The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

A Model of Domain-Polymorph Component for Heterogeneous System Design

Mohamed FEREDJ

University of Science and Technology Houari Boumediene, Faculty of Electronic and Computer Science. Department of Computer Science, USTHB, 32 Al-Alia, Beb Ezzouar, Algiers, Algeria.

Frédéric BOULANGER

Supélec, Département Informatique, 3 rue Joliot-Curie, 91192 Gif-sur-Yvette cedex, France

Aimé Mokhoo MBOBI

RedKnee Inc. 2560 Matheson Blvd East, Mississauga, L4W 4Y9, Great Toronto Area (GTA), Ontario-Canada

Abstract

Heterogeneous systems mix different technical domains such as signal processing, analog and digital electronics, software, telecommunication protocols, etc. Heterogeneous systems are composed of subsystems that are designed using different models of computation (MoC). These MoCs are the laws that govern the interactions of the components of a subsystem. The design of heterogeneous systems includes the design of each part of the system according to its specific MoC, and the connection of the parts in order to build the model representing the system. Indeed, this model allows the MoCs that govern different parts of system to coexist and interact.

To be able to use a component which is specified according to a given MoC, under other, different MoCs, we can use either a hierarchical or a non hierarchical approach, or we can build domain specific components (DSC). However, these solutions present several disadvantages. This paper presents a new model of component, called *domain polymorph component (DPC)*. Such a component is atomic and is able to execute its core behavior, specified under a given MoC, under different host MoCs. This approach is not a competitor to the approaches above but is complementary.

Key words: Heterogeneous Embedded Systems, Heterogeneous Design, Software Engineering, Components, Actors.

1. Introduction

Heterogeneous systems are used in numerous application domains and mix several technical domains, what makes them complex. They are composed of subsystems that are designed using models of computation (MoCs) that suits the needs of their designers. The MoCs are the laws that govern the interactions of the components in a model. Modeling environments such as Ptolemy [9] support numerous MoCs, for instance Synchronous Data Flow (SDF), in which components communicate through flows of data samples at fixed rates; Continuous Time (CT), which is the model used in ordinary physics; Discrete Events (DE) or Synchronous Reactive (SR) in which components communicate through events that occur at specific dates or instants.

The model of a system must allow the MoCs that govern its components to coexist and interact.

Email addresses: Mohamed.Feredj@gmail.com (Mohamed FEREDJ), Frederic.Boulanger@supelec.fr (Frédéric BOULANGER), aime.mbobi@redknee.com (Aimé Mokhoo MBOBI).

Therefore, components that obey different MoCs which are compatible [3] must be able to communicate. Two main approaches are possible to make MoCs interact: the hierarchical approach [1] or the non-hierarchical approach [2]. The hierarchical approach structures the system into hierarchical levels where each level contains components that obey the same MoC. Therefore, each change of MoC implies a change of level in the hierarchical structure of the system. On the contrary, the non-hierarchical approach allows the use of several MoCs at the same level of the hierarchy, and therefore requires Heterogeneous Interface Components — components with ports that obey different MoCs — which are used as bridges between MoCs.

In a modeling environment, components which have no model in the modeling environment are said to be *atomic*. Their behavior is defined using another formalism, contrary to *composite* components whose behavior is described by a sub-model in the same modeling environment. Atomic components are designed according to a MoC, and have requirements on the environment in which they can be used. So, in order to use an atomic component specified according to MoC_s, in a model that obeys a different MoC_i, we can either wrap it in a composite component that uses MoC_s and use hierarchy to adapt between MoC_s and MoC_i, or build a domain specific component (DSC) which specifically adapts the behavior of the original component to MoC_i. DSCs are atomic components whose behavior is guaranteed to be correct only in their target MoC.

However, these approaches present several disadvantages. The hierarchical approach creates artificial hierarchical levels that do not reflect the real structure of the system, impairs reusability, and makes the semantics of the interaction between MoCs implicit. The non-hierarchical approach requires that the semantics of the interactions between MoCs be specified for each pair of MoCs, what leads to a combinatorial explosion with the number of MoCs used. Last, DSCs are expensive because they are specific to a MoC and must be generated from the same specification for each MoC under which we want to use them.

To overcome these problems, we propose a new model of component, the "domain polymorph component" (DPC) [6]. This model is complementary to the above approaches. DPCs are atomic and are able to execute their core behavior, specified using a given MoC, in the context of different host MoCs. By decoupling the semantics of the specification and the semantics of the execution context, our approach improves modularity. The adaptation to the semantics of execution is automatic, so domain-polymorph components are easily reusable. It is possible to customize their adaptation to the host MoC in order to get explicit control on the interactions between MoCs, what facilitates the maintainability and the validation of the system. We have integrated the model of domain-polymorph components in Ptolemy II [9], without modifying its kernel. Finally, to validate our approach, we designed an example system using our model of domain-polymorph component and the hierarchical approach.

2. Heterogeneous Design Approaches

The design of heterogeneous systems consists in building an executable model which describes their behavior and their pertinent properties. This model is built from components that have a communication interface and obey specific MoCs. This leads to a set of interconnected components that obey different MoCs, whence the problem of heterogeneity. There are several approaches to solve this issue.

