DYNACOMM: THE EXTENSION OF COMMUNITY TO SUPPORT DYNAMIC RECONFIGURATION

BY: XIANG LING

B.ENG.

DYNACOMM: THE EXTENSION OF COMMUNITY

TO SUPPORT DYNAMIC RECONFIGURATION

By
XIANG LING, B.ENG.

A Thesis
Submitted to the School of Graduate Studies
In partial fulfillment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2007) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
DynaComm: The Extension of CommUnity to Support Dynamic Reconfiguration

AUTHOR: Xiang Ling, B.ENG. (Anhui University)

SUPERVISOR: Dr. Tom Maibaum

NUMBER OF PAGES: viii, 146

Abstract

Architecture Description Languages were developed to support the abstract level of software structuring that is the subject matter of software architecture. CommUnity is an ADL built on co-ordination principles and a categorical framework to support the composition of specifications of components to form the system's specification. However, an important problem of CommUnity is the lack of support for specifying the system's architectural changes in both the set of components and the connections between them.

This thesis presents DynaComm, an extension of CommUnity to support hierarchical design and dynamic reconfiguration of component based systems. Several new language constructs are introduced into DynaComm: subsystems are coarse grained components which are considered as the basic unit for the construction of systems, connectors encapsulate a component interaction pattern that can organize the possibly complicated interactions between the components of a subsystem. We also propose the idea of interface manager to solve the problem of incorrectly synchronized actions in CommUnity, and the concept of population manager to manage the live instances of components in a subsystem, through which we can model potentially complicated dynamic reconfigurations in a system.

To use the semantics of CommUnity in defining the semantics of DynaComm, a "normalization" technique is introduced to transform the parameterized (indexed) actions into "normal" actions of CommUnity and reduce the specification of connectors and subsystems to flat CommUnity designs, so that we can derive the system's semantics in a certain state.

Two illustrative examples, fault-tolerant dynamic client-server and vending machine systems, are also given to show the usage of DynaComm in modeling complicated and dynamic systems.

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Tom Maibaum, for his guidance, support and faith throughout my research of this thesis. Although usually very busy, he was always willing to answer my questions, discuss the problems and difficulties I encountered, and suggest directions or materials to explore. I am really grateful that he agreed to supervise me.

I wish to thank Dr. William Farmer, Dr. Jacques Carette and Dr. Wolfram Kahl who gave me useful advice and feedback on my thesis work. I would also like to thank all the members of the Software Quality Research Laboratory.

Special thanks to my wife, my daughter and my parents, for their unflagging love, encouragement and support, which provides me the best "Community" for my work.

Last, I would like to acknowledge the financial support from Ontario Graduate Scholarship of Science and Technology (OGSST).

Table of Contents

List of Figures

Chapter 1

Introduction

This thesis is mainly concerned with the design of an Architecture Description Language (ADL), DynaComm, to support dynamic reconfiguration and hierarchical organization of component-based systems. This chapter provides an introduction to this thesis, including the motivation, some basic concepts, the origin of ADLs, contributions and thesis outline.

1.1 Motivation

According to [28], the concept of ADL is proposed to provide formal modeling notations, analysis and development tools to support architecture-based development, which focuses on the system's high-level structure rather than the implementation details of any specific modules. CommUnity is an ADL built on a categorical framework to support the composition of specifications of components to form the system's specification, and serves as a basis for the design of the DynaComm ADL. The mechanisms for composing specifications consist of the notion of regulative superposition morphisms between the component specifications and the use of universal constructions, such as the pushout and colimit operations of category theory [15]. Through the separation of the concepts of action blocking and action progress and the use of underspecification for defining the effect of multiple assignments, CommUnity also provides a well-defined notion of refinement, which is combined with the composition mechanisms in a nice way (as we will show in section 2.4.4.3). Moreover, CommUnity has some tool support, the CommUnity Workbench [32], which provides an integrated environment for editing designs, defining the interconnection between designs and performing the colimit generation [15].

However, CommUnity does not provide a coarse-grained construction unit within the language, which can contain subcomponents, such that system specification will be organized in a hierarchical way. The only language construct of CommUnity is the notion of *design* and the system's architecture built from designs is a flat configuration diagram without hierarchical structure. We will introduce the concept of subsystems into DynaComm (see chapter 3) to support the hierarchical organization of the system's architecture.

Another problem of CommUnity, which is the main focus of this thesis, is the lack of support for specifying a system's structural evolutions during run time, because CommUnity itself does not provide the mechanisms for talking about structural change in the configuration diagram of the specified system. Although graph grammars [14] have been introduced to the CommUnity language, to define the reconfiguration operations for creating or deleting components and changing the connections between components so that the change of configuration diagrams during dynamic reconfiguration can be implemented by graph rewriting rules, it is not amenable to hierarchical structuring because it uses a meta-language for the definition of these operations. As a result, it will be difficult to reason about the properties of a system specified by these two different languages. The DynaComm ADL will incorporate dynamic reconfiguration operations into the language and some temporal logic based formalisms can be related to DynaComm specifications to support the verification of the properties of specified systems, including reasoning about reconfiguration (see future work in chapter 6).

1.2 Some basic concepts

As we mentioned in the above section, the main intention for our work on DynaComm is to describe the dynamic reconfiguration of component-based systems. Therefore, some related concepts need to be clarified to provide a context for the DynaComm ADL.

- Component-based systems

The term component is a common concept provided by most of the popular ADLs, such as Wright [11], Darwin [26], Acme [19] and CommUnity [15]. According to [31], "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". The key notion here is that the explicit dependencies must be declared in component specifications, which distinguishes components from the concept of classes in object-oriented languages, where the dependencies cannot be enforced. Components are coarse grained in the sense that they may contain subcomponents within themselves, and components can be developed by a variety of coding paradigms. Therefore, component-based systems have become popular in the software development field because

components are easy to understand, reuse, replace and deploy.

The key difference between component-based systems and traditional systems is the incorporation of an underlying framework to support the interactions between the components, which is called a component model. The entities developed within component-based systems must conform to this model, in which they are placed in containers and connected with other components. Although the methodology of decomposing a system into smaller units with a loose coupling among them is also used by traditional systems, it will not be feasible to check the system's compliance with a component model due to the lack of the underlying framework built into these systems.

To conform to the concept of designs in CommUnity, components are treated as the atomic construction units in the DynaComm ADL, so that substructures are not allowed to be included within components. Instead the notion of subsystem is proposed as the appropriate definition of component discussed above. A subsystem may contain (atomic) components or other subsystems as constituents, thus providing the coarse-grained building blocks to construct the systems in a hierarchical way and support the development of component-based systems with the DynaComm ADL. We also have well-defined relationship between the components, namely morphisms, inherited from CommUnity, to define the interaction patterns of the components and guide their composition.

- Software Architecture (SA)

"Software Architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on their patterns" [30]. The aim of SA is to address the gross decomposition and organization of systems, in which component interactions are identified as being first-class design entities, so that the dependencies between the system's components can be captured and it supports the reuse of individual components as well as the interaction patterns. Meanwhile, presenting a system at the architectural level reveals the overall structure of the system and makes it easier to understand and extend.

It is argued in [15] that the complexity of system construction mainly arises from the interconnections between components that regulate how they will interact, because the system's global properties emerging from these interactions are difficult to predict. To control the complexity of software systems resulting from the interactions between the components, Software Architecture uses

components and connectors to model complex systems [21]. The architecture of a system is a directed graph, where components are represented by nodes and connectors are the edges to interconnect the components. By explicitly modeling the interactions among components with the notion of connectors, the computational parts of the system are separated from the structure of a system (which focuses on the interactions), such that the system's complexity arising from the interactions between components can be addressed and controlled.

ADLs provide a formal way to describe the software architecture of a system in terms of components and component interactions, and enable the reasoning about the structural and behavioral properties of the system. The DynaComm ADL has a well-defined concept of connector, through which we are able to model the complicated and dynamic interaction patterns among components and subsystems (which will be illustrated by the fault-tolerant dynamic client-server example in chapter 3), thus supporting the component-based software architecture principles.

- Dynamic reconfiguration

Since one important objective of the DynaComm ADL is to incorporate dynamic reconfiguration mechanisms into the language, we need to clarify the notion of dynamic reconfiguration in the context of Software Architecture. From [13], the behavior of modifying the system's architecture during its execution (run time) is commonly known as run-time evolution, dynamism, or dynamic software architecture. We use the term dynamic reconfiguration to refer to the dynamism in a system's architecture and define it as the description of a system's structural evolution as execution progresses, where the composition of interacting components changes during the transition of the system from state to state. Our definition of dynamic reconfiguration reflects the operational view of the system's architecture, where each computation step of the system may change its state or the configuration structure of the system.

For example, let us consider the specification of a system's architecture in CommUnity, which is represented by a configuration diagram, where designs (components) are interconnected by "middle" designs (cables) through regulative superposition morphisms (see chapter 2). Because CommUnity does not support the specification of actions to change the interconnections between the components within the language, the configuration diagram of the system is fixed in CommUnity once the specification of the system's architecture has been accomplished. However, the configuration diagram of the system might be

changed as execution progresses, e.g. in a dynamic client-server architecture, a new client may request to be connected to the server, the communication protocol between the clients and the server can be changed, or a new server needs to be created when the number of connected clients exceeds a certain limit. Therefore, dynamic reconfiguration operations should be introduced into CommUnity to specify the change of the system's configuration diagram under the situations discussed above, which will be shown in our design of the DynaComm ADL.

On the other hand, from the view of architectural style, a system's style can be defined as a set of architectural elements and the rules governing how they are composed. Generally these rules are defined by some constraints to restrict the composition of these architectural elements, and a family of architectural instances can derived from an architectural style. For example, in Wright a style must be defined before specifying a system's architecture, which is declared by a set of component and connector types and the constraints on the configurations to which every instance of this style must conform. Assuming that we have declared a style for the above dynamic client-server example, which represents the set of architectural instances corresponding to those dynamic changes to the system's configuration, we may conclude that there is no change to the architectural style during system run time and dynamic reconfigurations occur at a lower level than the architectural style when we hold an operational view of the system's architecture. However, we still argue that dynamic reconfigurations should be considered as an important issue in Software Architecture to deal with the structural evolutions during system's execution. In addition, if we want to reason about some temporal properties of the system's architecture, the specification of dynamic reconfiguration is necessary for describing the system's structural changes during the run time in order to verify if these properties will hold during these changes.

1.3 The origin of ADLs

The notion of modularization was raised in the software development field decades ago when the complexity of software systems was increasing. The modularization methodology enables the developers to divide the system's functionality into separate modules, such that the complexity of each individual module can be managed. It also promotes the structured design principles and the reusability of the software system by the reuse of existing modules. At the level of architecture modeling, the consequences of the architectural design

decisions can be evaluated prior to the implementation stage by following the modularization principle [4].

Object oriented (OO) techniques originated from the idea of modularization, which encapsulate both data and behaviors into the powerful concept of objects and provide the mechanisms of class inheritance and polymorphism. OO paradigms are very suitable for modeling real world entities and processes and organizing software systems in a natural, elegant and clear way. On the other hand, Software Architectures [9][20] emerged as a new branch of software engineering. SA promotes modularization concepts at a higher level of abstraction, namely components, and introduces the notion of connectors as a second modularization concept for structurally representing and decomposing systems in more abstract ways than OO programming models of systems [4]. Meanwhile, the concept of association classes in UML models has some relationship to connectors.

Then ADLs were invented as the specification language to provide higher-level structural descriptions of systems (as introduced in section 1.1), since describing systems at the programming language module level only provided a low-level view of interconnections between components, and did not separate the concerns between computational mechanisms and architectural level issues (such as the interactions among the components) [21]. Meanwhile, another issue was raised for ADLs, namely dynamic reconfiguration, which requires the modification of a system's architecture at run time. Although some transformation rules and operations have been introduced into some ADLs (e.g. Dynamic Wright) to enable architectures to change dynamically, the verification of a system's properties is often performed informally in some meta-languages [17]. In particular, the previous work on the dynamism of CommUnity uses two different languages to describe the individual components and the dynamic changes of the configuration diagrams, which makes the reasoning about the properties of the "mixed" architectural specifications difficult. This problem has motivated our work to build the dynamic reconfiguration mechanisms into the DynaComm ADL.

1.4 Contributions and thesis outline

This thesis contributes both to the successful extension of the CommUnity ADL to support dynamic reconfiguration, and the illustration of DynaComm's suitability in specifying reactive and dynamic systems through the investigation of the design principles implied by different notions of morphisms between

designs, as well as their relationships and combinations, with two case studies.

The following is a list of specific contributions:

The DynaComm language:

- Clarified the language constructs of DynaComm, such as component, connector and subsystem, to support the hierarchical organization of system specification.

- Introduced indexed actions into DynaComm to specify population management and reconfiguration actions, as well as the interface manager to reduce the length of specification.

The methodological aspect of the language:

- Defined the concept of a population manager on the subsystem level to manage the live instances of components, subsystems and connectors of the subsystem.

- Provided the notion of interface manager to overcome the problem of incorrectly synchronized actions in CommUnity when using the interaction mechanisms in a naive way, and designed it as a regulator to be applied to the target component to support the incremental design principle.

- Demonstrated the usefulness of regulative superpositions as the structuring mechanism to build systems in an incremental way.

- Proved that in a well-formed configuration diagram, regulative superpositions can be combined with refinement morphisms or extension morphisms to form new regulative superpositions, thus enabling the use of these structuring mechanisms in building designs.

Regarding the semantics of DynaComm:

- Designed a normalization technique to eliminate the indices of actions in DynaComm and transform the specifications into CommUnity-like, flat designs.

- Provided the transformation procedure to derive a dynamic system's static semantics at a certain state.

Case studies:

- Specified fault-tolerant dynamic client-server and vending machine systems as the proof of concept for the DynaComm ADL, and explored the design principles supported (and enforced) by this language.

The remainder of this thesis is divided into five chapters.

Firstly, several representative ADLs with support for dynamic reconfiguration are surveyed in chapter 2, with a special focus on the syntax and semantics of CommUnity.

Next, chapter 3 defines the basic language constructs of DynaComm and demonstrates its suitability for specifying dynamic reconfiguration mechanisms in a reasonably big system containing complicated interactions.

To define the semantics of the DynaComm language, in chapter 4, the normalization technique is introduced to eliminate the indices of actions and transform complex DynaComm specifications into CommUnity-like designs in a systematic way.

Then, chapter 5 presents the approach of combining regulative superpositions and extension morphisms to add new behavior into the existing systems and a vending machine system is specified to illustrate the applicability and necessity of this approach.

