Author manuscript, published in "IEEE International Conference on Information Reuse and Integration. (IEEE IRI 2004), Las Vegas : United States (2004)"

hal-00262344, version 1 - 11 Mar 2008

An Approach for Domain-Polymorph Component Design

Mohamed Feredj
Mohamed.Feredj@supelec.fr

Frédéric Boulanger
Frederic.Boulanger@supelec.fr

Mokhoo Mbobi
Mokhoo.Mbobi@supelec.fr

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

Abstract

Heterogeneous modelling and design tools allow the design of software systems using several computation models. The designed system is built by assembling components that obey a computation model. The internal behavior of a component is specified either in some programming language or by assembling sub-components that obey a possibly different computation model.

When the same behavior is used in several computation models, it must be implemented in as many components as there are models, or, if the design platform supports it, it may be implemented as a generic component. Model-specific components require the recoding of the same core behavior several times, and generic components may not take model-specific features into account.

In this paper, we introduce the notion of domain-polymorph component. Such a component is able to adapt a core behavior to the semantics of several computation models. The core behavior is implemented only once and is automatically adapted to the semantics of different computation models.

Domain-polymorph components can be chosen by a system designer and integrated in a computation model: they will benefit from an appropriate execution environment and their semantics will be adapted to the host model. The designer will have the choice for several parameters of the adaptation. Contrary to generic components, such components adapt their behavior to the host model instead of letting the host model interpret their generic behavior.

We also present an implementation of the concept of domain-polymorph component in the Ptolemy II framework.

1. Introduction

The modelling and design environments that implement the actor-oriented design methodology [5] can support several domains. For instance, Simulink supports two domains: the continuous time and the discrete time domains; SPW[1] of Cadence supports only a data flow domain and Ptolemy II [2] supports an open set of domains which includes a discrete event domain, a synchronous data flow domain, a synchronous reactive domain, etc. A domain implements a Model of Computation (MoC) that defines the semantics of communication between ports and the flow of control among actors [4]. An actor is an encapsulation of parameterized actions and when it is activated by its domain, it performs its actions on input data to produce output data. Input and output data go through connections between ports. Ports and parameters are the interface of an actor [4].

We consider the case where a control logic drives data-processing operations in an application specific domain. If the same control logic must be used to drive similar operations in different domains, we must either implement the control as a generic component (a component that works in any domain), or we must implement it in each domain. The drawback of generic components is that they cannot take domain specific features into account, and the drawback of domain specific components is that they must be implemented in each domain because they have a domain-specific structure (class and communication ports) and must use domain-specific operations to communicate with other components.

We propose an alternative solution which avoids both the lack of adaptation of generic components and the need to implement each domain specific component. This solution allows the use of domain-polymorph components which adapt the semantics of their core behavior to the semantics of their host domain.

After an illustration of the drawbacks of domain-specific components, we will give the properties and the design-steps of domain-polymorph components and give an implementation of an infrastructure for such components in Ptolemy II.

[1] Signal Processing Worksystem

hal-00262344, version 1 - 11 Mar 2008

2. Domain-specific components

To illustrate the issues raised by the use of domain-specific components, we use a very simple example system that we implement in the Discrete Event and in the Synchronous Data Flow domains of Ptolemy II. This system has four components: a source of commands, a controller, a screen and an engine. The source asks the controller to perform operations on the engine. The controller has a state that tells whether an operation is possible or not. When it receives a request from the source, it performs it on the engine if the requested operation is possible. If it is not possible, it logs a message onto the screen.

In this example, we will consider the design of the controller only, as shown on figure 1.

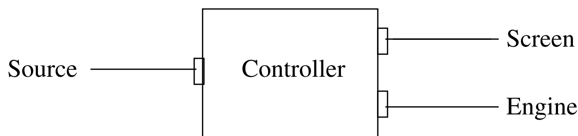

Figure 1. The system controller.

2.1. Implementation in the DE domain

In the Discrete Event domain, components share a global notion of time and communicate by posting events. Each event has a value and a time stamp. The implementation of our controller in the DE domain is a DE specific component with DE specific communication ports. Our controller is designed according to the reactive synchronous approach, so there is theoretically no delay between the occurrence of an input and the production of the corresponding output. The domain specific operations for our controller are therefore:

- the fact that a synchronous reactive component as a zero-delay processing implies that the time stamp of each output must be the same as the time stamp of the input that triggered it. This is achieved through DE-specific operations to get and set the time stamp of a data sample;

- the core of the controller must be executed in a suitable execution environment: input events from the DE domain must be transformed into input events for the controller, output events from the controller must be transformed into events for DE. Since the discrete event model is very close to the reactive synchronous model, this task is easy here, but this is not generally true.