2.1. *Heterogeneous Hierarchical Design*

The hierarchical approach structures the system into hierarchical levels where each level contains components that obey the same MoC. Therefore, changing the MoC implies moving to another level in the hierarchical structure of the system. This approach is an efficient way of managing the complexity of systems [8]. Actually, most design languages and platforms use the hierarchical approach. But, this approach presents some drawbacks :

(i) the hierarchy of the model is perturbed by the changes of MoC;
(ii) components with outputs and/or inputs that obey different MoCs cannot be used;
(iii) what happens to data and control at the boundary between MoCs depends on the design environment.

2.2. *Heterogeneous Non-Hierarchical Design*

The non-hierarchical approach [2][4][5], allows the use of several MoCs at the same level of the hierarchy, and therefore supports Heterogeneous Interface Components (HICs). Such components have inter-

The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

faces that obey different MoCs and can be considered as bridges between MoCs.

At first, this approach designs each subsystem of the original system in an independent manner. Then, it interconnects subsystems following their relationships of interconnection in the original system. Homogeneous components (that obey to same MoC) are directly interconnected and other components (that obey other MoCs) are connected through HICs. The result is a flat model in which different MoCs coexist at the same level of hierarchy.

The non-hierarchical approach manages the heterogeneity at execution time. When the flat model is executed, it is partitioned into homogeneous subsets in order to isolate MoCs. Then, the execution model projects HICs (following the relationships in the flat model) onto the subsets. The components of each homogeneous subset interact according to their MoC. The homogeneous subsets, which are seen as composite components, are at the same level and communicate through the HICs. To govern the interaction of the homogeneous subsets and HICs, the hierarchical approach defines a *heterogeneous* MoC.

3. Problem and Goal

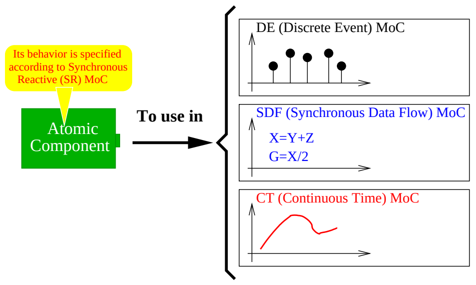
Its behavior is specified according to Synchronous Reactive (SR) MoC
Atomic Component
To use in
DE (Discrete Event) MoC
SDF (Synchronous Data Flow) MoC
X=Y+Z
G=X/2
CT (Continuous Time) MoC

Fig. 1. SR component to be used under DE, SDF and CT.

Some atomic components can be used in several MoCs, but only because they have a generic semantics (for instance, "producing a constant value") which can be interpreted in various MoCs (constant function of time in CT, repetition of the same value in SDF). Such components are said to be *domain polymorph* in Ptolemy, but we prefer to qualify them of *generic*. However, it is not possible to use an atomic component specific to MoC$_s$ under another MoC$_i$, because different MoCs assume different properties for components. Figure 1 shows an atomic component with a behavior specified according to SR, and which cannot be used directly under DE, SDF or CT. To solve this problem, we must manage the heterogeneity between the MoC used for the specification of the component and the MoC under which the atomic component will be used. This can be done according to the following approaches:

3.1. *Use of the Hierarchical Approach*

With the hierarchical approach, each level of the hierarchy is seen as a composite component. So, we encapsulate the MoC$_s$ domain specific atomic component in a composite component that uses MoC$_s$. Then, we place this composite component inside the composite component that uses MoC$_i$ (the host MoC) under which we want execute the atomic component(figure 2).

This allows the execution of an atomic component specified according to MoC$_s$ under a different host MoC. However, this approach presents the drawbacks cited in section 2.1.

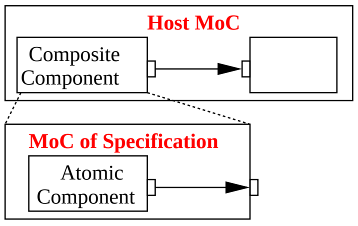

Fig. 2. Hierarchical approach for DSCs.

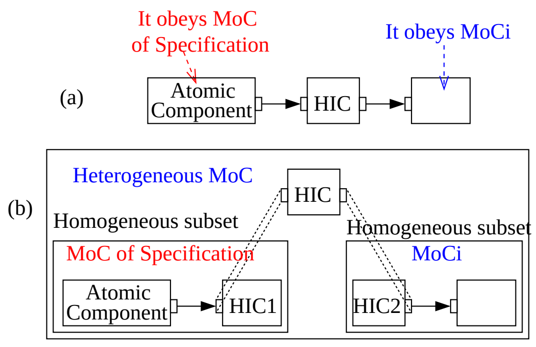

Fig. 3. (a) Flat model (b) Flat model transformed according to the non-hierarchical approach.

3.2. *Use of the Non-Hierarchical Approach*

With the non-hierarchical approach the different MoCs are at the same level. So, to allow an atomic component, the behavior of which is specified according to MoC$_s$, to interact and communicate with

The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

components that obey MoC$_i$, we must use a HIC in order to interconnect the atomic component to the components of MoC$_i$, what allows us to use a flat model (figure 3(a)). The HIC has heterogeneous ports: input ports obey MoC$_s$ and output ports obey MoC$_i$. At execution time, atomic components and the projection component of the HIC will be put into the same homogeneous subset (figure 3(b)).

This approach allows an atomic component which has a behavior specified according to a given MoC, to interact and communicate with components that obey another MoC. However, we have to specify the semantics of the interactions between MoCs for each pair (MoC$_s$,MoC$_i$) of MoCs.