Finally, chapter 6 briefly reviews this thesis, summarises contributions and suggests future work.

Chapter 2

Background

Software architecture research is directed at presenting the high-level decomposition and organization of systems, where component interactions are incorporated into the notion of connectors and identified as first-class design entities. Architecture description languages (ADLs) have been proposed to provide formal modeling notations, analysis and development tools to support architecture-based development, which focuses on the system's high-level structure rather than the implementation details of any specific modules [28].

There has been some work in surveying ADLs providing broad comparisons. The survey by Medvidovic, N., and Taylor, R.N., in [28] compared ADLs on their ability to model components, connectors and configurations as well as tool support for analysis and refinement. The survey by Bradbury, J.S., Cordy, J.R., Dingel, J., and Wermelinger, M., in [13] focused on the characteristics of different ADLs to support self-managing architectures, which not only implements the change internally but also initiates, selects and assesses the change itself without the assistance of an external user. In this thesis, we are interested in the ADLs with support for dynamic software architectures. Therefore, Dynamic Wright [11], Darwin [26] and Dynamic Acme [19][33] are surveyed on their language constructs, associated styles of specification and mechanisms to achieve dynamic reconfiguration. We consider these languages as a set of representatives of approaches to incorporate dynamism into such languages.

A detailed review of CommUnity [15] and its semantics are given, and, in particular, we rehearse the idea that the notion of superposition can be formalized as a morphism between designs in CommUnity. The concept of superposition is defined as a structure preserving transformation on designs through the extension of their state space and control activity while preserving their properties [15]. So, a regulative superposition morphism is proposed in CommUnity as a means of augmenting an existing component by superposing a regulator over it while preserving its functionality, thus supporting a layered approach to system design. In addition, several different kinds of morphisms (other than regulative superposition morphisms) between designs as well as their relationships are also investigated to explain the language's well-founded support for compositionality, reusability, enforcement of design principles and

refinement and traceability.

To illustrate the concepts, principles and styles of specifications adopted by these surveyed ADLs in specifying a system's architecture, an example of Producer-Consumer system is used through this chapter. The system's requirement is described as follows. We want to describe a system consisting of one producer and one consumer. The producer will produce items regularly and store them into a local queue, from which an item is chosen and sent to the consumer; then it will wait for a reply from the consumer. If the consumer is not busy, this item can be consumed and an acknowledgement will be sent back to the producer, so that it can get another item from the items queue and send it to the consumer again.

2.1 Dynamic Wright

Dynamic Wright is based on the Wright ADL [10], which originally did not support dynamic reconfiguration. A system is built from components and interrelated by means of connectors. In this section we will review the basic concepts of component, connector and configuration of Wright and use these constructs to specify a Producer-Consumer system. Then the dynamic reconfiguration mechanisms will be added to the system, by introducing the extensions of Wright proposed in Dynamic Wright.

Components are the computational and data storage units of systems in Wright which contain *ports* and *computations.* Ports declare the interface of a component, which represent points of interaction for this component between its computational part and the environment. The behavior of components is defined by their corresponding computations in terms of a sequence of events. We can view the behavior of a component as a state machine, in which the states are given by the valid event sequences that can be generated from the specification of ports and computations, and the alphabet of possible events defines the transitions.

Components are not allowed to interact with each other directly in Wright. Instead they must attach to the interaction points (roles) of connectors through their ports. Glue is the computational part of connectors, which specifies the coordination of the computations of interconnected components to form a larger computation. By separating interaction from computation, connectors enable the interconnected components to be completely independent. A system's architecture can be specified as a collection of components combined via connectors, and the families of architectures with common characteristics, which

are sometimes called an architectural style, is defined by imposing constraints on components, connectors and configurations as *style* in Wright.

Now we will specify the Producer-Consumer system by declaring a style in Wright, using components Producer and Consumer, connector Cable to interconnect the components and the constraints as defined below:

Style Producer-Consumer
Component Producer
Port p = send → ack → p || !
Computation = internalCompute → p.send → p.ack → **Computation** || !

Component Consumer
Port p = extract → reply → p [] !
Computation = p.extract → internalCompute → p.reply → **Computation** [] !

Connector Cable
Role s = send → ack → s || !
Role c = extract → reply → c [] !
// Interconnect two roles
Glue = s.send → c.extract → **Glue**
[] c.reply → s.ack → **Glue**
[] !

Constraints
$\forall p \in$ Component: TypeProducer(p) $\Rightarrow$ ($\exists c \in$ Component: TypeConsumer(c) $\wedge$ connected(p, c))
EndStyle

The protocol of Producer's port p specifies that the producer initiates event send, then waits for the reply until it observes event ack and repeats this process, or it can terminate by !. In the computation part of Producer, we can see its behavior: the producer iteratively generates items by internal computation, makes send events on port p and waits for event ack on the same port, then it will repeat this sequence or terminate by !. The use of internal choice || in Producer indicates that the producer can decide whether to send the item or terminate. In contrast, the use of external choice [] in the Consumer component

means that the choice is left to the environment, so that the consumer will consume the items sent to it continually unless it is "forced" to stop.

Connector Cable is defined to interconnect the producer and consumer, where its roles s and c have the same protocol as port p of Producer and port p of Consumer, respectively. The glue specifies the interaction protocol: it observes the send event of producer and transmits it to consumer's event extract; the reply event of the consumer is also observed and triggers the ack event of producer.

A static Producer-Consumer system can be specified as a configuration in Wright by using the Producer-Consumer style, where the components and connector types have been defined and are available to be put into the configuration. The attachment section specifies the connections between the component instances and the roles of the connector instance. Specifically, the port p of the producer instance P fills the role s of connector instance B, while the port p of the consumer instance is attached to the role c of the connector.

Configuration Static-Producer-Consumer
Style Producer-Consumer
Instances P: Producer; C: Consumer; B: Cable
Attachments P.p as B.s; C.p as B.c
EndConfiguration

To specify the dynamism in the Producer-Consumer system, we will introduce the extension of Wright in [11], Dynamic Wright, which supports the modification of run time structure of the architecture. The key idea is to separate the behavior related to the regular computations of components from the behavior that controls the reconfiguration process. The concept of *configuror* is introduced as a separate module to include the events that trigger reconfiguration actions. The configuror can create and delete instances by using *new* and *del*, and can connect or disconnect components from connectors with *attach* and *detach*.

Now we consider a dynamic Producer-Consumer system consisting of one producer and two consumers. Initially, the producer is connected to one consumer. If the consumer is busy with some internal computations, the reconfiguration action will change the system's configuration by switching the producer's connection to the free consumer. First, we need to modify the style specification of the original Producer-Consumer system, which takes the status change of consumer into consideration and then modify the glue definition as

well.

Style Dynamic-Producer-Consumer
Component Producer
Port p = send → ack → p || !
Computation = internalCompute → p.send → p.ack → **Computation** || !

Component Consumer
Port p = (extract → reply → p || control.busy → (![] control.free → p)) [] !
Computation = (p.extract → internalCompute → p.reply → **Computation** || control.busy → (! [] control.free → **Computation**)) [] !

Connector FTCable
Role s = send → ack → s || !
Role c = (extract → reply → c || control.changeOK → c) [] !
// Interconnect two roles
Glue = s.send → c.extract → **Glue**
[] c.reply → s.ack → **Glue**
[] !
// reconfiguration
[] control.changeOK → **Glue**

Constraints
$\forall p \in$ Component: TypeProducer(p) $\Rightarrow$ ($\exists c \in$ Component: TypeConsumer(c) $\wedge$ connected(p, c))
EndStyle

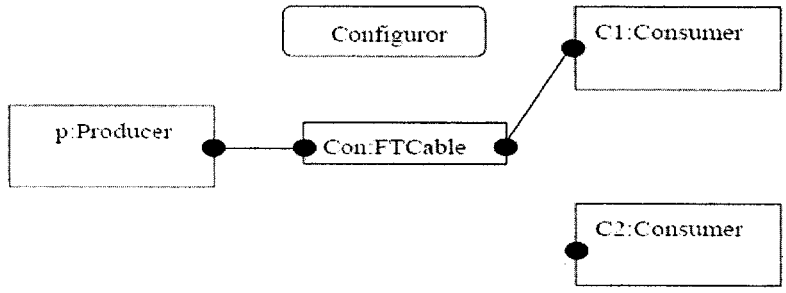
Then we specify the configuror as below, which takes care of the switch of the producer's connection to a live consumer, whenever the current consumer is busy.

Configuror Producer-And-Two-Consumers
Style Dynamic-Producer-Consumer
new.P : Producer
→ new.C1: Consumer
→ new.C2: Consumer

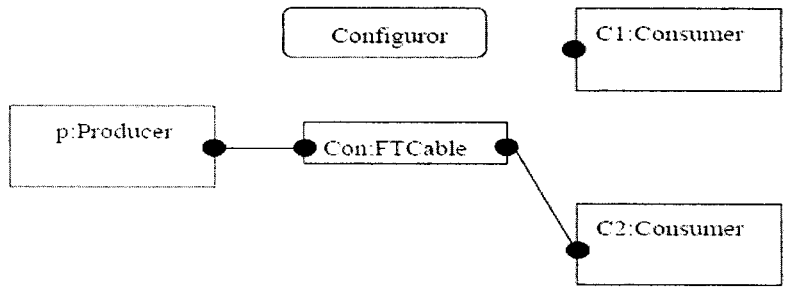
→ new.Con: FTCable
→ attach.P.p.to.Con.s
→ attach.C1.p.to.Con.c
→ WairForChangeConsumer2
where
WaitForChangeConsumer1 = (C2.control.busy → C1.control.free →
Con.control.changeOK →
Style Dynamic-Producer-Consumer
detach.C2.p.from.Con.c
attach.C1.p.to.Con.c
WaitForChangeConsumer2) [] !

WaitForChangeConsumer2 = (C1.control.busy → C2.control.free →
Con.control.changeOK →
Style Dynamic-Producer-Consumer
detach.C1.p.from.Con.c
attach.C2.p.to.Con.c
WaitForChangeConsumer1) [] !

In the above specification, the configuror first creates one instance of Producer, two instances of Consumer and one instance of connector FTCable, and connects the producer instance with the first consumer instance C1 through the connector. When the control event of C1 indicates busy and C2 is free, C1 is detached from role c of the connector and C2 is attached to role c. Then if the system detects that C2 is busy, the connection with the producer will be switched back to C1. A diagram to illustrate the mechanism of the configuror to achieve dynamic reconfigurations is as follows:



Producer instance p is connected to the first Consumer instance C1



Producer instance p is connected to the second Consumer instance C2

Figure 2. 1 Configuror Producer-And-Two-Consumers

By introducing the concept of configuror, the reconfiguration actions (such as attach and detach) corresponding to the events (which trigger the reconfiguration) can be described explicitly in Dynamic Wright. The imperative nature of the configuror specification imposes ordering requirements on the reconfiguration operations, and multiple reconfiguration events can be declared in the configuror, where each event corresponds to a sequence of reconfiguration actions to describe the change to the system's configuration when this event occurs. However, the specification of the configuror will be tedious if we want to have a dynamic number of consumers in the above example, because for each consumer (connected to the producer) in the system, a sequence of reconfiguration actions should be specified in response to the control event that it is busy.

2.2 Darwin

Darwin is a declarative language which is intended to be a general purpose notation for specifying the structure of distributed systems composed from diverse components using diverse interaction mechanisms [26]. It is different from an operational language in the sense that it focuses on the description of the system's structure, which separates the concerns of computation and interaction. In contrast, we must specify the interaction between components by the protocols of ports and connectors in Wright, and describe the concrete computation mechanisms of components in CommUnity. Darwin has built-in mechanisms to support dynamic reconfiguration, which includes the change of bindings between components and the creation and deletion of component instances during run time.

Component is the main computational unit of Darwin, which contains ports, instances of other components and bindings between ports. Ports signify the services that one component requires to interact with other components (indicated by *require* keyword) or provides to other components for them to interact with it (indicated by *provide* keyword). There is a type associated with a port, which specifies the kind of service required or provided by this port. To illustrate how components are defined in Darwin, we can look at the Producer-Consumer example again, and declare the components Producer and Consumer as follows:

```
interface Message {}
interface Item {}

component Producer {
    require ack: Message;
    provides send: Item;
}

component Consumer {
    require extract: Item;
    provides reply: Message;
}
```

Component Consumer has a required service called extract, which is of type Item and corresponds to the item to be consumed by the Consumer. It also provides a reply service to indicate that it is ready to consume one item. On the

other hand, component Producer requires a service ack of type Message to make sure that it can send out the produced item, and provides the service called send corresponding to the item to be sent by the Producer. The types of services Message and Item should be declared, and we skip the detailed description here. Darwin also has a graphical notation for the above specifications as depicted in Figure 2.2, where the empty circles represent services required by a component and filled circles represent services provided by a component.

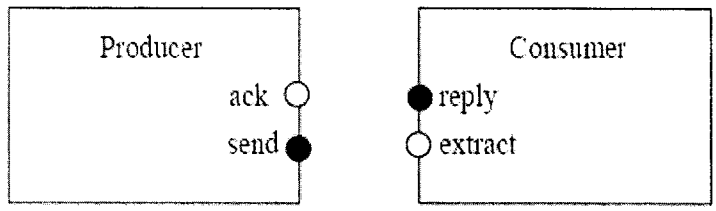


Figure 2. 2 Graphical Representation of Darwin Components

To enable the interaction between components, we need to introduce the concept of bindings, which defines a one-to-many relation between provided and required ports. Through a binding, the service required by a component is associated with the service provided by another component. Since the relation is one-to-many, there may be many required ports bound to a provided port, but a required port can have at most one provided port bound to it. Composite components can be constructed from the bindings of basic components, which allows for a hierarchical composition of systems. (However, composite components are not supported in Wright, thus limiting the hierarchical organization of a system's architecture.) For example, we can connect component Producer and Consumer to obtain a new component Producer_Consumer as follows:

```
component Producer_Consumer {
    inst
        p: Producer;
        c: Consumer;
    bind {
        p.send -- c.extract;
        p.ack -- c.reply;
```

```
    }
}
```

The graphical representation is shown in Figure 2.3:

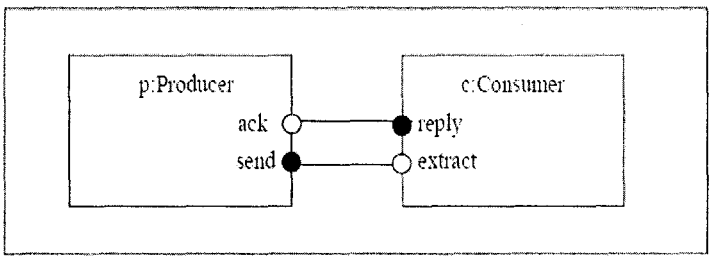


Figure 2. 3 Composite component Producer_Consumer

Darwin has the built-in mechanisms to support dynamic structures of the evolving system. Basically it provides two techniques to capture the system's structural dynamism: lazy instantiation and direct dynamic instantiation. The meaning of lazy instantiation is that the component providing a service is not instantiated until a user of that service requires it. For instance, in component Producer_Consumer, we can put the keyword *dyn* before the Consumer so that the Consumer instance c will be actually created when its reply service is required by the Producer.

