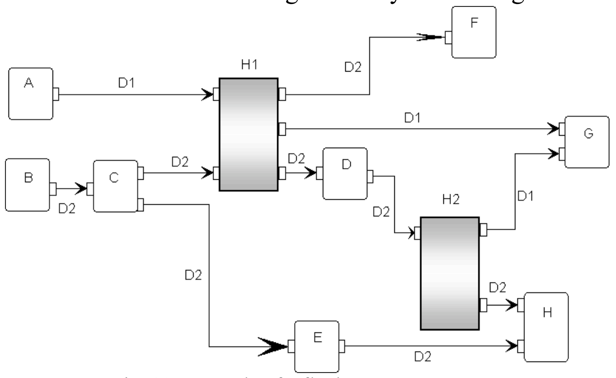


Fig. 10. Example of a flat heterogeneous system.

The partitioning of this example is shown on Fig. 11. (1) A cannot be put in the same subsystem as any other actor since actors that are on the same side of H1 do not use the same MoC, and actors which use the same MoC are reached by crossing a HIC, so A will be alone with the projection of H1 in its subsystem. (2) B, C and E use the MoC D2 and will be put in the same subsystem. However, D and H cannot be grouped with them because there is a path from C to D through H1, and there is a path from C to H through H1 and H2. (3) F can be put with D and E because they use the same MoC and do not communicate through a HIC. G is the only actor that uses D1 to the right of H1, so it will be in its own subsystem. (4) Since E and H are connected but are not placed in the same subsystem, there will be two relay actors Tx and Rx that handle communications along this abstract homogenous channel.

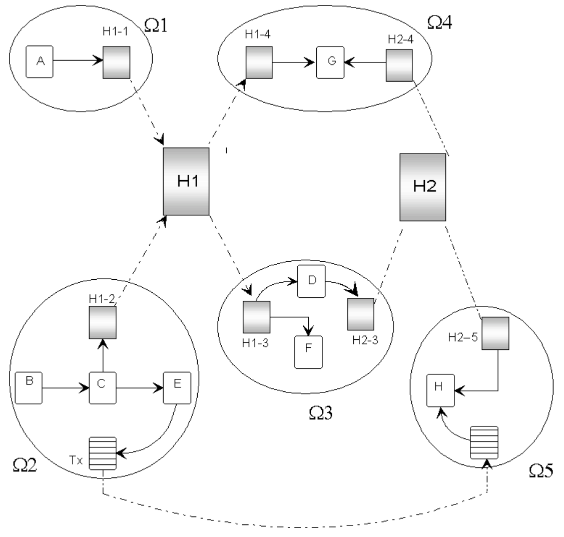


Fig. 11. Partitioning of the system.

The skeleton of this partitioning is shown in Fig. 12 and yields the following precedence relations:

$\Omega_1 < \Omega_3$	$\Omega_2 < \Omega_3$	$\Omega_4 < \Omega_4$
$\Omega_1 < \Omega_4$	$\Omega_2 < \Omega_4$	$\Omega_3 < \Omega_5$

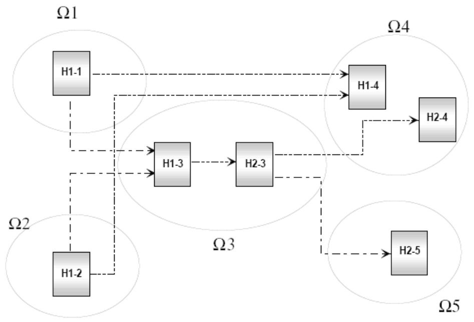


Fig. 12. Skeleton of the example system.

That gives the four following schedules:

$\Omega_1\ \Omega_2\ \Omega_3\ \Omega_4\ \Omega_5$	$\Omega_1\ \Omega_2\ \Omega_3\ \Omega_5\ \Omega_4$
$\Omega_2\ \Omega_1\ \Omega_3\ \Omega_4\ \Omega_5$	$\Omega_2\ \Omega_1\ \Omega_3\ \Omega_5\ \Omega_4$

V.4. Execution of a heterogeneous iteration

Scheduling makes it possible to activate the subsystems according to a well-defined sequencing. When this sequencing executes a complete cycle, we call it an "iteration" of the system. Since we forced on the heterogeneous execution model not to know the MoCs used by its subsystems, the interactions between the execution model and the subsystems will only be the activations and calls-back.