3.3. *Use of the Domain-Specific Components*

A Domain-Specific Component (DSC) is an atomic component which has a structure and ports of communication that are specific to its MoC. So, the correct behavior of the DSC is not guaranteed under other MoCs.

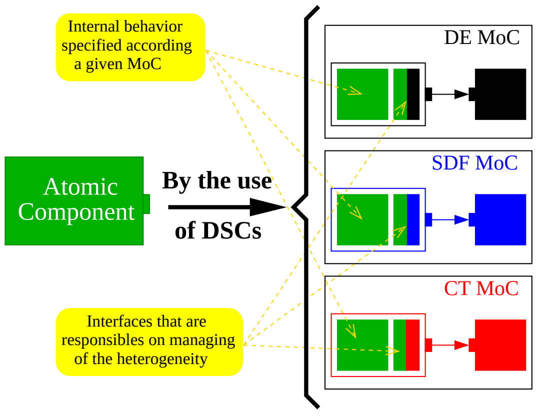
Internal behavior specified according a given MoC
Atomic Component
By the use of DSCs
Interfaces that are responsibles on managing of the heterogeneity
DE MoC
SDF MoC
CT MoC

Fig. 4. Use of domain-specific components.

With this approach, for an atomic components specified according to MoC$_s$, we must generate a different DSC for each permitted [3] MoC$_i$ under which we want use the atomic component (see figure 4). Each DSC has an internal behavior which obeys MoC$_s$ and which is adapted to the structure and ports of communication that are specific to MoC$_i$ for which it was generated.

To manage the heterogeneity between the internal behavior and the external MoC, we must specify an interface for managing heterogeneity. This interface performs the following tasks:

– transform data exchanged between other components and the internal behavior, because different MoCs may use different formats for data;
– adapt the semantics of the internal behavior to the semantic of the host MoC. This is realized by providing an execution environment for the internal behavior according to MoC$_s$ and by adapting its semantics to the semantics of the host MoC.

The use of DSCs to execute a given behavior, which is specified according MoC$_s$, under different host MoCs presents the following drawbacks:
(i) the adaptation of the semantics of the MoC of specification to the host MoC is ad hoc;
(ii) lack of reusability: because DSCs have a specific structure and specific communication ports, they can be used only under the MoC they were designed for;
(iii) duplication of code: since an atomic component specified according to MoC$_s$, must be implemented as a DSC for each MoC under which we want to use it, its internal behavior may be duplicated, so a change in one copy won't get propagated to the other copies;
(iv) no evolution: when a new MoC is added, the designers must make a new implementation of the atomic component for this MoC.
(v) the code is less readable since there is not a clear separation between the internal behavior and the domain specific aspects.
(vi) making a domain specific implementation of an atomic component requires a good knowledge of the implementation details of both MoC$_s$ and MoC$_i$. A designer may only master the semantics of the MoC he uses (MoC$_i$).

Our goal is to embed an atomic component, with a behavior specified according to a given MoC, in a model which obeys a different host MoC, without using any of the above mentioned approaches. We achieve this goal by using the model of DPC we present in the following.

4. Domain-Polymorph Component

A domain-polymorph component (DPC) is a component which is able to adapt its internal behavior (we call it *core*) to the semantics of a host MoC. This property has two aspects: first, such a component must provide its core with an execution environment that respects the semantics of the MoC of specification of its core; second, the component must be able to interpret data and control from its host MoC

The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

and to translate its outputs to the semantics of the host MoC. Such a component can be compared to a generic component because it can be used in several MoCs. However, it is different because its semantics is well defined (using its specification MoC) and is only adapted to the host MoC, while the semantics of a generic components is interpreted by the host MoC (CT interprets "constant" as a constant function of continuous time).

A DPC must have the following properties:

(i) it must be able to detect the nature of its host MoC;
(ii) after detecting the host MoC, it must be able to adapt its structure to what the host MoC expects (class, communication ports);
(iii) it must be able to use the communication and control primitives of the host MoC;
(iv) it must implement the communication and control primitives of its core behavior.
(v) it must be usable under all allowable [3] MoCs, even under MoCs that didn't exist at the time of its creation;
(vi) the design of the core behavior of a DPC must require a knowledge of the MoC used to define it only;
(vii) when the semantics of the core and the semantics of the host MoC can be adapted in several ways, the designer must be able to choose the one he wants because this choice is part of the design of the heterogeneous system.

Moreover, we want to respect the following constraints so that the notion of DPC is not restricted to one platform:

(i) DPCs must be portable: no change in the implementation of the MoCs are required for DPCs to operate correctly;
(ii) The implementation of the DPCs must be possible in any object-oriented language, so we don't rely on specific features like introspection or parameterized types.

5. Our Approach of Domain-Polymorph Component Design

In [7], we proposed an initial approach for DPC design and we complete it and improve it in this section.

Our approach is based on the actor-based design methodology [10] that consists in decomposing an heterogeneous system into elementary units of functionality, called actors. Actor orientation separates the functionality (modeled as actors) from the component interaction (modeled as frameworks), and gives well-defined scopes for model refinement and system realization. In the following, we present the successive steps in the evolution from domain specific components to DPCs.

5.1. *Step 1: Identification of the Concerns*

We have designed an atomic component according to the SR MoC, which is chosen as MoC of specification. Then we have generated DSCs (called Synchronous DSCs) from the synchronous reactive atomic component for several MoCs such as DE, CT, SDF, etc.