The other mechanism for dynamic reconfiguration is direct dynamic instantiation [26], which enables the system's structures to evolve in a flexible way. We will illustrate this mechanism by modifying the Produce_Consumer component to allow the creation of new consumers through a new service of the updated composite component Dynamic_Producer_Consumer.

```
component Dynamic_Producer_Consumer {
    provide new<dyn>;
    inst
        p: Producer;
    bind {
        new -- dyn Consumer;
        p.send -- Consumer.extract;
```

```
        p.ack -- Consumer.reply;
    }
}
```

The composite component Dynamic_Producer_Consumer provides the service new, which is bound to the component type Consumer prefixed with the keyword **dyn** and this port will create Consumer instances dynamically as a result of component Dynamic_Producer_Consumer's computation. Notice that we only declare an instance of component Producer in the above specification, because the Consumer instances created through direct dynamic instantiation are anonymous and cannot be referred to in the Darwin program. As a result, the bindings are from Producer instance p to component type Consumer and not to any instance of it. We can create a dynamic number of Consumer instances in Dynamic_Producer_Consumer and all the live instances of Consumer should be bound to the Producer instance p. The graphical notation for the composite component Dynamic_Producer_Consumer is as follows:

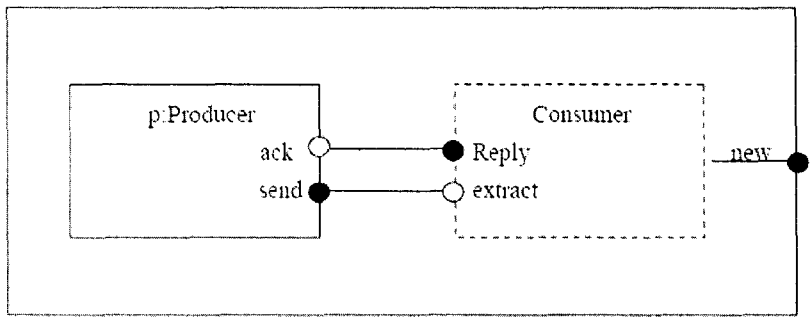


Figure 2. 4 Graphical representation of Dynamic_Producer_Consumer

Darwin's semantics is based on a process calculus, the π-calculus. Compared with Dynamic Wright, Darwin does not have the notion of connector as a language construct. If required, connectors can be modeled with components and ports along with their bindings, by defining glue and roles as components and their connections as the bindings. There are certain limitations for the possible reconfigurations in Darwin, since the two techniques introduced above for capturing dynamic structure are intended to retain the declarative notation of Darwin, but do not support general reconfiguration operations. When the system's architecture must change in response to some influence (events) during run time, some imperative constructs are required to describe the

sequence of actions for changing the system's architectural configuration, which is not supported by Darwin because the component instances created through direct dynamic instantiation are anonymous and cannot be referred to in these actions. For example, in the Dynamic Producer-Consumer system, if the current consumer (connected to the producer) is busy, we need the imperative constructs to specify the switch operation, which detaches the current consumer from the producer and attaches a free consumer to it.

2.3 Dynamic Acme

Acme is an ADL built on the experience of using other ADLs, which provides the essential elements of the architectural description for component-based systems [19]. Components are the basic computational and data storage units in Acme, and they are connected by means of connectors to interact with each other and characterize the joint behavior of component-based systems. In this section, we will briefly review the syntax of Acme, and use the language to specify a static Producer-Consumer system to give a view of how architectures of systems are described in Acme. Since there is no direct support in Acme for dynamic reconfiguration, the extension of the language, Dynamic Acme, will be introduced later to specify the dynamic Producer-Consumer system.

The interfaces of components are called *ports*, which identify the points of interaction between the components and their corresponding environments. This concept is similar to a port in Darwin, which declares the services of a component but does not specify how these services are implemented. Acme's declarative style originates from the language's intention to serve as a common representation for software architectures and permits the integration of diverse collections of independently developed architectural analysis tools [19].

Acme connectors are used to represent the interactions between components. The interface of a connector is defined by a set of *roles*, where each role specifies a participant of the interactions defined by this connector. *Systems* are a collection of interconnected components defined by graphs, in which the nodes represent components and the arcs represent connectors. We may also assign *properties* to components or connectors to specify auxiliary information about a system's architecture, such as the protocols of the interaction and the attributes of components or connectors. To put these elements together, we can specify the Producer-Consumer system in Acme as follows:

```
System Producer-Consumer = {
    Component producer = { Port send }

    Component consumer = {
            Port extract;
            Properties { busy: boolean } }

    Connector cable = {
            Role sender;
            Role receiver }

    Attachments {
            producer.send to cable.sender;
            consumer.extract to cable.receiver }
}
```

Component producer simply has one port send to deliver the produced items. The consumer component provides the port extract to get the item to be consumed, and has a property busy of the boolean type to indicate whether it is ready to work with the producer. In order to connect these two components, we define a connector cable with two roles designated sender and receiver, which provides the acknowledgement/reply communication protocol (the interaction protocol between the roles can be specified as properties of the connector and we skip the detailed description here) for the transmission of items, so that we do not specify the ack port in producer and the reply port in consumer. Then we can declare the attachments of components and connectors, with the producer's send port connected to the cable's sender port, and the consumer's extract port connected to the cable's receiver port. A graphical representation of the system is as follows:

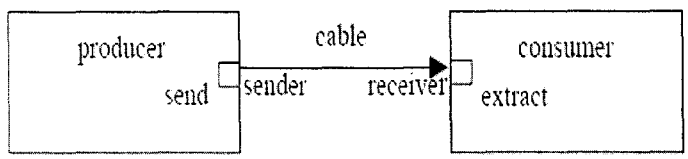


Figure 2. 5 Producer-Consumer system in Acme

We may also use component types or connector types to define them, which allows for the definition of multiple instances of the same type. To support the hierarchical organization of system specifications, Acme provides the concept of *representation* to allow any components or connectors to be represented by detailed and low-level descriptions. For each representation, a *rep-map* will be defined to specify the correspondence between the internal ports (internal system representation) and the external ports (its external interface).

Now we will introduce the extension of Acme, Dynamic Acme, which provides dynamic reconfiguration mechanisms through the notion of *open* systems. A system is said to be *closed* if one can assume that all of its parts have been described and that the only parts that will exist have been described [33]. When we use the keyword open before a system, it indicates that the complete specification of the system is not present, which implies that a dynamic number of instances of some components or connectors within the system might be created or deleted, and the configuration structure of the system will be changed during run time [33].

The key mechanism in Dynamic Acme to specify a dynamic element is the identification process. A component is alive when it is identified to exist in the system. For the same reason, the properties of the dynamic elements and the relationships between them need only hold when they have been identified. A prefix operator, *id*, is introduced into the predicate language to represent that an element of the architecture has been identified with some elements of the artifact being described. Now, we can specify a dynamic Producer-Consumer system with a finite number (n) of consumers in Dynamic Acme as follows:

```
open System Dynamic- Producer-Consumer = {
    Component producer = { Port send }

    open 1..n Component consumer = {
            Port extract;
            Properties { busy: boolean } }

    open 1..n Connector cable = {
            Role sender;
            Role receiver } }

    Attachments {
            producer.send to cable[*].sender;
            consumer[*].extract to cable[*].receiver }
```

Constraints:

forany i :: (**id** consumer[i]) **and** (**id** cable[i]) =>
(**forany** j :: j != i => (**not id** consumer[j]) **and** (**not id** cable[j]))
exists j :: (j > 0) **and** (j < n+1) **and** (**id** consumer[j]) **and** (**id** cable[j])
id consumer[j] => consumer[j].busy = false
}

Component producer is fixed in the system. Since we may have a dynamic number of consumers in the system and one of them should be free to be connected with the producer, keyword open is used to indicate consumers are dynamic elements of the system and there may exist one or many consumer instances. Meanwhile, the connector cable should also be dynamic to connect the producer with these consumers, respectively, as specified in the attachments section. The key section is the constraints section, where we state that only one live consumer and one live cable is connected with the producer (constraints 1), and there must exist a live connection between the producer and one consumer (constraint 2). Finally, the live consumer should not be busy (constraint 3). A graphical representation of the system is shown in Figure 2.6.

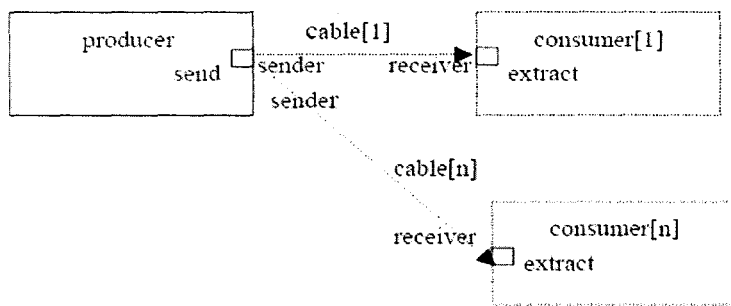


Figure 2. 6 The Dynamic-Producer-Consumer system

In the above example, the declarative style of specifications in Dynamic Acme enables us to describe a dynamic number of consumers within the system by the identification mechanism and using the constraints to specify possible architectural configurations. However, we are not able to define the reconfiguration operations (such as the switch operation) explicitly in the specification due to the lack of imperative constructs in the language.

2.4 CommUnity and its semantics

In this section we will review the syntax and semantics of CommUnity. The review is heavily based on the work of Tom Maibaum and J.L. Fiadeiro in [15], which serves as a milestone for the language. There have been some changes to CommUnity (e.g. the introduction of progress guards, the mapping of the actions is from the target design to the source design and partial) [18] since the publication of that paper, and we have updated the corresponding definitions, propositions and proofs in this thesis.

CommUnity is an ADL developed to illustrate the categorical formalization of parallel program design, which is in the style of UNITY and combines elements from Interacting Processes for a richer model of system interconnection and superposition. The language and the design framework have been extended to provide a formal platform for testing ideas and experimenting with techniques for the architectural design of open, reactive and reconfigurable systems.

The modeling of software architecture requires the description of the architectural structure by connected graphs of components, connectors and the bindings between them, which are called architectural configurations. In CommUnity architectural configuration is described by a configuration diagram, which consists of the components, connectors, and the morphisms between them. The model of interaction between components is based on action synchronization and the interconnection of input channels of a component with output channels of other components, which are standard means of interconnecting components. This mechanism gives rise to CommUnity's approach to architectural description: to coordinate the components and build large systems from simpler components. In addition, the categorical framework on which CommUnity is based ensures that the system's semantics can be derived from the colimit of its configuration diagram. In the following discussion (heavily based on [15]) we will focus on the distinguishing features of CommUnity and clarify the reasons for the language's general suitability to architectural description of complex systems.

- **Compositionality**

Compositionality is a mechanism that allows architectures to describe software systems at different levels of detail: complex structure and behavior may be explicitly represented or they may be abstracted into a single component or connector [28]. To enable an ADL to support this feature, modularity and

layered, incremental system design must be incorporated into the design of the language.

CommUnity is based on Category Theory, which supports the definition of "scientific laws" of system modularization and composition. The unification of modularization principles provided by Category Theory applies not only to mathematical models of program behavior and their logical specifications, but also to parallel program design languages, such as CommUnity, based on the notion of *superposition*. Superposition was proposed as a means of supporting a layered approach to systems design by which we are allowed to build on already developed components by "augmenting" them through extending their state space and/or their action/control activity while preserving their properties. In mathematics, preservation of structure is usually formalized in terms of homomorphisms between the objects concerned, so in CommUnity superposition is formalized in terms of morphisms of programs.

For example, with the notion of regulative superposition, we can design a simple component (say account, with the functionality of deposit and withdraw) and superpose a regulator upon it (a guard for withdraw). An explicit diagram to show the configuration of the regulated account consists of account, regulator, the implicit cable to interconnect them and the regulative superposition morphisms between these components. The colimit of this configuration diagram is considered as a new component, which represents the regulated account that itself can be interconnected with other components in the system.

- **Reusability**

CommUnity supports the discipline of reuse in the sense that components can be developed independently and interconnected at system configuration time. Names are local to designs in CommUnity, which means that the use of the same name in different designs is treated as being purely accidental and expresses no relationship between the components. This use of local names, as opposed to the usual approach of a global name space, is essential to support such a degree of reusability and the resulting systems are also structured because they are connected to their components through the colimit morphisms.

Locality of names is intrinsic to Category Theory and it forces interconnections to be explicitly established outside the designs. Hence, the categorical framework of CommUnity is much more apt to support the complete separation between the structural language that describes the software architecture and the language in which the components are themselves programmed or specified.

- **The enforcement of design principles**

This ability to characterize the structure of objects in terms of relationships (morphisms) with other objects and to define operations of composition that preserve that structure is one of the reasons that make the categorical framework so useful for formalizing disciplines of decomposition and organization of systems into components. That is, choosing a particular notion of morphism, we define a way of establishing relationships between objects and, hence, of structuring our world according to the components that these relationships allow us to identify.

Indeed, one of the basic principles of the categorical approach is that, for every notion of structure, there is a corresponding notion of transformation (morphism) that preserves that structure. For instance, with respect to the category of designs with regulative superposition morphisms, one of the structural notions enforced is encapsulation of local state (attributes): the fact that morphisms are required to preserve the locality of design attributes implies that any operation on designs defined, like colimits, in terms of universal properties of morphisms, will guarantee that the attributes of the component designs remain local.

In this sense, we can claim that categories can be used to formalize system design disciplines. By changing from one category to another, for instance by keeping the same objects (designs), but changing the way we can interconnect them (morphisms), we obtain a different paradigm.