2.2. Implementation in the SDF domain

In the SDF domain [6], an actor consumes a fixed number of data samples from each input port and produces a fixed number of data samples on each output port each time it is activated. Here too, the implementation of our controller will be an SDF-specific component with SDF communication ports. Since the native semantics of the controller is less adapted to SDF, more work is needed to make it behave soundly:

- events must be coded as data samples. For instance, we may choose to interpret a null data sample as "no event" and other data samples as effective events. The lack of reaction (no event produced) in the controller, must be translated into a default action in SDF which means producing a null data sample.

- when the SDF component is activated, we may not necessarily activate the reactive core of the controller. If no input sample is interpreted as an event, there is no need to make the core react. In some complex cases, we may even have to determine if the input samples form a meaningful event for the core or not, to know if the core must be activated.

This example shows that one cannot just wrap a given behavior in a DSC and hope it will work as expected in each domain.

2.3. Drawbacks of domain-specific components

Using domain-specific components to implement the control of a system in several domains has many drawbacks:

- lack of reusability: because DSCs have a specific structure and specific communication primitives, they can be used only in the domain they were designed for.

- duplication of code: since a given control behavior must be implemented as a DSC for each domain in which we want to use it, its core behavior may be duplicated, and a change to the core in one copy won't get propagated to the other copies.

- no evolution: when a new domain is added, the designers must make a new implementation of the system for this domain.

- the code is less readable since there is not a clear separation between the core and the domain specific aspects.

- making a domain specific implementation of a control behavior requires a good knowledge of the implementation details of the domain. A designer may only master the semantics of the domain he uses.

hal-00262344, version 1 - 11 Mar 2008

3. Domain-polymorph component

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

Such a component is similar to a generic component since it can be used in different domains, but it is also similar to a domain-specific component because it can perform operations that are specific to its host domain.

A DPC must have the following properties:

- it must be able to detect the nature of its host domain;
- after detecting the host domain, it must be able to adapt its structure to what the host domain expects (class, communication ports);
- it must be able to use the communication and control primitives of the host domain;
- it must implement the communication and control primitives of its core behavior.

3.1. Toward domain-polymorph components

After defining what a domain-polymorph component is, we must now tell how DPCs are designed. Our goals are:

- a DPC must be usable in all domains, even in domains that didn't exist at the time of its design;
- the design of the core behavior of a DPC must require a knowledge of the model used to define it only.
- when the semantics of the core and the semantics of the host domain can be adapted in several ways, the designer must be able to choose because this choice is part of the design of the system.

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

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

In the following, we present the three successive steps in the evolution from domain specific components to domain-polymorph components. Each step is less powerful but has less overhead than the step that follows it and can therefore be chosen when the gain in adaptability is not worth the increased overhead.

3.2. The Modular Domain-Specific Component

The principle of the separation of concerns states that a given problem involves different kinds of concerns, which should be identified and separated to achieve the required engineering quality factors [3].

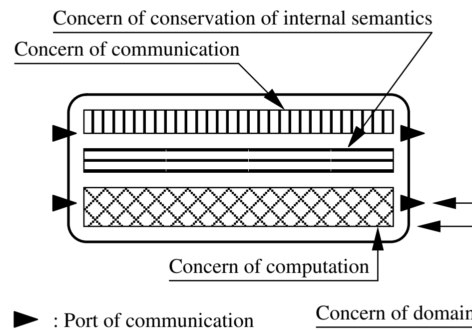

Figure 2. Concerns in a DS component

By separating concerns, we distinguish two principal concerns in any DSC that represents an entity having a semantics that must be preserved (see figure 2):

- Functional (basic) concern: what the DSC does
- Non-functional concerns: the concern of communication corresponds to communication operations with other components, the concern of domain corresponds to the structure (class and ports) of the domain-specific component and the concern of conservation of the internal semantics corresponds to the operations of conservation of the semantics of the core.

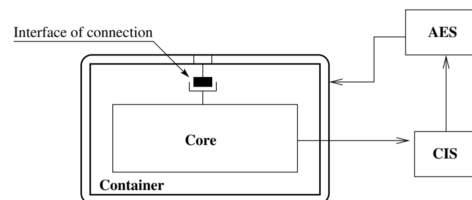

Figure 3. Architecture of a modular DSC.