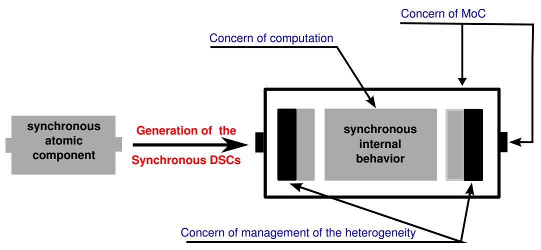
Concern of MoC
Concern of computation
synchronous atomic component
Generation of the Synchronous DSCs
synchronous internal behavior
Concern of management of the heterogeneity

Fig. 5. Synchronous DSCs and their transversal concerns.

By analyzing the set of DSCs, which preserve the synchronous semantics in different MoCs, we generalize to any DSC, which preserves a given semantics in its target MoC, and identify the following concerns as shown on figure 5:

– functional (basic) concern: the concern of computation that corresponds to the core behavior of the DSC;
– non-functional concerns due to heterogeneity:
  · communication: transforming data to and from the core, synchronizing with the host;
  · conservation of the internal semantics: provide the core with the environment needed for its execution;
  · adaptation to the host MoC: the component must implement the interface of the components of the host MoC (be of the "right" class, and have suitable communication ports).

5.2. *Step 2: Modular Domain-Specific Component*

By separating concerns and modeling each of them by a component, and then by interconnecting them, we obtain a new component that we call a "modular domain-specific" component (figure 6).

The components that model the concerns are detailed as follows:

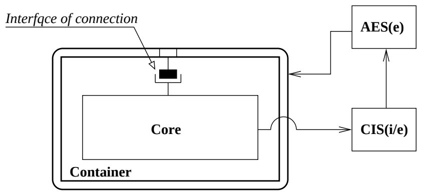

Fig. 6. Architecture of the Modular DSC.

– Container Component: it models the concern of adaptation to the host MoC. It has no ports but uses the connection interface to create them at run-time according to the needs of the Core component. A Container is domain-specific but can encapsulate any permitted Core component. The Container hides the MoC specific aspects from the Core, what guaranties the first part of the “usability” property (number ‘v’) of the DPC;
– Core Component: it models the functional concern. It has a generic interface that allows it to be encapsulated by any suitable Container. A Core component is therefore reusable under any permitted MoC. It interacts with its external environment in a transparent way by performing the operations provided by the CIS component;
– AES(e) Component (Adaptation to the External Semantics): this component models the concern of communication and it is reusable only under its MoC and for any permitted Core and CIS components. The AES provides the CISs with the communication operations needed to communicate under the host MoC. The AES(e) is only specific to the MoC which is used as external MoC;
– CIS(i/e) Component (Conservation of the Internal Semantics): this component models the concern of conservation of the internal semantics and is reusable only under its MoC but for any Core component that obeys to same MOC. The CIS provides an execution environment to the Core and implements it using the operations of the AES. The CIS(i/e) is only specific to the MoC which is used as internal MoC. AES and CIS(i/e) guarantee properties iii and iv of the DPC;

5.3. *Step 3: Flexible Domain-Specific Component*

The modular domain-specific component is a step toward the reusability of a core behavior under several MoCs, but it must be built specifically at compile time. To solve this problem, i.e. to build the modular domain-specific component dynamically at run-time, we proceed as follows:

(i) Since a Container may accept many different types of Cores, we use a Factory to instantiate the Core from its name at runtime;
(ii) Since the Core may use any permitted semantics, we use a policy of selection to associate the right CIS to the Core and the AES at runtime. The policy of selection guaranties the first part of the property ii of the DPC.

The result is a ”flexible domain-specific component” (see figure 7) where the solid lines show compile-time links and the dotted lines show runtime links.

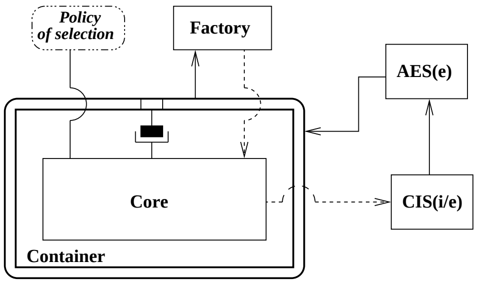

Fig. 7. Architecture of the Flexible DSC.

5.4. *Step 4: Domain-Polymorph Component*

The flexible DSC allows us to plug any permitted Core behavior in a given domain-specific container. However, to use the same core behavior under another MoC, we must first create a container which is suitable for this MoC. Indeed, we do not have a real DPC yet.

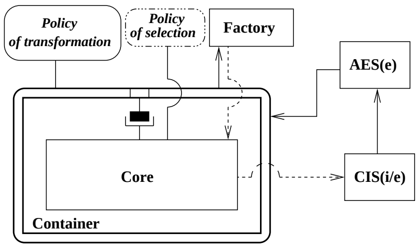

Fig. 8. Architecture of the DPC.

To achieve real domain-polymorphism, we add a policy of transformation to the Container (see figure 8). This policy guaranties the property i (detection) and the second part of the property ii (adaptation) of the DPC. It first identifies the MoC under which the DPC is used and then transforms the

The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

Container to allow it to have the structure and all the features required by the identified host MoC. Second, this policy selects and instantiates the right AES, and links it to the Container. Then, when any DPC is used under a MoC, the policy of transformation transforms it into a flexible domain-specific component, and the factory component and the policy of selection allow it to have the full and final structure required by its MoC.