- **Refinement and Traceability**

Refinement is an important dimension in structuring development. A notion of refinement morphism has been defined in CommUnity to capture the refinement relation for CommUnity designs, which gives rise to the well-defined concept of refinement of components and connectors. As a result, CommUnity enables correct and consistent refinement of architectures into executable systems and traceability of changes across levels of architectural refinement, which enables us to overcome the problems associated with the use of informal "boxes and lines" diagrams for designs and informal programming language concepts.

2.4.1 The syntax of the language

In CommUnity, the basic building block of the language is called a design or, as

a special case, a program. The syntax of a CommUnity design is:

design P **is**
out out(V)
in in(V)
prv prv(V)
do
[**prv**] g[D(g)] : L(g), U(g) -> R(g)

A fixed collection of data types (say S) is assumed to be given by a first-order algebraic specification and the design is defined over such data types. Because data types chosen in the design determine the nature of the elementary computations that can be performed locally by the components, the emphasis in the language is put on the coordination mechanisms between system components rather than data refinement which focuses on computational aspects. As a result, it does not support polymorphism directly.

In the above example, V is the set of *channels* in the design P. Each channel v is typed with a sort from S. in(V) represents input channels, which read data from the environment of the component and the component has no control over them. out(V) and prv(V) are output channels and private channels, respectively. They are controlled locally by the component. Output channels allow the environment to read data produced by the component, while private channels support internal activity that does not involve the environment. We use loc(V) to represent out(V) $\cup$ prv(V).

For any action g, D(g) is a subset of loc(V) consisting of the local channels that can be written to by action g (we call it the write frame of g). U(g) is a progress condition, which establishes the upper bound for enabledness and L(g) indicates the lower bound. In a program, L(g) = U(g), so the guards in a design define the "interval" within which the guard of the action in a program implementing the design must lie. R(g) is a condition on V and D(g)', where by D(g)' we mean the set of primed channels from D(g). Primed channels account for references to the values of channels after the execution of an action. The condition is a first-order logic formula built from V and D(g)'. Usually, we define it as a conjunction of implications of the form pre $\Rightarrow$ post, which corresponds to a pre/post condition specification in the sense of Hoare and where pre does not contain primed channels. Using this form, the number of conjuncts in the formula will correspond to the number of channels in the write frame of g, so that we can understand the meaning of the action fairly easily. Moreover, it will be convenient for us to calculate the colimit of the diagram

(see 2.4.4.3) if we have put all the designs in this form.

In order to study the relationship between designs, we need the formal definition for designs as follows:

Definition 2.1 A *design signature* is a tuple (V, Γ, tv, ta, D) where:
* V is the set of *channels*, which is an S-indexed family of mutually disjoint sets. The channel is typed with sorts in S, which is a fixed set of data types specified as usual via a first-order specification.
* Γ is a finite set of actions.
* tv is a total function from V to {prv, in, out}, which partitions V into three disjoint sets of channels, namely private, input and output channels, respectively. Loc(V) represents the union of private and output channels.
* ta is a total function from Γ to {sh, prv}, which divides Γ into private and shared actions. Only shared actions can serve as the synchronization points with other designs.
* D is total function from Γ to $2^{loc(V)}$. The write frame of action g is represented by D(g).

Since we allow parameters in the definition of component, connector and subsystem, it may influence the signature of the design. For example, if the parameter is a sort which can be instantiated by some sort s in S, the design 's signature will depend on the sort s we choose. Therefore, in this thesis when we talk about the signature of a design, we actually refer to the instantiated design, where its parameters have already been instantiated.

All these sets of symbols are assumed to be finite and mutually disjoint. Channels are used as atoms in the definition of terms:

Definition 2.2: Given a design signature θ=(V, Γ, tv, ta, D), the language of *terms* is defined as follows:
for every sort $s \in S$,
* $t_s ::= a$, where $a \in V$ and of type s
* $t_s ::= c$, where c is a constant with sort s
* $t_s ::= f(t_1,\ldots,t_n)$, where $t_1: s_1,\ldots, t_n: s_n$ and $f : s_1 \times \ldots \times s_n \rightarrow s$

The language of *propositions* is defined as follows:
$\phi ::= (t_{1s}\ p_s\ t_{2s}) \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi$
where p_s is a binary predicate defined on sort s. The set of predicates defined on sort s must contain $=_s$.

Having defined the signature of designs and given the language of terms and propositions, we can formalize the notion of designs as follows:

Definition 2.3: A *design* is a pair (θ, Λ), where $\theta = (V,\Gamma,tv,ta,D)$ and Λ is (I,R,L,U) where:
* I is a proposition defined on θ, which constrains the values of the channels when the program is initialized.
* R assigns to every action $g \in \Gamma$ an expression R(g).
* For every action $g \in \Gamma$, L(g) assigns the enabling guard to it and U(g) assigns the progress guard.
* For every action $g \in \Gamma$, for any $a \in D(g)$, $tv(a) \in \{prv, out\}$.

Recall that R(g) specifies the effect of action g on its write frame. For any channel $a \in D(g)$, we will use R(g,a) to denote the expression that represents the effect of action g on channel a.

2.4.2 The semantics of CommUnity designs

Before we define the semantic structures for a design, a model for the abstract data type specification (S) needs to be introduced. The model is given by a Σ-algebra U, i.e., a set s^U is assigned to each sort symbol $s \in S$, a value in $s^U(c^U)$ is assigned to each constant symbol c of sort s, a (total) function $f^U : s_1{}^U \times \ldots \times s_n{}^U \rightarrow s^U$ is assigned to each function symbol f in S, and a relation $p_s{}^U \subseteq s \times s$ is assigned to each binary predicate p_s defined on sort s.

The semantic interpretation of designs is given in terms of transition systems:
Definition 2.4: A *transition system* $(W, w_0, E, \rightarrow)$ consists of:
* a non-empty set W of states or possible worlds
* $w_0 \in W$, the initial state
* a non-empty set E of events
* an E-indexed set of partial functions $\rightarrow$ on W, $W \rightarrow (E \rightarrow W)$, defines the state transition performed by each event.

Having transition systems to represent the state transitions of a design, we can interpret the signature of a design with the following structure:

Definition 2.5: A *θ-interpretation structure* for a signature θ=(V, Γ, tv, ta, D) is a triple (*T*, *A*, *G*) where:
* *T* is a transition system (W, w_0, E, $\rightarrow$)
* *A* is an S-indexed family of maps A_s: $V_s \rightarrow (W \rightarrow s^U)$.
* G: $\Gamma \rightarrow 2^E$.

That is to say, *A* interprets attribute symbols as functions that return the value that each attribute takes in each state, and *G* interprets the action symbols as sets of events -- the set of the events during which the action occurs.

It is possible that no action will take place during an event. Such events correspond to environment steps, which means steps performed by the other components in the system. Interpretation structures are intended to capture the behavior of a design in the context of a system of which it is a component. Because environment steps are taken into account, state encapsulation techniques can be formalized through particular classes of interpretation structures.

Definition 2.6: A *θ-interpretation structure* (*T*,*A*,*G*) for a signature θ=(V, Γ, tv, ta, D) is called a *locus* iff, for every a∈loc(V) and w, w' ∈ W, if (w, e, w') is in $\rightarrow$, and for any g∈D(a), e is not in G(g), then A(a)(w') = A(a)(w).

This means a *locus* is an interpretation structure in which the values of the program variables remain unchanged during events in which no action occurs that contains them in their write frame.

Having defined the interpretation structures for designs and the model for the abstract data type specification (S), we are able to give the semantics of the terms and propositions in the language given by the design signature.

Definition 2.7: Given a signature θ = (V, Γ, tv, ta, D) and a *θ-interpretation structure* *S*= (*T*,*A*,*G*), the semantics of terms (for every sort s, term t of sort s and w∈ W, $[t]^s(w) \in s^U$ is the value taken by t in the world w) is defined as follows:
* if t is a∈A_s, $[a]^s(w) = A(a)(w)$
* if t is a constant c, $[c]^s(w) = c^U$
* if t is $f^U : s_1{}^U \times \ldots \times s_n{}^U \rightarrow s^U$, $[f(t_1,t_2,\ldots,t_n)]^s(w) = f^U([t_1]^s(w), [t_2]^s(w), \ldots, [t_n]^s(w))$

The semantics of propositions is defined as:
* (S,w) |= ($t_1 =_s t_2$) iff $[t_1]^s(w) = [t_2]^s(w)$
* (S,w) |= (t_1 p_s t_2) iff $[t_1]^s(w)$ $p_s{}^U$ $[t_2]^s(w)$
* (S,w) |= $\phi_1 \Rightarrow \phi_2$, iff ($S$,w) |= ϕ_1 implies (S,w) |= ϕ_2
* (S,w) |= ($\neg\phi$) iff $\neg$((S,w) |= ϕ)

Now on the semantic level, we can represent whether a proposition (in a signature) is true or valid in the interpretation structure of the signature:

Definition 2.8: A θ-proposition ϕ is *true* in an θ-interpretation structure S, written $S \models \phi$, iff $(S,w) \models \phi$ at every state w. A proposition ϕ is *valid*, written $\models \phi$, iff it is true in every interpretation structure.

Having introduced the above concepts, we can now define when an interpretation structure is a model of a design.

Definition 2.9: Given a design (θ, Λ), where $\theta = (V,\Gamma,tv,ta,D)$ and Λ is a triple (I,R,L,U), a *model* of (θ, Λ) is an interpretation structure $S=(T,A,G)$ for θ, such that:

* $(S,w_0) \models I$
* for every $g \in \Gamma$, $a \in D(g)$, $e \in G(g)$, and (w, e, w') is in $\rightarrow$, then $A(a)(w')= [R(g,a)]^s(w)$
* for every $w \in W$ and $g \in \Gamma$, if $e \in G(g)$ and for some $w' \in W$, (w, e, w') is in $\rightarrow$, then $(S,w) \models L(g)$.

That is to say, a *model* of a design is an interpretation structure for its signature that enforces the assignments, only permits actions to occur when their enabling guards are true, and for which the initial state satisfies the initialization constraint.

A model is said to be a *locus* if it is a locus as an interpretation structure, which enforces the encapsulation of local attributes.

Since the progress guard was introduced after the publication of [15], we need to update the definition of *politeness*. A model S is said to be *polite* iff for every $w \in W$ and $g \in \Gamma$, $(S,w) \models U(g)$ implies that there exists $e \in G(g)$ and $w' \in W$ such that(w, e, w') is in $\rightarrow$. That is to say, a model is *polite* if actions are allowed to affect transitions in every world in which their progress guard is satisfied. This notion generalizes the notion of *fairness* as used in parallel program design.

This classification of models reflects the existence of different levels of semantics for the same design (taken as a set of models), depending on which subset of the set of its models is considered. These different semantics are associated with different notions of superposition (design morphism) that have been used in the literature, namely regulative, invasive and spectative. This means that there is no absolute notion of semantics for designs: it is always relative to the use one makes of designs. This corresponds to the categorical way of capturing

the "meaning" of objects through the relationships (morphisms) that can be defined between them.

2.4.3 The morphisms between designs

The concept of superposition has been proposed and used as a structuring mechanism for the design of parallel programs and distributed systems. Structuring preserving transformations are usually formalized in terms of homomorphisms between the objects concerned, thus justifying the formalization of superposition in terms of morphisms of designs in CommUnity.

Having defined designs over signatures in the above section, we first introduce signature morphisms as a means of relating the "syntax" of two designs.

Definition 2.10: A *signature morphism* σ from a signature $\theta_1=(V_1,\Gamma_1, tv_1, ta_1, D_1)$ to $\theta_2=(V_2,\Gamma_2, tv_2, ta_2, D_2)$ consists of a total functions σ_α: $V_1 \rightarrow V_2$, and a partial mapping $\sigma_\gamma : \Gamma_2 \rightarrow \Gamma_1$ such that:
* For every $v \in V_1$, $\sigma_\alpha(v)$ has the same type as v.
* For every $o \in out(V_1)$, $\sigma_\alpha(o) \in out(V_2)$.
* For every $p \in prv(V_1)$, $\sigma_\alpha(p) \in prv(V_2)$.
* For every $i \in in(V_1)$, $\sigma_\alpha(i) \in out(V_2) \cup in(V_2)$.
For every $g \in \Gamma_2$, such that $\sigma_\gamma(g)$ is defined:
* $g \in sh(\Gamma_2)$, then $\sigma_\gamma(g) \in sh(\Gamma_1)$.
* $g \in prv(\Gamma_2)$, then $\sigma_\gamma(g) \in prv(\Gamma_1)$.
* $\sigma_\alpha(D_1(\sigma_\gamma(g)) \subseteq D_2(g)$.

A signature morphism maps attributes of a design to attributes of the system of which it is a component, and the direction of the mapping is reversed for actions. The first condition enforces the preservation of the type of each attribute by the morphism. Output and private attributes of the component should keep their classification in the system, while input attributes may be turned into output attributes, when they are synchronized with output channels of other components and thus represented as output channels of the system. The restriction over action domains means that the type of each action is preserved by the morphism. In other words, the images of the write frame of an action in the source program must be contained in the write frame of the corresponding action in the target program. Notice that more attributes may be included in the

domain of the target program's action via a morphism. This is intuitive because an action of a component may be shared with other components within a system and, hence, has a larger domain.

Signature morphisms provide us with the means for relating a design with its superpositions. However, superposition is more than just a relationship between signatures on the level of syntax. To capture its semantics, we need a way of relating the models of the two designs as well as the terms and propositions that are used to build them.

Signature morphisms define translations between the languages associated with each signature in the obvious way:

Definition 2.11: Given a signature morphism σ: $\theta_1 \rightarrow \theta_2$, we can define translations between the languages associated with each signature:

* t is a term

$\sigma(t) ::= \sigma(a)$	t is a variable a
c	t is a constant c
$f(\sigma(t_1),\ldots, \sigma(t_n))$	$t= f(t_1,\ldots, t_n)$

* ϕ is a proposition

$\sigma(\phi) ::= \sigma(t_1) = \sigma(t_2)$	ϕ is $t_1 = t_2$
$\sigma(t_1)\ p_s\ \sigma(t_2)$	ϕ is $t_1\ p_s\ t_2$
$\sigma(\phi_1) \Rightarrow \sigma(\phi_2)$	ϕ is $\phi_1 \Rightarrow \phi_2$
$\sigma(\phi_1) \wedge \sigma(\phi_2)$	ϕ is $\phi_1 \wedge \phi_2$
$\neg\sigma(\phi')$	ϕ is $\neg\phi'$

Definition 2.12: Given a signature morphism σ: $\theta_1 \rightarrow \theta_2$ and a θ_2–interpretation structure $S = (T,A,G)$, its *σ-reduct*, $S|_\sigma$, is the θ_1–interpretation structure $(T, A|_\sigma, G|_\sigma)$, where $A|_\sigma(a) = A(\sigma(a))$, $G|_\sigma(g) = \cup\ G(\sigma^{-1}(g))$. (We change the definition of $G|_\sigma(g)$ in [15] since the mapping of actions has been changed as $\Gamma_2 \rightarrow \Gamma_1$.)