V.5. Execution in Ω_1

The execution in Ω_1 is shown on the Fig. 13. Suppose that Ω_1 receives an initial triggering coming from the heterogeneous execution model M.
(1) M triggers Ω_1; (2) Ω_1 performs A_1; (3) A_1 checks the availability of HicTx; (4) A_1 writes data to channel C_1; (5) A_1 notifies Ω_1 of the end of its activities; (6) Ω_1 triggers HicTx; (7) HicTx requests the triggering of the original HIC by M ; (8) M triggers the HIC; (9) the HIC computes its behavior; (9-1) the HIC checks data available on C_1; (9-2) the HIC reads data from C_1; (10) the HIC notifies M of the end of its activities; (11) M returns; (12) HicTx notifies Ω_1 of the end of its activities. The same execution process happens in Ω_2.

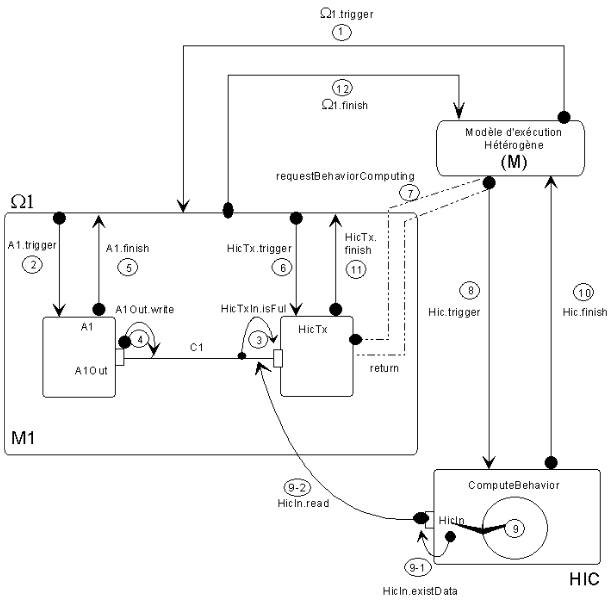


Fig. 13. Execution process in Ω_1.

VI. Integration and simulation in Ptolemy II

VI.1. Integration

This approach has been integrated in PTOLEMY II [1]. This hasn't required any change to the Ptolemy kernel. Its implementation is just another Ptolemy II domain. A flat heterogeneous model is a composite actor controlled by a heterogeneous Director called "FHDirector" and using one or several HICs that extends the HicActor class which inherits from the class AtomicActor; any class of HIC must extend it. The methods **`initialize()`** and **`fire()`** must be overwritten to allow the computation of the heterogeneous behavior. Depending on the specification of a HIC, the methods **`prefire()`** and **`postfire()`** could also be overwritten to implement some specific constraints. The class diagram of HicActor is shown on Fig. 14

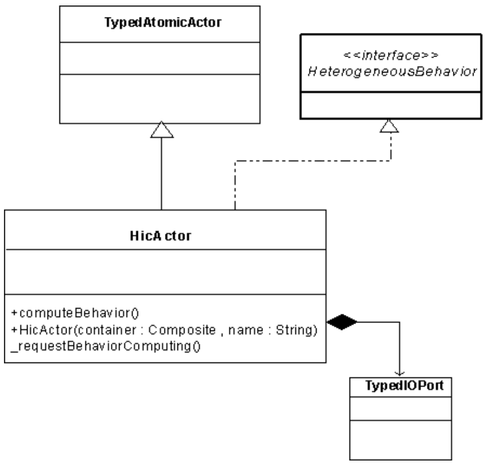
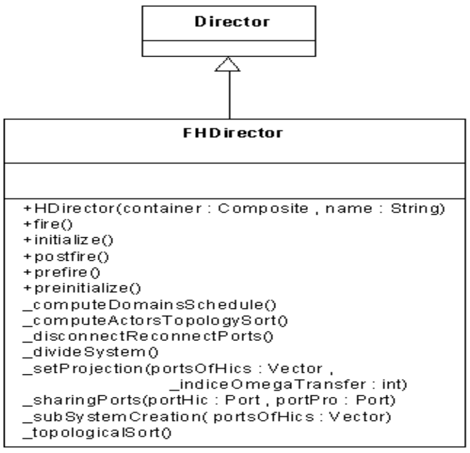


Fig. 14. UML Class diagram of HicActor.

The class FHDirector which extends the class Director has also been defined. The methods **`preinitialize()`**,**`initialize()`** and **`fire()`** are overwritten to make it possible for FHDirector to deal with the additional tasks of subsystems creation, HIC projection, actors motion their associated subsystems, ports management in subsystems and scheduling of the subsystems.



Director
FHDirector
+HDirector(container : Composite , name : String)
+fire()
+initialize()
+postfire()
+prefire()
+preinitialize()
_computeDomainsSchedule()
_computeActorsTopologySort()
_disconnectReconnectPorts()
_divideSystem()
_setProjection(portsOfHics : Vector , _indiceOmegaTransfer : int)
_sharingPorts(portHic : Port , portPro : Port)
_subSystemCreation(portsOfHics : Vector)
_topologicalSort()

Fig. 15. UML Class diagram of FHDirector.