5.5. *Step 5: Improvement of the DPC*

The DPC obtained in step 4 satisfies our goals. However, since each of its CISs components is only specific to an internal MoC and an external MoC, the DPC needs a polynomial number of CIS components. Let N be the number of compatibles MoCs [3] to consider, and suppose that any MoC can be used as semantics of specification (internal MoC) as well as semantics of environment (host MoC). Then, to use a DPC under any permitted MoC with a Core specified according to any pemitted MoC, we must design N^2 CIS components.

To reduce the number of CIS components, we apply again the separation of concerns on the abstract architecture of the CIS. We identify the concerns shown on figure 9: the concern of external MoC; the concern of passage; and the concern of internal MoC.

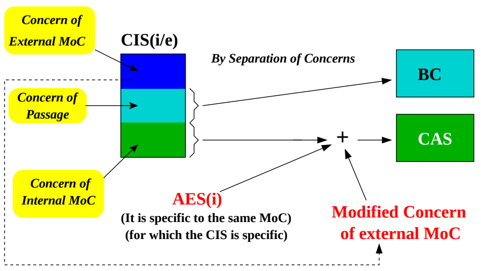
Concern of External MoC
CIS(i/e)
By Separation of Concerns
BC
Concern of Passage
+
CAS
Concern of Internal MoC
AES(i)
(It is specific to the same MoC)
(for which the CIS is specific)
Modified Concern of external MoC

Fig. 9. Application of the separation of concerns on CIS component.

After separation of the concerns, we model the concern of passage by a component that we call BC (Border Component). We keep the concern of internal MoC for which we add both the behavior of the AES component and the concern of external MoC that we modify, and we model them by a component that we call CAS (Conservative and Adapter of Semantics).

The main property of the CAS component is that it can be used both for preserving the internal semantics (it will be called internal CAS and noted iCAS) when its MoC is used as internal MoC (semantics of specification) and for adapting to the host MoC (it will be called external CAS and noted eCAS) when its MoC is used as external MoC. Indeed, to use a Core component specified according to a given MoC, we use only one iCAS specific to this MoC instead of using N CIS components. So, one CAS component replaces N CIS components at the internal level and one AES component at the external level. However, the BC component is the part in which the designers of a system can specify the transformations (by interpretation, by sampling, etc.) of data exchanged between external and internal MoCs, and this without modifying the rest of the DPC. The BC guaranties property vii (design choice) of the DPC.

The optimal architecture of the DCP is obtained by replacing the N CISs by one iCAS and each AES by one eCAS. To interconnect iCAS, eCAS and BC components, we introduced a CS (Connector of Semantics) component, see figure 10. The CAS and CS components guarantee the second part of the property V of the DPC.

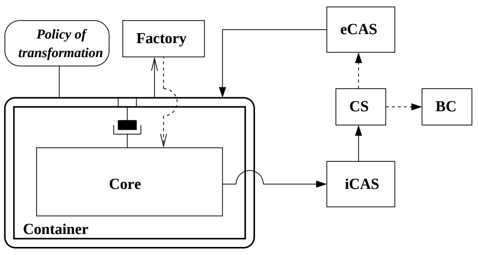

Fig. 10. Architecture of the improved DPC.

6. Behavior of the Domain-Polymorph Component

At runtime, the first task performed by the DPC is the assemblage phase. This phase allows the DPC to have a structure that is specific to both the external MoC (under which the DPC is used) and the internal MoC (MoC of specification of the Core component). After the assemblage phase, the DPC has a set of variables noted $DPC.V = \{DPC\_In, DPC\_Out, DPC\_Parameter, DPC\_State\}$ where DPC_In and DPC_Out are respectively input ports (from which the DPC reads data) and output ports (through which the DPC sends data). These ports are requested by the Core and created by the DPC.$DPC\_Parameter$ and DPC_State are the parameters and current state of DPC.

The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

6.1. *Operations*

When a DPC is used under a given external MoC, it must communicate with the components connected to it and it must guarantee the execution of the Core component according to the internal MoC. For this, a DPC has four sets of operations: data flow operations, control flow operations, synchronization flow operations and passage flow operations, (see figure 11).

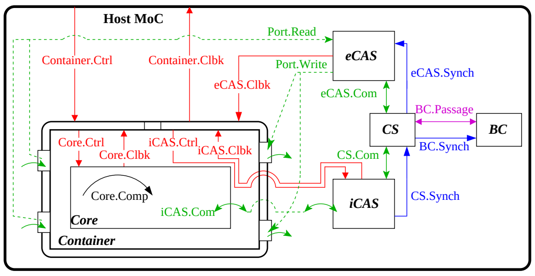

Fig. 11. Operations of the improved DPC.

6.1.1. *Data flow operations*

They are determined by a set of computational operations, noted *DPC.Comp*, and a set of communication operations, noted *DPC.Com*. Operations in *DPC.Comp* are in charge of computing the behavior of the Core component. Operations in *DPC.Com* tell how the DPC gets and sends data from and toward its external environment. *DPC.Comp* = {*computeBehaviorCore*} and *DPC.Com* = *iCAS.Com* ∪ *CS.Com* ∪ *eCAS.Com* ∪ *Port.Read* ∪ *Port.Write* = {*existData, read, isFull, write, haseCASData, geteCASData, haseCASPlace, sendeCASData, hasData, getData, hasPlace, sendData, hasInputData, receiveData, hasFreePlace, emitData*}.