That is, we take the same transition system of the target design and interpret attribute symbols of the source design in the same way as their images under σ, and action symbols of the source design as the union of their images under σ^{-1}. Reducts provide us with the means for relating the behavior of a design with that of the superposed one. The following proposition establishes that properties of reducts are characterized by translation of properties.

Proposition 2.1: Given a θ_1 proposition ϕ and a θ_2–interpretation structure $S=(T,A,G)$, we have for every $w \in W$: $(S,w) \models \sigma(\phi)$ iff $(S|_\sigma, w) \models \phi$.

We prove this proposition as follows:

We can prove this proposition by induction on the structure of ϕ. The base case is easy to check. For the induction step, we assume this proposition holds on the subformulae of ϕ:

* ϕ is $t_1 = t_2$

If $(S,w) \models \sigma(\phi)$, so $(S,w) \models (\sigma(t_1) = \sigma(t_2))$, which implies in state w, $A(\sigma(t_1)) = A(\sigma(t_2))$. Since $A|_\sigma(t_1) = A(\sigma(t_1))$, $A|_\sigma(t_2) = A(\sigma(t_2))$, we have in state w, $A|_\sigma(t_1) = A|_\sigma(t_2)$, therefore $(S|_\sigma,w) \models t_1 = t_2$, $(S|_\sigma ,w) \models \phi$.
The other direction can be proved similarly.

* ϕ is $t_1\ p_s\ t_2$

If $(S,w) \models \sigma(\phi)$, so $(S,w) \models (\sigma(t_1)\ p_s\ \sigma(t_2))$, which implies in state w, $A(\sigma(t_1))\ p_s\ A(\sigma(t_2))$. Since $A|_\sigma(t_1) = A(\sigma(t_1))$, $A|_\sigma(t_2) = A(\sigma(t_2))$, we have in state w, $A|_\sigma(t_1)\ p_s\ A|_\sigma(t_2)$, therefore $(S|_\sigma,w) \models t_1\ p_s\ t_2$, $(S|_\sigma ,w) \models \phi$.
The other direction can be proved similarly.

* ϕ is $\phi_1 \Rightarrow \phi_2$

Suppose $(S|_\sigma, w) \models \phi$, so $(S|_\sigma, w) \models \phi_1 \Rightarrow \phi_2$, which means $(S|_\sigma, w) \models \phi_1$ implies $(S|_\sigma, w) \models \phi_2$; from the induction hypothesis, $(S, w) \models \sigma(\phi_1)$ iff $(S|_\sigma, w) \models \phi_1$, $(S, w) \models \sigma(\phi_2)$ iff $(S|_\sigma, w) \models \phi_2$, so we have $(S, w) \models \sigma(\phi_1)$ implies $(S, w) \models \sigma(\phi_2)$, $(S, w) \models \sigma(\phi_1) \Rightarrow \sigma(\phi_2)$, since $\sigma(\phi_1 \Rightarrow \phi_2) = \sigma(\phi_1) \Rightarrow \sigma(\phi_2)$, $(S, w) \models \sigma(\phi_1 \Rightarrow \phi_2)$, $(S, w) \models \sigma(\phi)$.
The other direction can be proved similarly.

* ϕ is $\neg\phi'$

If $(S, w) \models \sigma(\phi)$, then $(S, w) \models \sigma(\neg\phi')$. Since $\sigma(\neg\phi') = \neg\sigma(\phi')$, we have $(S, w) \models \neg\sigma(\phi')$. From the definition of the semantics of propositions,
$(S, w) \models (\neg\phi)$ iff $\neg$ $((S, w) \models \phi)$, so we have $\neg$ $((S, w) \models \sigma(\phi'))$. From the induction hypothesis, $(S, w) \models \sigma(\phi')$ iff $(S|_\sigma, w) \models \phi'$, so we have $\neg((S|_\sigma, w) \models \phi')$, $(S|_\sigma, w) \models \neg\phi'$, $(S|_\sigma, w) \models \phi$.
The other direction can be proved similarly.

To identify the properties of morphisms required for capturing the superposition relationship between designs, invasive superposition is introduced to enforce that the functionality of the base design be preserved in terms of the assignments performed on its local channels, and it allows for the guards of its actions to be strengthened. However, there is nothing to prevent "old attributes",

i.e. translations of attributes of the base design, to be changed by "new actions", i.e. actions of the target design that are not mapped to the actions of the base design. Therefore, superposition morphisms that preserve locality are introduced as regulative superposition morphisms as follows:

Definition 2.13 A *regulative superposition morphism* σ from a design (θ_1, Λ_1) to another design (θ_2, Λ_2) is a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$ such that:
1. $\models (I_2 \Rightarrow \sigma(I_1))$.
2. If $v \in loc(V_1)$, $g \in \Gamma_2$ and $\sigma_\alpha(v) \in D_2(g)$, then g is mapped to an action $\sigma_\gamma(g)$ and $v \in D_1(\sigma_\gamma(g))$.
For every $g \in \Gamma_2$ for which $\sigma_\gamma(g)$ is defined,
3. If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then $\models (R_2(g, \sigma_\alpha(v)) \Leftrightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)))$.
4. $\models (L_2(g) \Rightarrow \sigma(L_1(\sigma_\gamma(g))))$.
5. $\models (U_2(g) \Rightarrow \sigma(U_1(\sigma_\gamma(g))))$.

Notice that we do not require σ_α to be injective, and two channels of the same category (output/private/input) in the source design can be mapped to one channel of the target design. Because we only consider the actions in the target design mapped to the source design, σ_γ does not need to be surjective.

The second condition implies that actions of the system in which a component C is not involved cannot have local channels of the component C in their write frame, which corresponds to the locality condition: new actions cannot be added to the domains of attributes of the source program. The justification is as follows: suppose system action g has $\sigma_\alpha(v)$ in its write frame, $v \in loc(V_1)$, then $\sigma_\gamma(g)$ must be defined, and $\sigma_\gamma(g) \in D_1(v)$. Therefore, component C is involved in the system action.

Regulative superposition morphisms require that the functionality of the base design in terms of its variables be preserved (the underspecification cannot be reduced) and allows for the enabling and progress conditions of its actions to be strengthened. Strengthening of the lower bound reflects the fact that all the components that participate in the execution of a joint action have to give their permission for the action to occur. On the other hand, the progress of a joint action can only be guaranteed when the involved components can locally guarantee so. Regulative superpositions preserve encapsulation and do not change the actions themselves, as far as they relate to the basic variables.

Proposition 2.2: Let σ: $(\theta_1, \Lambda_1) \rightarrow (\theta_2, \Lambda_2)$ be a regulative superposition morphism. Then the reduct of every model of (θ_2, Λ_2) is also a model of (θ_1, Λ_1).

We prove this proposition as follows:

Suppose $S=(T,A,G)$ is a model of (θ_2, Λ_2); we need to show its *σ-reduct,* $S|_\sigma = (T, A|_\sigma, G|_\sigma)$ is a model of (θ_1, Λ_1).

1. $(S|_\sigma, w_0) \models I_1$

We have $(S, w_0) \models I_2$; since $\models (I_2 \Rightarrow \sigma(I_1))$, so $(S, w_0) \models \sigma(I_1)$, and from proposition 2.1, we can get $(S|_\sigma, w_0) \models I_1$.

2. For every $g \in \Gamma_1$, $a \in D(g)$, $e \in G|_\sigma(g)$, and (w, e, w') is in $\rightarrow$, then $A|_\sigma(a)(w') = [R(g,a)]^{S|\sigma}(w)$.

Since $e \in G|_\sigma(g) = \cup\, G(\sigma^{-1}(g))$, pick one action $g' \in \Gamma_2$ mapped to g and $e \in G(g')$. Because $a \in D(g)$, from the definition of signature morphism, $\sigma(a) \in D(g')$. (w,e,w') is in $\rightarrow$, and $S=(T,A,G)$ is a model of (θ_2, Λ_2), so we have $A(\sigma(a))(w') = [R_2(g',\sigma(a))]^S(w)$.

By the third condition of regulative superposition morphism, $[R_2(g',\sigma(a))]^S(w) = [\sigma(R(g,a))]^S(w)$, and from the definition of σ-reduct, it is easy to see $[R(g,a)]^{S|\sigma}(w) = [\sigma(R(g,a))]^S(w)$ and $A|_\sigma(a)(w') = A(\sigma(a))(w')$.

Therefore, we have $A|_\sigma(a)(w') = [R(g,a)]^{S|\sigma}(w)$.

3. For every $w \in W$ and $g \in \Gamma_1$, if $e \in G|_\sigma(g)$ and for some $w' \in W$, (w, e, w') is in $\rightarrow$, then $(S|_\sigma, w) \models L_1(g)$.

$e \in G|_\sigma(g) = \cup\, G(\sigma^{-1}(g))$, pick up one action $g' \in \Gamma_2$ mapped to g and $e \in G(g')$. Because $S=(T,A,G)$ is a model of (θ_2, Λ_2) and (w, e, w') is in $\rightarrow$, we have $(S, w) \models L_2(g')$.

By the definition of regulative superposition morphism, $L_2(g') \Rightarrow \sigma(L_1(\sigma_\gamma(g')))$, so we have $L_2(g') \Rightarrow \sigma(L_1(g))$, so $(S, w) \models \sigma(L_1(g))$.

From proposition 2.1, $(S, w) \models \sigma(L_1(g))$ iff $(S|_\sigma, w) \models L_1(g)$.

We find that in the proof of proposition 2.2, we do not use condition 2 of regulative superposition morphism, which means this proposition will hold without enforcing the encapsulation principle. When we consider condition 2 and the definition of signature morphism, we will have the following assertion:

Proposition 2.3: If $v \in loc(V_1)$, then $D_1(v) = \sigma_\gamma(D_2(\sigma_\alpha(v)))$.

Proof:

(1) $D_1(v) \supseteq \sigma_\gamma(D_2(\sigma_\alpha(v)))$.

This is obvious from condition 2.
(2) $D_1(v) \subseteq \sigma_\gamma(D_2(\sigma_\alpha(v)))$.
Suppose $g' \in D_1(v)$ and $g'' \in \Gamma_2$ is mapped to g'. Since we have $\sigma_\alpha(D_1(\sigma_\gamma(g)) \subseteq D_2(g)$, so $\sigma_\alpha(D_1(g')) \subseteq D_2(g'')$, with $v \in D_1(g')$, we have $\sigma_\alpha(v) \in D_2(g'')$. Therefore, $g'' \in D_2(\sigma_\alpha(v))$, since $g' = \sigma_\gamma(g'')$, we get $g' \in \sigma_\gamma(D_2(\sigma_\alpha(v)))$.
Note: we only consider the actions in $D_1(v)$ which have been mapped to the target design.

This result implies the following property:

Proposition 2.4: Let σ: $(\theta_1, \Lambda_1) \rightarrow (\theta_2, \Lambda_2)$ be a regulative superposition morphism, then the reduct of every locus of (θ_2, Λ_2) is also a locus of (θ_1, Λ_1).

The reason is that through regulative superposition, the domains of the attributes remain the same up to translation, as we prove above. Therefore, it will prevent "old attributes" from being changed by "new actions", i.e., actions of the target design not mapped to the source design.

It is easy to see that for regulative superposition morphisms, the reduct of a polite model of the target design is not necessarily polite for the source design and the counterexample can be derived from the proof of proposition 2.5. However, if we do not allow the enabling and progress guards to be strengthened, it can be proved (in proposition 2.5) that reducts preserve politeness. Such superposition morphisms are called spectative superpositions:

Definition 2.14: A *spectative superposition morphism* σ from a design (θ_1, Λ_1) to another design (θ_2, Λ_2) is a regulative superposition morphism such that:
1. σ_γ is bijective, σ_α is injective.
2. For every proposition ϕ in the language of θ_1, if $|=_{\theta 2} (I_2 \Rightarrow \sigma(\phi))$, then $|=_{\theta 1} (I_1 \Rightarrow \phi)$.
For every $g \in \Gamma_2$ where $\sigma_\gamma(g)$ is defined,
3. $|= (L_2(g) \Leftrightarrow \sigma(L_1(\sigma_\gamma(g))))$.
4. $|= (U_2(g) \Leftrightarrow \sigma(U_1(\sigma_\gamma(g))))$.

Injectivity of σ means that no confusion is introduced among actions or attributes. The second condition requires that the strengthening of the initial condition be conservative, i.e, it cannot put further constraints on the initial values of the attributes of the source program. The enabling and progress guards should remain unchanged in spectative superposition.

Proposition 2.5: Let σ: $(\theta_1,\Lambda_1) \rightarrow (\theta_2, \Lambda_2)$ be a spectative superposition morphism, if $S=(T,A,G)$ is a polite model of (θ_2, Λ_2), then its *σ-reduct,* $S|_\sigma = (T, A|_\sigma, G|_\sigma)$ is a polite model of (θ_1, Λ_1).

Proof:

In $S|_\sigma$, for any $w \in W$ and $g \in \Gamma_1$, if $(S|_\sigma, w) \models U_1(g)$, then according to proposition 2.1, $(S, w) \models \sigma(U_1(g))$. Since σ is a spectative superposition morphism and σ_γ is bijective, action g' of the target design is mapped to g and $U_2(g') \Leftrightarrow \sigma(U_1(g))$, so $(S, w) \models U_2(g')$. Since S is polite, there exists $e \in G(g')$ and $w' \in W$ such that(w, e, w') is in $\rightarrow$. Because $G|_\sigma(g) = \cup\, G(\sigma^{-1}(g))$, $e \in G|_\sigma(g)$, so g will occur and we know that $S|_\sigma = (T, A|_\sigma, G|_\sigma)$ is a polite model of (θ_1, Λ_1).

We can prove a fundamental property of spectative superposition: it is model-expansive, which means that it does not change the base (source) design, i.e., the base design is extended by spectative superposition without affecting its underlying behavior. Model-expansive transformations have been identified as playing a very important role in modularity and this property suggests the definition of the notion of superposing an observer on a base design. It will also appear in the extension morphism we will introduce later.