By modelling each concern by a component and by interconnecting them, we obtain a new component that we call a

"modular domain-specific component" (figure 3). The components that model the concerns are detailed as follows:

- Container Component: it models the concern of domain. 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 Core component. The Container hides the domain specific aspects from the Core.
- Core Component: it models the basic concern. It has a generic interface that allows it to be encapsulated by any Container. A Core component is therefore reusable in any domain. It interacts with its external environment in a transparent way by performing the operations provided by the CIS component (see below).
- AES Component (Adaptation to the External Semantics): this component models the concern of communication and it is reusable only in its domain and for any Core component and any CIS. The AES provides the CISs with the communication operations needed to communicate in the external domain.
- CIS Component (Conservation of the Internal Semantics): this component models the concern of conservation of the internal semantic and is reusable only in its domain but for any Core component. The CIS provides an execution environment to the Core and implements it using the operations of the AES.

3.3. Flexible domain-specific component

The modular domain-specific component is a step toward the reusability of a core behavior in several domains, but it must be built specifically at compile time. To resolve this problem, i.e. to build the modular domain-specific component dynamically, we proceeded as following:

1. since the Container may accept any Core, we use a Factory to instantiate the Core from its name at runtime.
2. since the Core may use any semantics, we use a policy of selection to associate the right CIS to the Core and the AES at runtime.

The result is a "flexible domain-specific" component, as shown on figure 4 where the solid lines show compile-time links and the dotted lines show run-time links.

3.4. Domain-polymorph component

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

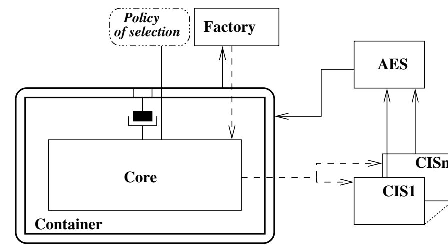

Figure 4. Flexible DSC.

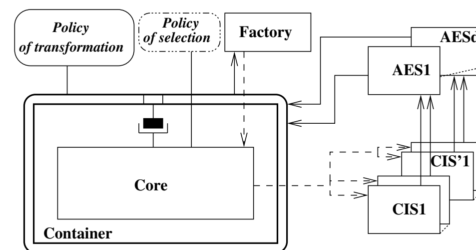

Figure 5. Domain-polymorph component.

To achieve real domain-polymorphism, we add a policy of transformation to the Container (see figure 5). This policy, firstly identifies the domain in which the DPC is placed and then transforms the Container to allow it to have the structure and all the features required by the identified domain. Secondly, this policy selects and instantiates the right AES, and links it to the Container. Then, when any DPC is used in a domain, 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 domain.

4. Internals of domain-polymorph components

After the assembly phase, the DPC is completely built — this is also true for the flexible domain specific component — and is composed of a Container, an AES which corresponds to the host domain, a CIS that corresponds to the internal semantics of the core, and the Core which implements the behavior of the component. We have now to explain how these components work together and to determine the necessary conditions for a DPC to behave properly.

hal-00262344, version 1 - 11 Mar 2008

hal-00262344, version 1 - 11 Mar 2008

4.1. Operations

A domain-polymorph component performs two kinds of operations: control operations and data-transfer operations, as shown on figure 6. A communication operation in the Core triggers a communication operation in the CIS, which is interpreted in the semantics of the host domain by a communication operation on a port of the Container in the AES.

OCS : Operations of Conservation of the Semantics.

Figure 6. Control and communication.

4.2. Functioning Conditions

There are two categories of conditions that must hold to ensure the correct functioning of a DPC: at the level of the core and at the level of its execution machine, the CIS.
Conditions on the Core: the core must be implemented in such a way that the CIS is able to preserve its semantics and to translate it to the semantics of the host domain in collaboration with the AES.

For instance, it may happen that the Core does not produce any data in reaction to some input. However, this absence of production may need to be translated into the production of some data on the ports of the Container in the host domain. For this, the CIS needs to be informed of the progress and status of the reaction of the Core through the use of operations of conservation of the semantics (OCS). These operations allow the CIS to interpret the behavior of the Core in a standard form and to cooperate with the AES to translate it in the host domain.
Conditions on the CIS : to use a core component in two domains A and B, we need two different CIS since the CIS depends both on the model used by the core and on the host domain. However, the core is written only once, so every CIS must provide the same set of operations of conservation of the semantics. This set of operations defines the programming interface of core components. Any CIS must implement these operations (even in a trivial manner if the operation is a no-op for this domain).

4.3. Degrees of polymorphism

The domain-polymorph component is the form of component that has the maximal 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 maximal degree of polymorphism is therefore only achieved when the programming language offers enough possibilities to avoid any manual or automatic external activity to build and assemble the components of a DPC.