The FHDirector execution phase is performed during its **`fire()`** operation. Partitioning and scheduling phases are respectively executed in **`preinitialize()`** and **`initialize()`**. **`preinitialize()`** being where the system creates receivers and validates attributes and ports, receivers for consuming actors (including projections) will be supplied by the regular Directors after partitioning. This occurs when FHDirector invokes **`preinitialize()`** in its super class witch invokes the same method on the subsystems, which also invokes that operation in their respective directors to create receivers on projections, validate their attributes and ports. During this operation, FHDirector invokes its **`divideSystem()`** to divide the system at the border of MoCs. Then, **`computeDomainsSchedule()` is called** to schedule the subsystems before the execution phase. In the iteration phase, if the subsystem is ready to be performed, FHDirector invokes its **`fire()`** method in which the projections request the activation of their HICs from FHDirector.

VI.2. Simulation in Ptolemy II

Let's model a simple system that uses three MoCs: SDF, DE and DT at the same level of hierarchy.

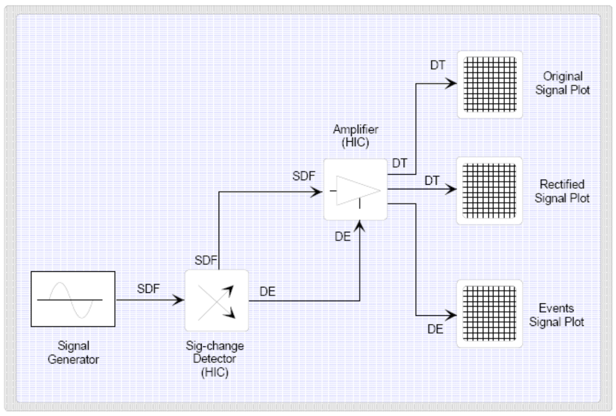
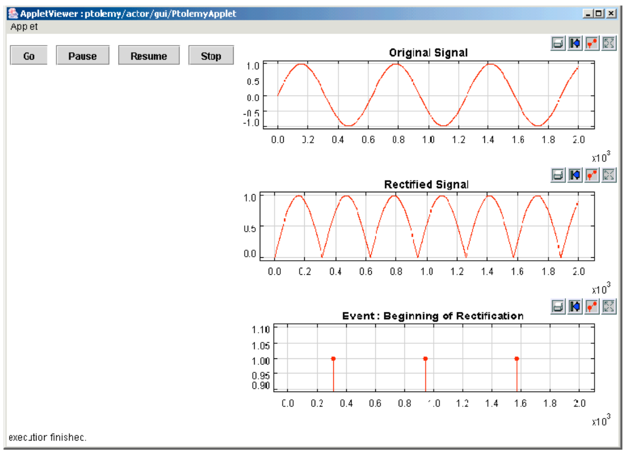


Fig. 16. Simulated system

The system was built by assembling actors using the Java API of PTOLEMY II. Fig. 17 shows the result of the simulation in PTOLEMY II. The upper plot is the original sinusoid signal, the middle one is the amplified signal and the lowest is the events that drive the rectification.



AppletViewer : ptolemy/actor/gui/PtolemyApplet
Applet
Go
Pause
Resume
Stop
Original Signal
Rectified Signal
Event : Beginning of Rectification
x10³
execution finished.

Fig. 17. Result of the simulation

VII. Conclusion

We presented a flat heterogeneous modeling approach that allows more natural modeling of heterogeneous interface components and gives more control on the semantics of the interactions between MoCs. This approach offers several advantages since the use of components that have heterogeneous inputs or outputs allows the use of several MoCs at the same hierarchical level of a model, and the explicit specification of the heterogeneous behavior by the designer in the HICs allows him to specify what happens at the boundary between different MoCs. Nevertheless, this approach does not support dependency loops between heterogeneous subsystems. However, if a loop is local to a subsystem and if the corresponding model of computation supports loops, the loop is accepted and its semantics will be given by the domain of the subsystem.

References

[1] E. Lee and all. *Heterogeneous concurrent modeling and design in Java*, (Volume 1, introduction to Ptolemy II). Technical Report No. UCB/EECS-2007-7, UC Berkeley, January 11 2007.

[2] E. 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(12), December 1998.

[3] J. Buck and R. Vaidyanathan, *Heterogenous modeling and simulation of embedded systems in el greco,* Proc. the 8th international workshop on Hardware/software, codesign, San Diego, California, USA, no. ISBN:1-58113-268-9, November 2000, pp. 142–146

[4] C. Kong and P. Alexander. *The Rosetta meta-model framework*. In Proceedings of the IEEE Engineering of Computer-Based Systems Symposium and Workshop (ECBS'03), april 2003.