Proposition 2.6: Let σ: $(\theta_1, \Lambda_1) \rightarrow (\theta_2, \Lambda_2)$ be a spectative superposition morphism; for every model S of (θ_1, Λ_1), there will exist a model S' of (θ_2, Λ_2), such that $S \sim S'|_\sigma$.

This result implies that for every model S of the base design, we will be able to find a model S' of the target design, whose reduct is equivalent to S and demonstrates the same behavior as S. By $S \sim S'|_\sigma$ we mean that these two interpretation structures (for the signature $\theta = (V, \Gamma, tv, ta, D)$) are equivalent and there exists an isomorphism between them, which consists of a relation R between W and W', and a relation Q between E and E' such that:

* w_0 R w_0';
* if (w_1, e, w_2) is in $\rightarrow$ and w_1 R w_1', then there exists $e' \in E'$, $w_2' \in W'$, such that e Q e', w_2 R w_2', and (w_1', e', w_2') is in $\rightarrow'$.

  if (w_1', e', w_2') is in $\rightarrow'$ and w_1 R w_1', then there exists $e \in E$, $w_2 \in W$, such that e Q e', w_2 R w_2', and (w_1, e, w_2) is in $\rightarrow$.
* if w R w', then for every $a \in V$, $A(a)(w) = A'(a)(w')$.

* if e Q e', then $G(e) = G'(e')$.

It is worth mentioning that this definition comes from bisimulation [25], which defines an equivalence relation between state transition systems.

We find that this result will not hold for a regulative superposition morphism. The reason is as follows:

If this proposition holds for regulative superposition, then every run P of the source design will have a corresponding run P' in the target design. Suppose s is the start state of P, in which I_1 will hold. The corresponding start state s' of P' will interpret the channels mapped from the source design with the same values as in s, so $\sigma(I_1)$ will hold. However, we only have $\models (I_2 \Rightarrow \sigma(I_1))$, so we cannot guarantee that I_2 will hold in s'. So, P' may not be a run of (θ_2, Λ_2).

Now we will introduce the notion of extension morphism related to the model-expansive property. The motivation of extension morphism originated from the substitutability principle from object oriented program design, which says if a component P_2 extends another component P_1, then we can replace P_1 by P_2 and the "clients" of P_1 must not perceive the difference. This principle cannot be characterized by regulative superpositions or refinement morphisms, as we may want to extend the component by breaking encapsulation.

Definition 2.15: An *extension morphism* σ from a design (θ_1, Λ_1) to another design (θ_2, Λ_2) is a signature morphism such that:

1. σ_γ is surjective.
2. σ_α is injective.
3. There exists a formula β, which contains only channels from $(V_2 - \sigma_\alpha(V_1))$, and β is satisfiable, $\models I_2 \Leftrightarrow \sigma(I_1) \wedge \beta$.

For every $g \in \Gamma_2$ for which $\sigma_\gamma(g)$ is defined,

4. If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then there exists a formula β, which contains only primed channels from $(V_2' - \sigma_\alpha(V_1)')$, and β is satisfiable,
$\models \sigma(L_1(\sigma_\gamma(g))) \Rightarrow (R_2(g, \sigma_\alpha(v)) \Leftrightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)) \wedge \beta)$.
5. If $v \in loc(V_1)$, $g \in D_2(\sigma_\alpha(v))$, then $v \in D_1(\sigma_\gamma(g))$.
6. $\models (\sigma(L_1(\sigma_\gamma(g))) \Rightarrow L_2(g))$.
7. $\models (\sigma(U_1(\sigma_\gamma(g))) \Rightarrow U_2(g))$.

This definition of extension morphism is based on [8], where we clarify an extension morphism as a signature morphism first and use the notations introduced so far. Because we expect that the extended design can replace the original design in a system and the clients of the original component should not

perceive any difference, the first two conditions ensure the preservation of its interface. The initialization condition of the original design can be strengthened in its extended version, while respecting the initialization of the channels of the original component, as required in the third condition. The fourth condition indicates that the actions corresponding to those of the original design should preserve the assignments to old channels and the assignments to new channels must be realisable, when the safety guards of their image actions in the original design are satisfied. The fifth condition establishes that for each action of the extended design that is mapped to an action of the original design, it can only modify old channels that have been modified by the corresponding action of the original design. The last two conditions indicate that both the enabling and progress guards can be weakened, but not strengthened.

Because an extension morphism relaxes the enabling guard of the source design, the reduct of a model of the target design may not be a model of the source design. However, the model-expansive property holds for extension morphism [8], which means the extended design can replace the source design and the clients of the original design will not perceive the difference.

Definition 2.16: Regulative and spectative superposition morphisms, and extension morphisms define categories we shall denote REG, SPE and EXT respectively.

To prove this result, the key point is to show that the composition of regulative (spectative, extension) morphisms is a regulative (spectative, extension) morphism (composition law), then the remaining parts to prove that the proposed structures are categories will be straightforward. The composition of two regulative (spectative, extension) morphisms $\sigma = \sigma_1 ; \sigma_2$ is defined by the composition of the corresponding mappings of channels and actions:

Suppose $\sigma_1 : \theta_1 \rightarrow \theta_2$ and $\sigma_2 : \theta_2 \rightarrow \theta_3$, the morphism σ is defined as:

* σ_α is a total function: for every channel v in θ_1, $\sigma_\alpha(v) = \sigma_{2\alpha}(\sigma_{1\alpha}(v))$.
* σ_γ is a partial mapping: for every action g in θ_3, if $\sigma_{2\gamma}(g)$ is defined and $\sigma_{1\gamma}(\sigma_{2\gamma}(g))$ is also defined, $\sigma_\gamma(g) = \sigma_{1\gamma}(\sigma_{2\gamma}(g))$; otherwise, it is undefined.

For regulative and spectative morphisms, it is easy to verify the corresponding conditions of these morphisms and the composition law will hold. In the case of extension morphism, the proof is more complicated and has been shown as Theorem 1 in [8].

These three categories just differ in their morphisms. It is the morphisms that characterize the structural properties of a category, meaning that the different notions of superposition have different algebraic properties.

The notion of morphisms defined above does not ensure that any implementation of the target design provides an implementation for the source design, which we call the refinement relation. The reason is that regulative and extension morphisms do not preserve the interval assigned to the guard of each action. Because refinement is an important aspect of structuring development, we need to introduce refinement morphism and discuss the ways of supporting it in a categorical setting.

Definition 2.17: A *refinement morphism* σ from a design (θ_1, Λ_1) to another design (θ_2, Λ_2) is a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$ such that:
1. For every $i \in in(V_1)$, $\sigma_\alpha(i) \in in(V_2)$.
2. σ_α is injective on input and output channels.
3. σ_γ is surjective on shared actions in Γ_1 .
4. $\models (I_2 \Rightarrow \sigma(I_1))$.
5. If $v \in loc(V_1)$, $g \in \Gamma_2$ and $\sigma_\alpha(v) \in D_2(g)$, then g is mapped to an action $\sigma_\gamma(g)$ and $v \in D_1(\sigma_\gamma(g))$.

For every $g \in \Gamma_2$ where $\sigma_\gamma(g)$ is defined,
6. If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then $\models (R_2(g, \sigma_\alpha(v)) \Rightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)))$.
7. $\models (L_2(g) \Rightarrow \sigma(L_1(\sigma_\gamma(g))))$.

For every shared action $g \in \Gamma_1$,
8. $\models (\sigma(U_1(g)) \Rightarrow \bigvee U_2(\sigma_\gamma^{-1}(g)))$.

A refinement morphism identifies a way in which design (θ_1, Λ_1) is refined by a more concrete design (θ_2, Λ_2). The first three conditions must be established to ensure that refinement does not change the interface between the system and its environment. Notice that we do not require σ_γ to be injective because the set of actions in the target design that are mapped to action g of the source design can be viewed as a menu of refinements that is made available for implementing g. Different choices can be made at different states to take advantage of the structures available at the more concrete level.

As for the "old actions", the last two conditions in the refinement morphism definition require that the interval defined by their enabling and progress conditions must be preserved or reduced. This is intuitive because refinement should reduce underspecification, so the enabling condition of any implementation must lie in the "old interval": the lower bound cannot be weakened and the upper bound cannot be strengthened. This is also the reason why the underspecification regarding the effects of the actions of the more abstract design are required to be reduced.

2.4.4 The composition of designs

One of the advantages of working with CommUnity in the proposed categorical framework is that mechanisms for building complex systems out of components can be formalized through universal constructs. A general principle is given by J.Goguen in his work on General System Theory [22][23][24]: "given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them."

In this section, we will first investigate the applicability of these principles to the composition of CommUnity designs based on regulative superposition, and then propose the framework in which regulative superpositions can be combined with refinement morphisms and still ensure that the colimit of the updated configuration diagram could be calculated. The use of regulative superposition as a program composition operator, which is a special kind of concurrent composition operation, can be formalized using these categorical principles.

First we will show two cases of parallel composition of designs.

2.4.4.1 Disjoint parallel composition

Coproducts are the categorical construction that explains how two components can be put together in a system without any interconnection between them. Given two designs P_1 and P_2, it consists of finding the minimal program $P_1 \parallel P_2$ that is a regulative superposition of both P_1 and P_2 . The coproduct consists of a third design $P_1 \parallel P_2$ and two regulative superposition morphisms $l_i : P_i :\rightarrow P_1 \parallel P_2$ (i=1,2) such that, given any other design P and regulative superposition morphisms $\sigma_i : P_i \rightarrow P$, there is one and only one regulative superposition morphism k: $P_1 \parallel P_2 \rightarrow P$ such that $l_i; k = \sigma_i$ (i=1,2). Minimality is expressed by the requirement on the existence and uniqueness of k. The corresponding categorical diagram is shown below:

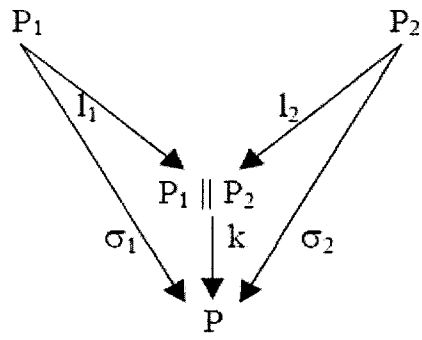


Figure 2. 7 Disjoint parallel composition of designs

Coproducts of designs are computed by first determining the coproduct of the underlying signatures. Design signatures are based on sets and functions between sets, for which coproducts compute disjoint unions.

Proposition 2.7: Signature morphisms between designs define the category SIG, which admits coproducts. The coproduct of two signatures $\theta_1=(V_1,\Gamma_1, tv_1, ta_1, D_1)$ and $\theta_2=(V_2,\Gamma_2, tv_2, ta_2, D_2)$ is given by the signature $\theta_1||\theta_2 =(V,\Gamma,tv,ta,D)$ and morphisms l_1 and l_2 ,where V is the disjoint union of V_1 and V_2, $l_{i\alpha}$ can be easily defined from channels in V_i to the corresponding channels in V; and Γ is the disjoint union of Γ_1 and Γ_2, $l_{i\gamma}$ can be defined similarly from actions in Γ to the corresponding actions in Γ_i. For every $g \in \Gamma$, suppose $l_{i\gamma}(g)$ is in Γ_i, we define $D(g) = l_{i\alpha}(D_i(l_{i\gamma}(g)))$. Having V and Γ defined, tv and ta can be defined easily.

It is easy to show l_i (i=1,2) are signature morphisms based on definition 2.10. These two morphisms keep track of the renamings in the disjoint union: for example, mapping different channels in V to the channel of V_1 and V_2 with the same name.

At the level of designs, we will move to the category REG and have the following proposition for the coproducts:

Proposition 2.8: REG admits coproducts. A coproduct of two designs $P_1 = (\theta_1, \Lambda_1)$ and $P_2 = (\theta_2, \Lambda_2)$ is given by the design $(\theta_1||\theta_2, \Lambda)$ and morphisms l_i (i=1,2) obtained as follows:

* $\theta_1||\theta_2$ and l_i (i=1,2) are a coproduct of θ_1 and θ_2.

Λ = (I,R,L,U) can be computed as follows:

* $I = l_1(I_1) \wedge l_2(I_2)$.

* For every $g \in \Gamma$, $a \in D(g)$, if $l_{i\gamma}(g)$ is in Γ_i, $l_i(a_i)=a$, we define $R(g,a) = l_i(R_i(l_{i\gamma}(g), a_i)$.
* For every $g \in \Gamma$, if $l_{i\gamma}(g)$ is in Γ_i,
$L(g) \Leftrightarrow l_i(L_i(l_{i\gamma}(g)))$,
$U(g) \Leftrightarrow l_i(U_i(l_{i\gamma}(g)))$.

Based on the definition of regulative superposition morphism, it is easy to check that l_i (i=1,2) satisfies all the conditions and together with $(\theta_1 \| \theta_2, \Lambda)$ gives the coproduct.

2.4.4.2 Parallel composition with interaction

Coproducts allow us to put together systems of components that run side by side with no interference between them. However, most systems are put together by interconnecting components. The categorical mechanisms responsible for composition with interconnections are *pushouts*.

Suppose we have two designs P_1 and P_2, and they are interconnected by a "middle" design C with the corresponding regulative superposition morphisms σ_1 and σ_2, where the channels in C identify the synchronized channels of V_1 and V_2 and the actions of C establish the rendez-vous points; the morphisms σ_1 and σ_2 identify the actions of P_1 and P_2 participating in this point of interaction as well as the channels being bound.

The design $P_1 \|_C P_2$ that we are looking for can be characterized as providing the minimal superposition $\mu_1 : P_1 \rightarrow P_1 \|_C P_2$ and $\mu_2 : P_2 \rightarrow P_1 \|_C P_2$ of P_1 and P_2 such that $\sigma_1 ; \mu_1 = \sigma_2 ; \mu_2$. The triple $< P_1 \|_C P_2, \mu_1, \mu_2 >$ is called the *pushout* of σ_1 and σ_2.

The resulting design and morphisms are related to the coproduct computed in the previous section by a morphism (coequaliser) $\mu : P_1 \| P_2 \rightarrow P_1 \|_C P_2$ such that $\mu_1 = l_1 ; \mu$ and $\mu_2 = l_2 ; \mu$. This morphism computes quotients for the equivalence relations defined by the pairs of actions and channels identified through the design C and the morphisms σ_1 and σ_2. The equivalence classes provide us with the required synchronization sets and bindings of channels, that is to say, μ imposes the required interconnections on top of disjoint parallel composition.