5. Optimization of the DPC

The current architecture of the domain-polymorph components allows to adapt the semantics of a core component to any host domain in a fully automatic way. However, the fact that the CIS depends on both the semantics of the core component and the semantics of the host domain makes the number of types of CIS grow as the square of the number of supported domains: if n domains are supported, we need n types of AES, and since there are n^2 pairs of domains, we need n^2 types of CIS.

In order to reduce the necessary number of CIS, we make them independent of the semantics of the external domain. This leads us to the following optimized architecture of DPCs, as shown on figure 7.

→▷ : Static link between the Core and its corresponding ICAS.

Figure 7. Optimal DPC architecture

In this optimized architecture — that we won't describe in details because it is still a work in progress — we define a Conservative Adapter of the Semantics (CAS) for each domain. A CAS can be used both to provide an execution environment to a core and to adapt the semantics of a core

hal-00262344, version 1 - 11 Mar 2008

to a host domain. In the first case, the CAS is called an Internal CAS because it adapts to the internal semantics of the DPC; in the second case, it is called an External CAS because it adapts to the semantics of the external domain.

A Border Component (BC) is in charge of all the mappings between semantics, and a Connector of Semantics (CS) connects ECASs and ICASs and allows data and control to flow through domains.

6. Implementation in Ptolemy II

We chose Ptolemy II [2] to implement our concepts, because this platform supports many domains and is open to the creation of new domains, its architecture is the most generic we have found. Since Ptolemy II is programmed in Java which has static types, we could not use the complete DPC approach without modifying the kernel of Ptolemy II, so we used the Flexible Domain Specific Component approach. With our implementation, it is now possible to use a core behavior in any domain by dragging a Container from a list of domain specific Containers and dropping it onto the system being designed. Then, by setting a parameter of the container, a core is associated to it, and it performs the assembly phase to reach its integral form as shown on figure 8.

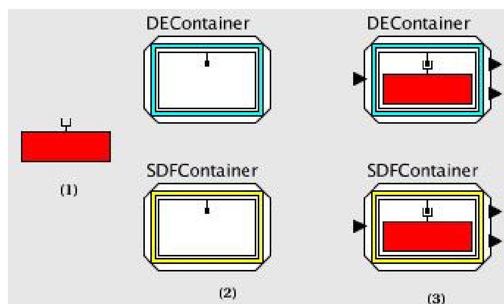

Figure 8. (1)Core; (2)Containers; (3)Core in Containers

7. Conclusion

In this article, we showed the down sides of the use of domain-specific components for representing behaviors that are usable in several domains. We introduced the notion of domain-polymorph components to adapt the semantics of a core behavior to the semantics of its host domain. We presented several steps in the design of such components, achieving different degrees of polymorphism, that can be reached according to the availability of some mechanisms in the programming language used for the implementation.

Domain-polymorph components allow the reuse of the same behaviors in several domains. Contrary to generic components, that have a generic behavior which is interpreted by the host domain, domain-polymorph components actively adapt their semantics to the host domain. 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 and not hard-coded in the development platform.

An implementation of domain-polymorph actors, limited to the "flexible DSC " level, has been done in Ptolemy II and is used to experiment on the relation between domain-polymorphism and non-hierarchical heterogeneity, an other research focus of our team. In [7], we introduced the notion of Heterogeneous Interface Component, a component that works at the boundary of several domains, and DPCs may be a good implementation for heterogeneous interface components (HICs), with containers as the projections of the HIC on the different domains, all embedding the same core in different semantics.

References

[1] J. Liu. Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems. Ph.D. thesis, Technical Memorandum UCB/ERL M01/41, University of California, Berkeley, December, 2001.

[2] http://ptolemy.eecs.berkeley.edu.

[3] M. Akšit, B. Tekinerdoǧan, L. Bergmans, The Six Concerns for Separation of Concerns. Proc. of the ECOOP Workshop on Advanced Separation of Concerns, 2001.

[4] J. Liu, J. Eker, X. Liu, J. Reekie, E.A. 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.

[5] E.A. Lee, S. Neuendorffer, M.J. Wirthlin, Actor-oriented design of embedded hardware and software systems, Invited paper, Journal of Circuits, Systems, and Computers, Version 2, November, 2002

[6] E. A. Lee and D. G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE, vol. 75, no. 9, pp. 1235 to 1245, Sept. 1987.

[7] M. Mbobi, F. Boulanger, M. Feredj Non-hierarchical heterogeneity, International Conference on Computer, Communication and Control Technologies, July-August 2003, Orlando, Florida, USA International Institute of Information and Systemics Volume III, ISBN 980-6560-05-1, pp 430 to 435.