6.1.2. *Control flow operations*

Control operations specify when the DPC computes and when it communicates:

– the external MoC controls the DPC by *Container.Ctrl* and *Container.Clbk*. *Container.Ctrl* are callback operations triggered by the external MoC and implemented by the DPC. *Container.Clbk* are callback operations triggered by the DPC and implemented by the external MoC. *Container.Ctrl* = {*initialization, preCondition, trigger, postCondition*}, are the operations by which the external MoC activates the DPC, and *Container.Clbk* = {*finish*}, is the operation by which the DPC notifies the external MoC of the end of its activities;

– the Core component is controlled by *Core.Ctrl* and *Core.Clbk* that are respectively callback operations triggered by the Container and callback operations triggered by the Core on the Container. *Core.Ctrl* = {*initCore, preReaction, reaction, postReaction*} and *Container.Clbk* = {*finishReaction*};

– the iCAS is controlled by *iCAS.Ctrl* and *iCAS.Clbk* that are respectively callback operations triggered by the Container and callback operations triggered by the iCAS on the Container. *iCAS.Ctrl* = {*InitCASi, BoR (Begin of Reaction), EoR (End of Reaction)*} and *iCAS.Clbk* = {*Ready, NotReady*};

– *eCAS.Clbk* are callback operations that the eCAS triggers on the Container.

So, the control operations of the DPC are:
DPC.Control = *Container.Clbk* ∪ *Container.Ctrl* ∪ *Core.Ctrl* ∪ *Core.Clbk* ∪ *iCAS.Ctrl* ∪ *iCAS.Clbk* ∪ *eCAS.Clbk*

6.1.3. *Synchronization flow operations*

The synchronization between the external and the internal MoCs is managed by *CS.Synch*, *BC.Synch* and *eCAS.Synch* which are sets of synchronization operations. These operations allow the initialization and the activation of the CS, BC and eCAS components. The initialization is performed only once and the activation is performed before and after each reaction of the Core component. *CS.Synch* = {*InitCS, BoRCS, EoRCS*}, and *BC.Synch* = {*InitBC, BoRBC, EoRBC*} and *eCAS.Synch* = {*IniteCAS, BoReCAS, EoReCAS*} where InitX stands for “initialize component X” and BoRX and EoRX for “activate component X”.

So, the synchronization operations of the DPC are:
DPC.Synch = *CS.Synch* ∪ *BC.Synch* ∪ *eCAS.Synch*

6.1.4. *Passage flow operations*

BC.Passage is a set of passage operations that are responsible for the interpretation of data exchanged between the internal and external MoCs. Also, since the data consumed and produced by the MoCs don’t have the same format (for instance, the format of data in DE MoC is <Date, Value> while in SDF it is just <Value>), *BC.Passage* is responsible for the transformation of data. *BC.Passage* = {*DefaultInputData, DefaultOutputData, In-*

putDataFromOutputData, OutputDataFromInputData}. The behavior of the passage operations must be specified by the designer of heterogeneous systems because it is part of the system design.

6.1.5. *Execution of a DPC*

An external MoC executes a DPC by a set of activation cycles called iterations. The steps of an iteration of a DPC are shown on figure 12.

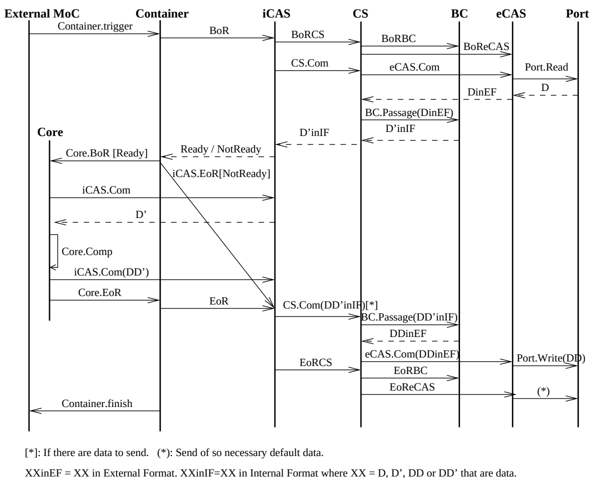

Fig. 12. UML’s sequence diagram showing the steps of iteration of the DPC.

6.2. *Degrees of polymorphism*

The DPC is the form of component that has the highest degree of polymorphism. However, it is not always possible to implement it in a completely automatic manner. Each component of a domain polymorph component can be implemented in any object-oriented language, but the policy of transformation of the Container and the policy of selection may be difficult or impossible to implement. According to the features of the programming language (introspection, reflexivity), these policies can be fully implemented in the language, or can be implemented using external tools and dynamic loading of code, or even worse, can be implemented by generating the necessary code at compile time only. The highest degree of polymorphism is therefore achieved only when the programming language offers enough possibilities to avoid any manual or automatic external activity to build and assemble the components of a DPC.

7. Integration And Simulation In Ptolemy II

7.1. *Integration*

The model of DPC has been integrated in the Ptolemy II [9] platform, because this platform supports many domains (a domain is the implementation of a MoC in Ptolemy), and its architecture is open to the creation of new domains. This integration hasn’t required any change of the Ptolemy II kernel.