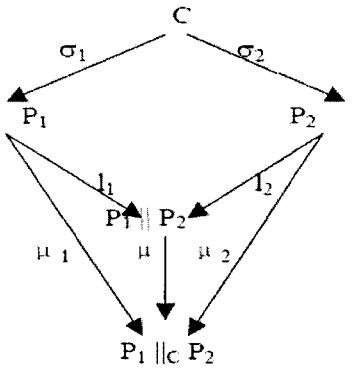


Figure 2. 8 The pushout of two designs

Proposition 2.9: REG admits pushouts. A pushout of two regulative superposition morphisms σ_1: $(\theta, \Lambda) \rightarrow (\theta_1, \Lambda_1)$ and σ_2: $(\theta, \Lambda) \rightarrow (\theta_2, \Lambda_2)$ is given by the design $(\theta_1 \|_m \theta_2, \Lambda')$ and regulative superposition morphisms μ_1: (θ_1, Λ_1) $\rightarrow (\theta_1 \|_m \theta_2, \Lambda')$ and μ_2: $(\theta_2, \Lambda_2) \rightarrow (\theta_1 \|_m \theta_2, \Lambda')$ obtained as follows:

* $\theta_1 \|_m \theta_2$, μ_1 and μ_2 are a pushout of σ_1 and σ_2 as signature morphisms. $\theta_1 \|_m \theta_2$ = (V',Γ',tv',ta',D') where V' is the amalgamated sums of V_1 and V_2 relative to V, and Γ' is the amalgamated sums of Γ_1 and Γ_2 relative to Γ. $\mu_{i\alpha}$ and $\mu_{i\gamma}$ (i=1,2) can be easily defined from P_1 and P_2 to the amalgamated sums of channels and actions based on σ_1 and σ_2. For every $g \in \Gamma'$, suppose $\sigma_{i\gamma}(g)$ is defined in both Γ_i,(i=1,2), we define $D'(g) = \cup \mu_{i\alpha}(D_i(\sigma_{i\gamma}(g)))$.

* Λ' = (I',R',L',U') can be computed as follows:
let $\mu : P_1 \| P_2 \rightarrow P_1 \|_C P_2$ be the morphism given by the coequaliser:
-- $I' = \mu_1(I_1) \wedge \mu_2(I_2)$.
-- For every $g \in \Gamma'$, $a \in D(g)$, if $\mu_{i\gamma}(g)$ is in Γ_i, $\mu_i(a_i)=a$, we define $R(g,a) = \mu_i(R_i(\mu_{i\gamma}(g), a_i)))$. (i=1,2)
-- For every $g \in \Gamma'$, if $\mu_{i\gamma}(g)$ is defined in both Γ_i, (i=1,2)
$L(g) \Leftrightarrow \wedge \mu_i(L_i(\mu_{i\gamma}(g)))$,
$U(g) \Leftrightarrow \wedge \mu_i(U_i(\mu_{i\gamma}(g)))$,
and if $a \in D(g)$, $\mu_i(a_i) = a$, $a_i \in D_i(\mu_{i\gamma}(g))$, then we have
$\mu_1(R_1(\mu_{1\gamma}(g), a_1)) = \mu_2(R_2(\mu_{2\gamma}(g), a_2))$.

2.4.4.3 Regulative superposition with refinement

In section 3.2.2 we will introduce the notion of connector, which is the essential building block for building systems in the DynaComm language. It consists of a glue g and a set of roles R_1, R_2, ... R_n, as shown in Figure 2.9. The set of roles can be instantiated with specific components of the system under construction, and the glue describes how the activities of the role instances are to be coordinated. θ_1, θ_2,...θ_n are "middle" designs to interconnect the glue and the roles with regulative superposition morphisms. In this thesis, we use the following graphical notations to represent categorical diagrams, where the rectangles refer to components, the circles represent cables (special components to be introduced later), and the morphisms between components are represented by the lines with arrows.

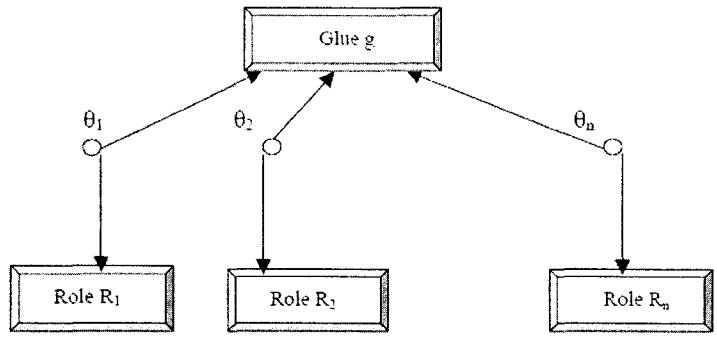


Figure 2. 9 Graphical notation of a connector

The generalization of the pushout operation to complex diagrams like this one is called a *colimit* and a category that admits (finite) colimits is said to be (finitely) *cocomplete.* From [12, page 77], we have the following proposition:

Proposition 2.10: A category C is finitely cocomplete iff it has initial objects and pushouts of all pairs of morphisms with common source.

In REG, the initial object will be the empty design (the set of channels and actions are empty) and as shown in proposition 2.8, for each pair of regulative superposition morphisms with common source we can derive the pushout. Therefore, we conclude that REG is finitely cocomplete. Meanwhile, it has been

shown in [18, page 187] that for any finite configuration diagram in REG, the colimit will exist. The semantics of configurations is given by the colimit of the underlying diagrams.

Actually, we can calculate the colimit of the configuration diagram in Figure 2.9 incrementally through a sequence of pushouts as below:

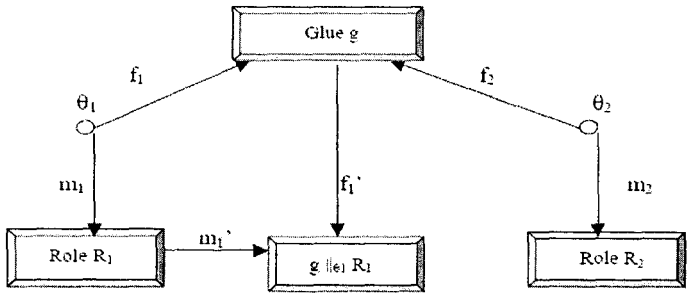


Figure 2. 10 Calculate the colimit of a connector

First we compute the pushout of g and R1 to get g $\|_{\theta 1}$ R1 and the morphisms f1' and m1' based on the method proposed in the above section. Then we can combine f2 and f1' (the composition law) to get a new regulative superposition morphism from θ2 to g$\|_{\theta 1}$R1, through which we can compute the pushout of R2 and g$\|_{\theta 1}$R1. Because glue g is a subcomponent of g$\|_{\theta 1}$R1 and the diagram is finite, we can repeat this procedure to obtain the colimit of the connector.

According to the defintion of connector instantiation in [18, page 200], when a connector is instantiated, each role of the connector will be instantiated by the role instance through a refinement morphism. To ensure that the diagram defined by the instantiation has a colimit, we need to show how refinement morphisms will work together with the regulative superpositions to give semantics to the instantiated connector.

First, we will clarify the notion of "middle" designs (such as θ_1, θ_2, ...θ_n) in the configuration digram of a connector. In a "middle" design C, we will only expect input channels, which can be used to interconnect designs. The reason is that output channels cannot be used in C to connect the input channel of one design with the output channel of another design, and it will make no sense to interconnect output channels of different designs. Also we set the enabling guard and progress guard of each action to true and set R(g) to "skip" (by "skip" we mean this action has no effects on the local channels of the design), which is enough to synchronize the actions. Generally a "middle" design of the above

form is called a *cable* in this thesis, containing only input channels and its actions having the following form g: true -> skip.

As we will show in the proof of proposition 2.11, it is crucial to have the notion of cables to interconnect the glue and the roles, in order to combine the regulative superpositions from the cables to the roles and the refinement morphisms from the roles to the role instances, thus ensuring that the instantiated connector has a defined colimit. The following discussion will establish this result.

Consider the case that in the configuration diagram of a connector, one of its roles R_i is instantiated by the role instance C_i through refinement morphism. Our intention is to show that there exists a regulative superposition from the cable θ_i to this instance, given the fact that there is a regulative superposition from θ_i to the role R_i and a refinement morphism from R_i to C_i, as shown in the following diagram:

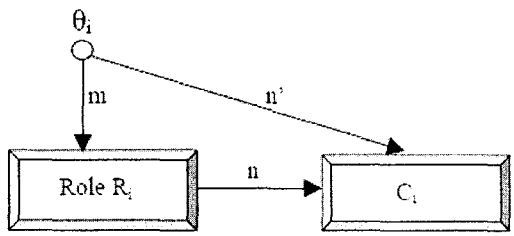


Figure 2. 11 Combine regulative superposition and refinement morphism

We have the following proposition:

Proposition 2.11: Suppose m is a regulative superposition morphism from cable θ_i to R_i, and n is a refinement morphism from R_i to C_i; there will exist a regulative superposition morphism n' from θ_i to C_i.

Proof:

The morphism n' is defined as:

* n'_α is a total function: for every channel v in θ_i, $n'_\alpha(v) = n_\alpha(m_\alpha(v))$.
* n'_γ is a partial mapping: for every action g in C_i, if $n_\gamma(g)$ is defined and $m_\gamma(n_\gamma(g))$ is also defined, $n'_\gamma(g) = m_\gamma(n_\gamma(g))$; otherwise, it is undefined.

Because both m and n are signature morphisms and SIG is a category, we know n' is a signature morphism. To show n' is a regulative superposition morphism,

we need to check the following conditions:
* $I_{Ci} \Rightarrow n'(I_{\theta i})$.
We have $I_{Ci} \Rightarrow n(I_{Ri})$, $I_{Ri} \Rightarrow m(I_{\theta i})$, so $n(I_{Ri}) \Rightarrow n(m(I_{\theta i})) \Leftrightarrow n'(I_{\theta i})$, and $I_{Ci} \Rightarrow n'(I_{\theta i})$.

* If $v \in loc(\theta_i)$, $g \in \Gamma_{Ci}$ and $n'_{\alpha}(v) \in D_{Ci}(g)$, then g is mapped to an action $n'_{\gamma}(g)$ and $v \in D_{\theta i}(n'_{\gamma}(g))$.
* For every $g \in \Gamma_{Ci}$ where $n'_{\gamma}(g)$ is defined, if $v \in loc(\theta_i)$ and $g \in D_{Ci}(n'_{\alpha}(v))$, then $R_{Ci}(g, n'_{\alpha}(v)) \Leftrightarrow n'_{\alpha}(R_{\theta i}(n'_{\gamma}(g), v))$.
Because θ_i only contains input channels, $loc(\theta_i)$ is empty, so these two conditions hold.

* $L_{Ci}(g) \Rightarrow n'(L_{\theta i}(n'_{\gamma}(g)))$.
* $U_{Ci}(g) \Rightarrow n'(U_{\theta i}(n'_{\gamma}(g)))$.
From our definition of "middle" design, $L_{\theta i}(n'_{\gamma}(g)) \Leftrightarrow$ true , $U_{\theta i}(n'_{\gamma}(g)) \Leftrightarrow$ true, so these two conditions hold.

Having this proposition, the diagram in Figure 2.11 can be replaced by a simpler diagram, which consists of the cable θ_i, the role instance C_i and the regulative superposition n' between them. Therefore, in the configuration diagram of a connector, we can combine each pair of regulative superposition and refinement morphisms and obtain a regulative superposition from the cable θ_i to design C_i, as shown in Figure 2.12. So, we obtain an updated configuration diagram of the instantiated connector containing only regulative superpositions, through which we are able to derive its colimit.

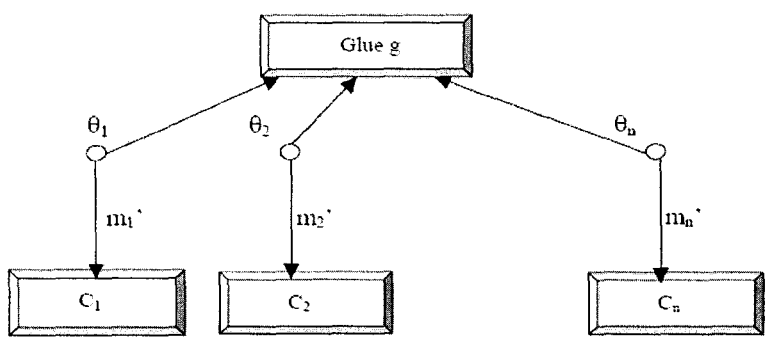


Figure 2. 12 Graphical notation of an instantiated connector

This result can be generalized to the case of a well-formed configuration diagram. In a well-formed configuration diagram, all the designs (components) are interconnected by cables through regulative superposition morphisms. From proposition 2.11, we have the conclusion that in a well-formed configuration diagram of a system, we can refine any subcomponents of the system and obtain an updated well-formed configuration diagram, through which the system's semantics can be derived from its colimit.

From the proof of proposition 2.11, it is not difficult to see that the composition of a regulative superposition and a refinement morphism may not give a regulative superposition, if θ_i is not a cable. So, it is crucial to enforce the designs to be interconnected by cables in a well-formed configuration diagram (connector is a special case), so that the colimit will exist after refining any of the designs in the diagram.

In proposition 2.11, we restrict the form of θ_i to ensure the combination of regulative superposition and refinement morphism will give a new regulative superposition. Actually, we can relax the form of θ_i and one condition of the regulative superposition and give a general condition for the combination to work.

Proposition 2.12: Suppose θ_i is a design with one restriction, that the enabling and progress guards of its actions are true, and in condition 3 of the definition of regulative superposition, we relax the formula to allow the underspecification to be reduced. Then suppose m is an updated regulative superposition from θ_i to R_i, and n is a refinement morphism from R_i to C_i, there will exist an updated regulative superposition n' from θ_i to C_i.

Proof:

The morphism n' is defined as:

* n'_α is a total function: for every channel v in θ_i, $n'_\alpha(v) = n_\alpha(m_\alpha(v))$.
* n'_γ is a partial mapping: for every action g in C_i, if $n_\gamma(g)$ is defined and $m_\gamma(n_\gamma(g))$ is also defined, $n'_\gamma(g) = m_\gamma(n_\gamma(g))$; otherwise, it is undefined.

Because both m and n are signature morphisms and SIG is a category, we know n' is a signature morphism. To show n' is a regulative superposition morphism, we need to check the following conditions:

* $I_{Ci} \Rightarrow n'(I_{\theta i})$.

We have $I_{Ci} \Rightarrow n(I_{Ri})$, $I_{Ri} \Rightarrow m(I_{\theta i})$, so $n(I_{Ri}) \Rightarrow n(m(I_{\theta i})) \Leftrightarrow n'(I_{\theta i})$, and $I_{Ci} \Rightarrow n'(I_{\theta i})$.