[5] B. Lee and E. Lee, *Hierarchical concurrent finite state machines in Ptolemy,* Proc. International Conference on Application of Concurrency to System Design, Fukushima, Japan, March 1998, pp. 34–40.

[6] B. Lee and E. A. Lee, *Interaction of finite state machines and concurrency models,* Proc. the Thirty Second Annual Asilomar Conference on Signals, Systems and Computers, Pacific Grove, California, November 1998.

[7] A. Girault, B. Lee, , and I. E. A. *Lee, Fellow, Hierarchical finite state machines with multiple concurrency models, Proc. the DATE99 conference, March 1999, pp. 382–383.*

[8] M. Mbobi, F. Boulanger, and M. Feredj, *Execution model for non-hierarchical heterogeneous modeling,* Proc. the 2004 IEEE International Conference on Information Reuse and Integration (IEEEIRI 2004), Las Vegas, Nevada, USA, no. ISBN: 2004113902, November 1998, pp. 139–144.

[9] A. M. Mbobi, *Modélisation hétérogène non-hiérarchique,* Ph.D. Thesis, Université Paris XI, December 2004. Available from http://www.supelec.fr/ecole/si/biblio/Mbobi.pdf

[10] J. Daveau, *Spécification système et synthèse de la communication pour le co-design logiciel/matériel,* Ph.D. Thesis, INPG and TIMA Laboratory, December 1997. Available from http://tel.archives-ouvertes.fr/docs/00/04/54/18/PDF/tel-00002996.pdf

[11] L. de Alfaro and T. A. Henzinger, *Interface theories for component based design,* Proc. the First International Workshop on Embedded Software, EMSOFT, 2001.

[12] F. Vahid and D. Gajski, *Specification partitioning for system design,* Proc. the IEEE Design Automation Conference, June 1992, pp. 219–224.

[13] W.-T. Chang, S. Ha, and E. Lee, *Heterogenous simulation - mixing discrete-event models with dataflow*, Journal of VLSI Signal Processing, Kluwer Academic Publishers, vol. 15, pp. 127–144, 1997.

[14] G. Agha, I. A. Mason, S. F.Smith, and C. L. Talcott, *A foundation for actor computation,* Journal of Functional Programming, vol. 7(1):1-72, 1997.

[15] J. Liu, J. Eker, X. Liu, J. Reekie, and E. Lee, *Actor-oriented control system design : A responsible framework perspective*, IEEE Transaction on Control System Technology, special issue on Computer Automated Multi-Paradigm Modeling, March 2003.

[16] Keutzer, S. Malik, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, *System-level design: Orthogonalization of concerns and platform-based design,* IEEE transactions on computer aided design of integrated circuits and systems, vol. 19(12), pp. 1507–1522, December 2000.

[17] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, , and D. Sturman, *Abstraction and modularity mechanisms for concurrent computing,* Proc. IEEE Parallel and Distributed Technology: Systems and Applications, vol. 1(2), pp. 3–14, May 1993.

Authors' information

[1,] RedKnee Inc., Department of Product Operations, 2560 Matheson Blvd East, Mississauga, L4W 4Y9, Ontario - Canada

[2,] Supélec, Department of Computer Science, Plateau de Moulon, 3 rue Joliot-Curie, 91192 Gif-sur-Yvette cedex, France

[3,] University of Science and Technology Houari Boumediene, Faculty of Electronique and Computer Science. Department of Computer Science USTHB, 32 Al-ALIA, Alger, Algeria

Aimé Mokhoo Mbobi is a Technology Advisor at RedKnee Inc. in Canada. He holds a Telecom Engineering Degree from Enic-Telecom Lille1 in 1994, a Masters Degree in Computer Science from Ecole des Mines de Paris in 2002, and a Ph.D in Computer Science from the University of Paris XI in 2004, both in France. His research activities include the specification of new mechanisms which would enable interactions between components governed by different models of computation.
Dr. Mbobi is a member of the International Institute of Informatics and Systemics (IIIS).

Frédéric Boulanger is a professor at Supélec, a major French grande école. He got his engineering degree from Supélec in 1989, and a PhD in Computer Science from Paris-Sud University in 1993. His current interest is in heterogeneous modeling and in the precise definition of the interactions between models of computation.

Mohamed Feredj is an Assistant Professor at the University of Sciences and Technology H. Boumediene, He has a Computer Science engineering degree from the University of Sciences and Technology H. Boumediene, Algeria in 1997. He also holds a Masters Degree in Computer Science from the University of Versailles, France in 2002 and a Ph.D in Computer Science from the University of Paris XI, France in 2005.
Dr. Feredj is interested in Software Engineering, especially on Heterogeneous Systems Modeling and Design.