Since Ptolemy II is programmed in Java and relies on static typing, with mandatory inheritance relations for actors in a given domain, we could not use the complete DPC model without modifying the kernel of Ptolemy II, so we used the FDSC model (see section 5.3). With our integration, it is now possible to use a Core behavior in any allowed domain by dragging a Container from a list of domain specific Containers as shown on figure 13(a) and dropping it onto the system being designed. Then, by setting parameters of the Container, a Core is associated to it, and it performs the assembly phase to take its integral form as shown on figure 13(c).

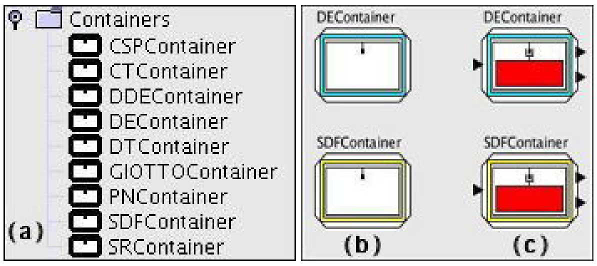

Fig. 13. (a) List of Domain Specific Containers
(b) Containers (c) DPCs after assembly phase

7.2. *Simulation*

To validate our model of DPC, we chose the production cell [11] case study as an heterogeneous system to design and simulate.

The path of a piece through the cell starts on a feed belt which conveys it to an elevating rotary table (see figure 14). If the table is in loading position, the belt pushes the piece on the table. Afterwards it is brought in a certain position by the table, where a robot can pick it up. With the help of its arm1 the robot takes it into the press, where the piece is forged. Now the arm2 of the robot transports it to a deposit belt. At the end of the deposit belt, a crane

The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

picks up the piece and unloads it on the feed belt again (so that the simulation can go on forever).

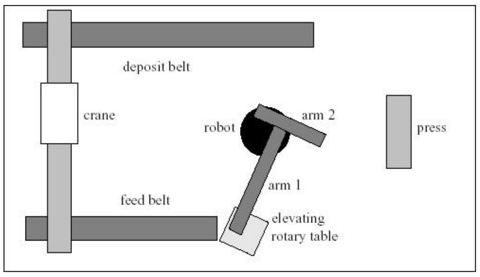

Fig. 14. Schema of the production cell.

During the production cycle, the six machines in the cell should avoid dropping the pieces to the floor or colliding against each other.

At the design level, we decompose each machine into two parts: a controller part and dynamic part. When a piece reaches the position from which it will be forwarded to the next machine, the controller part of the machine is responsible for the synchronization with the controller part of the next machine. So, the controller parts avoid colliding. The dynamic part represents the physical displacement of a piece on the machine.

However, to inform the controller part on the position of a piece at any time, each machine includes sensors. So, a controller part can see if a piece is at the position from which it will be forwarded. Indeed, the sensors allow the controller parts to avoid dropping pieces to the floor by stopping the machines when a piece reaches predetermined positions.

In this paper, we cannot present the complete design of the production cell. However, we present the design of the feed belt machine (FB), the other machines being designed in a similar way.

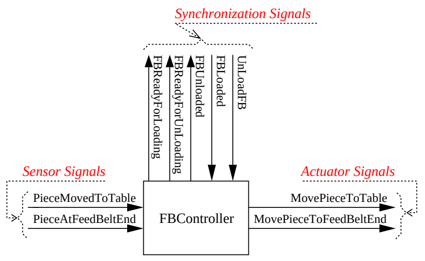

Fig. 15. Controller of the Feed Belt.

We specify the behavior of the controller of the FB, noted FBController, according to the Synchronous Reactive approach, which is suitable for specifying control, by using the Esterel [12] language. The synchronous module representing the controller of FB (see figure 15) has three categories of signals: synchronization signals, sensor signals and actuator signals. We developed a translator to build synchronous Core components usable in Ptolemy II from synchronous modules written in Esterel.

We represent the feed belt machine by a composite component, called FBWithFBController (see figure 16), in which we placed FBController and a composite component, called HSFB (Hybrid System FB), which represents the dynamic part of the Feed Belt machine. FBController and HSFB obey the DE MoC. So, FBController is a DPC that obeys the DE MoC and encapsulates a Core component which obeys the Synchronous Reactive MoC. HSFB is a FSM (Finite States Machine) component which has two states, one represents the displacement of the piece on the machine and the other represents the forwarding of a piece to another machine. Each state is refined by a composite component, called FBDynamic, in which we placed components representing the displacement of the piece, and a DPC, called DPCSensor, representing the sensor which informs FBController. Components of FBDynamic obey the CT MoC. The Core component of DPCSensor is also specified using the Esterel language, so DPCSensor obeys the CT MoC and its Core component obeys the SR MoC.

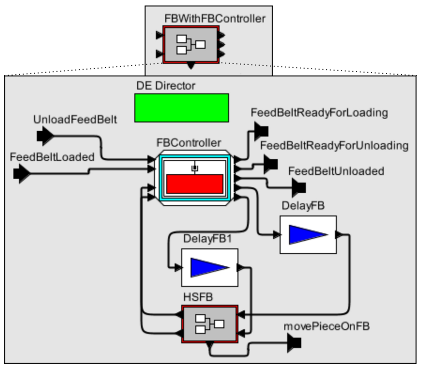

Fig. 16. Model of the Feed Belt.

The composite components representing the six machines are placed at the same hierarchical level (top level) and are connected to the cell central controller, called DPCCentralController, to form the model of the production cell. All components at this level obey the DE MoC. DPCCentralController is the manager of the synchronization between the controllers of the machines. It obeys the DE MoC and its Core component obeys the SR MOC.

The Journal of Systems and Software vol. 82 (2009) #1 112–120, Elsevier, doi:10.1016/j.jss.2008.05.031, ISSN: 0164-1212

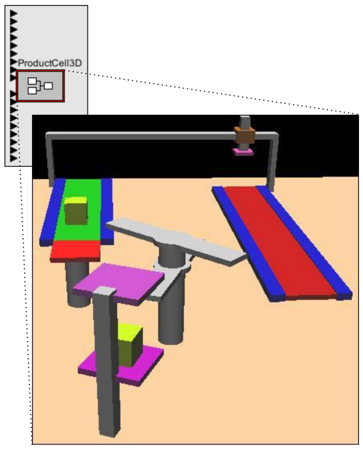

Fig. 17. The production cell in 3D.

To visualize the functioning of the production cell, we designed a 3D representation using the GR domain of Ptolemy II (see figure 17). This representation is animated according to the dynamics of the parts of the cell. The simulation shows that the model of the product cell is executed correctly, the semantic adaptation between the core and the container of the DPC being able to conciliate heterogeneous views of the different parts of the system.

8. Conclusion

We presented a model of domain polymorph component that allows the adaptation of the semantics of a Core behavior to the semantics of an external MoC. This model provides several advantages such as reusability, maintainability, etc. It allows the explicit specification of the exchange of data between the internal and the external MoC. When the adaptation can be done in several ways, the choice, which is part of the design of the system, is left to the designer. This model is not a competitor to the existing approaches but complements them to facilitate the tasks of the designers.

An implementation of domain polymorph components, limited to the flexible domain specific component level, has been realized in Ptolemy II. To validate our model, we designed the production cell benchmark by combining the hierarchical approach and DPCs.

In future works, to reduce the complexity of the architecture of the DPC, we will improve the model by using the aspect-oriented approach [13]. Also, in section 2.2, we introduced the notion of HIC, a component that works at the boundary of several domains, and DPCs may be a good implementation for HICs, with containers as the projections of the HIC on different domains, all embedding the same core in different semantics.

References

[1] J. Eker, J. W. Jannek et al., *Taming Heterogeneity – the Ptolemy Approach.* In proceeding of the IEEE, special issue on modeling and design of embedded software, volume 91, pp. 127-144, January 2003.

[2] A. M. Mbobi, F. Boulanger and M. Feredj, *An Approach of Flat Heterogeneous Modeling based on Heterogeneous Interface Components.* International Review on Computers and Software (IRECOS), Vol. 2, N. 2, pp. 179-189, March 2007.

[3] Antoon Goderis et al. *Composing Different Models of Computation in Kepler and Ptolemy II.* In Proc. of the 2nd Int. Workshop on Workflow Systems in e-Science (WSES07) at the Int. Conf. on Computational Science 2007, pp. 182–190 Beijing, China, May 27-30, 2007.

[4] A. M. Mbobi, F. Boulanger and M. Feredj, *Execution Model for Non-Hierarchical Heterogeneous Modeling* . In Proc. of the IEEE Int. Conf. on Information Reuse and Integration 2004 (IEEE IRI 2004), pp. 139-144, November 2004, Las Vegas, Nevada, USA.

[5] A. M. Mbobi, F. Boulanger and M. Feredj, *Non-hierarchical heterogeneity* . In Proc. of the Int. Conf. on Computer, Communication and Control Technologies, pp. 430-435, July 31, August 12, 2003, Orlando, Florida-USA. Best paper of -Process and Product Control-session.

[6] M. Feredj, *Etude des Méthodes de Conception de Composants Domaine-Polymorphes*, Ph.D. Thesis, University Paris XI, 236 pages, December 2005.

[7] M. Feredj, F. Boulanger, M. Mbobi, *An Approach for Domain Polymorph Components Design*, In proc. of the IEEE IRI 2004, pp. 145-150, November 2004, Las Vegas, Nevada, USA.

[8] A. Girault, B. Lee, and Edward A. Lee, *Hierarchical Finite State Machines with Multiple Concurrency Models*, IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems, Vol. 18, No. 6, pp. 742-760, June 1999

[9] C. Brooks, E.A. Lee et al. *Heterogeneous Concurrent Modeling and Design in Java*, EECS Department, University of California, Berkeley, UCB/EECS-2007-7/8/9, 684 pages, January 11, 2007.

[10] J. Liu et al., *Actor-Oriented Control System Design : A Responsible Framework Perspective.* IEEE Transaction on Control System Technology, vol. 12, No. 2, pp. 250-262, March 2004.

[11] C. Lewerentz and T. Lindner,*Formal Development of Reactive Systems: Case Study Production Cell.* Volume 891 of LNCS, 394 pages. Springer, january 1995.

[12] G. Berry. *The Foundations of Esterel.* MIT Press, 1998. Edited by C. Stirling, G. Plotkin and M. Tofte.

[13] G. Kiczales et al. *Aspect-Oriented Programming.* In Proc. of the ECOOP'97. Vol. LNCS 1241. pp. 220-242. Springer-Verlag. June 1997.