* If $v \in loc(\theta_i)$, $g \in \Gamma_{Ci}$ and $n'_\alpha(v) \in D_{Ci}(g)$, we know $n_\alpha(m_\alpha(v)) \in D_{Ci}(g)$, since n_α preserves the categories of channels, we have $m_\alpha(v) \in loc(R_i)$. So we know g is mapped to an action $n_\gamma(g)$ and $m_\alpha(v) \in D_{Ri}(n_\gamma(g))$. With the condition $v \in loc(\theta_i)$, we have $n_\gamma(g)$ is mapped to an action $m_\gamma(n_\gamma(g)) = n'_\gamma(g)$, and $v \in D_{\theta i}(m_\gamma(n_\gamma(g)))$.
Therefore, we have g is mapped to $n'_\gamma(g)$ and $v \in D_{\theta i}(n'_\gamma(g))$.

For every $g \in \Gamma_{Ci}$ where $n'_\gamma(g)$ is defined,
* If $v \in loc(\theta_i)$ and $g \in D_{Ci}(n'_\alpha(v))$,
since n is a refinement morphism, we have $m_\alpha(v) \in loc(R_i)$, $n_\gamma(g)$ is defined, $g \in D_{Ci}(n_\alpha(m_\alpha(v)))$, so
$R_{Ci}(g, n_\alpha(m_\alpha(v))) \Rightarrow n_\alpha(R_{Ri}(n_\gamma(g), m_\alpha(v)))$.
Because $v \in loc(\theta_i)$, $n_\gamma(g) \in D_{Ri}(m_\alpha(v))$, and $m_\gamma(n_\gamma(g))$ is defined, we will have
$R_{Ri}(n_\gamma(g), m_\alpha(v)) \Rightarrow m_\alpha(R_{\theta i}(n'_\gamma(g), v))$.
So, $n_\alpha(R_{Ri}(n_\gamma(g), m_\alpha(v))) \Rightarrow n_\alpha(m_\alpha(R_{\theta i}(n'_\gamma(g), v)))$
$\Leftrightarrow n'_\alpha(R_{\theta i}(n'_\gamma(g), v))$.
Therefore, $R_{Ci}(g, n'_\alpha(v)) \Rightarrow n'_\alpha(R_{\theta i}(n'_\gamma(g), v))$.

* $L_{Ci}(g) \Rightarrow n'(L_{\theta i}(n'_\gamma(g)))$.
* $U_{Ci}(g) \Rightarrow n'(U_{\theta i}(n'_\gamma(g)))$.
From our restriction of θ_i, $L_{\theta i}(n'_\gamma(g)) \Leftrightarrow$ true, $U_{\theta i}(n'_\gamma(g)) \Leftrightarrow$ true, so these two conditions hold.

2.4.5 The Producer-Consumer example

Now we want to specify the Producer-Consumer system discussed in section 2.3 in CommUnity. The syntax of CommUnity has been introduced in section 2.4.1. It is assumed that basic data types such as nat, int, bool, string, list, array, and enumeration have already been defined in S. According to the system's requirements, a component design of the producer is as follows (for the list data type, operation * is used to add an element into the list):

```
design producer
out o_item: s
prv cur_item: s;
    item_q: list (s);
    st_g: bool;   // guard for storing the item
```

ack_g: bool // guard for awaiting the reply
init st_g = false ∧ cur_item = NULL ∧ ack_g = true ∧ item_q = NULL
do
prv prod [cur_item,st_g]: ¬st_g ,false -> cur_item' = random(s) ∧ st_g' = true
[] **prv** store [item_q, st_g]: st_g ,false -> item_q' = item_q * cur_item ∧ st_g' = false
[] send [o_item,ack_g,item_q]: ack_g ∧ item_q <> NULL , false -> o_item' = head(item_q) ∧ item_q' = tail(item_q) ∧ ack_g' = false
[] ack [ack_g] : true, false -> ack_g'= true

We assume that the item type s has already been defined in S. Private actions prod and store will produce the items and store them into the items queue (item_q), and the guard st_g is used to prevent the producer from producing a new item until the current item has been stored. Send action obtains an item from the head of item_q and puts it in the output channel o_item. Before the action ack is called, send action will not be executed again. The design of component consumer has the following form:

design consumer
in i_item: s
prv cur_item: s;
con_g: bool; // guard for consuming the item
re_g: bool // guard for replying producer
init con_g = false ∧ cur_item = NULL ∧ re_g = false
do
prv consume [re_g,con_g]: ¬re_g ∧ con_g ,false -> con_g' = false ∧ re_g' = true
[] extract [cur_item,con_g]: ¬con_g, false -> cur_item' = i_item ∧ con_g' = true
[] reply [re_g] : re_g, false -> re_g'= false

The producer's send action is intended to be synchronized with the extract action of the consumer, such that when the consumer wants an item, the producer will get it from the items queue. The consumer will not extract another item until the current item has been consumed by the consume action, and it will also send the reply to the producer after consuming the current item. Also the producer cannot send a new item to the consumer (by using the guard ack_g) until it receives the acknowledgement from the reply action of consumer, because ack_g can only be enabled in the ack action, which is synchronized with the reply action of consumer.

In order to achieve action synchronization and channel connections in CommUnity, special kinds of components, "cables", are defined, which are then connected to the interacting components through regulative superposition morphisms. In this example, we define a cable as follows:

design component cable
in v: s
do
sync1: true -> skip
[] sync2: true -> skip

The following configuration diagram gives the mapping of channels and actions between the cable and producer, consumer. The joint behavior of the configuration is then characterized by the colimit of this configuration diagram based on the pushout operation of Category Theory.

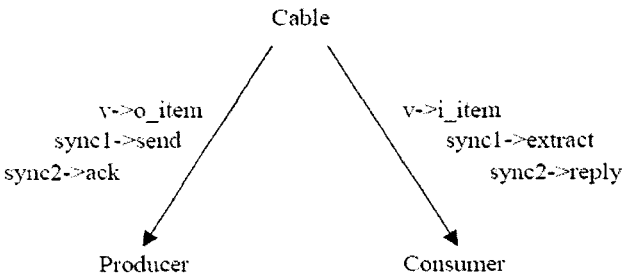


Figure 2. 13 Configuration diagram of Producer-Consumer system

Now we will consider the dynamism of the system, in which the consumer will be deleted and replaced by a newly created consumer if it does not work normally due to unexpected problems. By means of graph grammars, we are able to define an operation to delete the old consumer node and to create a new consumer node in the graph of the system's configuration and the operational semantics of reconfiguration can be obtained in this way [14]. However, in order to verify any properties of the reconfigurable system, some meta-language must be introduced to perform the reasoning [2].

2.5 Summary

In this chapter we have reviewed four mainstream ADLs to support the specification of dynamic software architectures. The reasons for extending the CommUnity language to support dynamic reconfiguration will be discussed in section 3.1. For the other three ADLs, we think they have the following shortcomings:

* Dynamic Wright

It has provided some dynamism for connectors, as we show in the Producer-Consumer example, to change the connections between the roles and components. However, the language only allows the connector to have a fixed number of roles and the glue cannot be changed, which has restricted the reconfigurability of the connector. In addition, Dynamic Wright does not support the hierarchical organization of system's architecture, due to the lack of language constructs for specifying composite components in Wright.

* Darwin

Darwin does not provide the language construct of connectors, and if we want to model the interaction between components, we can only simulate it by using components and ports. It will limit the language's ability to model and reuse the interaction patterns of the components.

* Dynamic Acme

Since the intention of Acme is to provide an interchange language, to relate different ADLs, it does not provide a fully defined semantics. Therefore, it will be difficult to reason about the properties of the specified systems within the language, because Acme does not consider the semantics of the different layers in a specification and their relationships in its first-order logic semantics [19].

As we will show in the following chapters, DynaComm has nice support for overcoming the above problems of these ADLs, thus serving as a competitive ADL for modeling the dynamic aspects of complex systems. In addition, based on Category Theory and the formal semantics being derived, in our future work, we will try to relate DynaComm specifications to first-order temporal logic [3], to analyze and reason about the specifications.

Chapter 3

The DynaComm language

We now start describing DynaComm, a new ADL defined as an extension of CommUnity, to support hierarchical design and dynamic reconfiguration in specifying complex, reactive and concurrent systems. First, the motivation for the extension of CommUnity is discussed. Then the syntax of the DynaComm language is defined to clarify the concepts introduced in [7] and provide constructs of DynaComm, such as components, connectors and subsystems, for the specification of large and hierarchical systems. We also propose the idea of interface manager to solve the synchronization problem in CommUnity, and the concept of population manager to manage the live instances of components in a subsystem, through which we can model complicated reconfiguration in a system. A dynamic client- server example is given in order to show how we apply the above constructs and ideas of DynaComm to specify the dynamism of the system, and to illustrate the design principles motivated by the different categories of morphisms between designs we discussed in the previous chapter.

3.1 The motivation of DynaComm

The main problems of CommUnity are the lack of support for specifying systems capable of modifying their architecture at run time and the lack of a coarse-grained construction unit, which can contain subcomponents, so that system specification will be organized in a hierarchical way. The solution of the second problem is essential for the support of hierarchical organization of systems, and the corresponding language construct, subsystem, will be proposed in the DynaComm language to solve this problem.

Dynamism is the main reason for our work on the extension of CommUnity. Actually, as we mentioned in the above section, graph grammars have been introduced into the CommUnity language, to trace the configuration change in the system's configuration graph during dynamic reconfiguration. However, we must use some other meta-languages to reason about the properties of a system's dynamically changing graph described by graph grammars, while some first-order temporal logic formalisms need to be related to CommUnity to verify the designs' properties. It will be very difficult to combine these different

formalisms to reason about the properties of systems specified in this way, especially when dealing with complicated dynamic reconfigurations in large and complex systems. We think the key problem is that CommUnity itself does not provide the mechanisms for talking about the architectural changes of the specified systems. Therefore, our intention is to incorporate the dynamic reconfiguration actions into the DynaComm language and provide the semantics based on CommUnity's semantic model to unify the static and dynamic parts of the language. Then in a standard way [6], some first-order temporal logic formalisms [4] can be related to the DynaComm language to reason about the properties of specified systems within the same level of the language, as shown in the research work of [6] to reason about the temporal properties of the designs in CommUnity.

To cope with the issues of the complexity and size of large-scale software, ADLs should provide a mechanism to scale an architecture by adding elements to the interior or boundaries of the system. It is expected that adding new components into a connector will not require the modification to this connector instance. Additionally, ADLs that allow a variable number of components to be attached to a single connector are better suited to scaling up than those that specify the exact number of components a connector can service [28]. We will propose the concept of interface manager in DynaComm to enable the connector to have the capacity of containing a variable number of components by instantiating a dynamic number of roles, which overcomes the use of static action synchronization in CommUnity.

3.2 Syntax

As reviewed in section 2.4, CommUnity only provides the concept of *designs* as a basic unit for the construction of systems and connector types to organize the coordination between components, through which the system's configuration diagram can be built. To support the hierarchical and incremental design of system, we will sharpen the concepts proposed in CommUnity and introduce a set of language constructs, such as component, connector and subsystem of DynaComm in this section.

3.2.1 Component

At earlier stages of system development, the architecture of the system is given

in terms of components that are usually abstractions of programs and can be refined into programs in later steps. In DynaComm, we propose the concept of Component as the encapsulation of isolated units of computation without internal structure. So, the notion of design as defined in CommUnity can be adopted for our purpose, with the following syntax:

design component component-name (parameters)
in var-name: type
out var-name: type
prv var-name: type
init init-expr
actions
[**prv**] action-name [var-name:type]: [L-expr, U-expr] -> R-expr
endofdesign

The definition of channels and actions is the same as for *designs* in CommUnity, and two extensions are introduced into components as follows:

- Initial condition

The keyword **init** indicates the declaration of the initial condition of the component, which constrains the values of its local channels when a component instance is created. Init-expr is a first-order logic sentence defined on the local channels of the component to describe its initial state. It is worth mentioning that some previous versions of CommUnity supported the definition of initial conditions, such as the ones we require.

- Parameters of the component

In the constructor methods of a class in most object-oriented languages, parameterization is a key mechanism to provide the features of multiple inheritance and polymorphism. We introduce parameters into component definitions to facilitate the design and development of systems. As for the definition of channels in *designs* of CommUnity, we only allow the type of a component's channel to be from S, i.e., to be a basic type. However, the parameter of a component can have a value which is a type in S or a specific value in a type from S, as shown in the following two examples.

* The type of a parameter is some type from S

We may consider this parameter as an input channel of the component, with the restriction that its value will be fixed after the component is instantiated. It can appear in the initial condition and L(g), U(g) and R(g) of the actions. The signature of the component will include this parameter in V and it will be mapped to *in* by the tv function. For example, in the design of the component server below, the parameter *smode* can have value main or backup, which will be set when an instance of component server is created and indicates whether the server is in main mode or backup mode, such that it will provide different interfaces to the clients. (The keyword *enum* represents enumeration type, which is defined in S.)

design component server(smode: enum(main,backup))
in syn_in: int
out res_data: int;
syn_out: int;
mode: enum(main,backup)
prv stat: int;
res: bool //indicate if the server can send data to client
init mode = smode ∧ res = false ∧ stat = 0
actions
update[stat,syn_out]: mode=main, false -> stat' = stat + 1 ∧ syn_out' = stat+1
[] sync[stat]: mode = backup, false -> stat' = syn_in
[] accept[res]: mode=main ∧ ¬res, false -> res' = true
[] send[res_data,res]: res, false -> res_data' = stat ∧ res' = false
endofdesign

* The value of a parameter is a type in S

If we declare a parameter of a component to be of type S, the value of this parameter will be used to define the types of channels of the component. Therefore, we cannot define the signature of this component until its parameters of type S have been instantiated. Generally before we put this component into a configuration diagram, we will refine this kind of parameter by some type in S to instantiate the component.

For example, the parameter s is of type S in the following bounded buffer component, so this buffer can contain elements of any fixed type from S when it

is instantiated. The other parameter bound is of type integer from S, which defines the size of this buffer.

design component buffer(s:S, bound:int)
in i:s
out o:s
prv rd:bool;
b:list(s)
init rd = false
actions
put[b]: |b|<bound -> b'=b*i
[] **prv** next[o,b,rd]: |b|>0 ∧ ¬rd -> o' = head(b) ∧ b' = tail(b) ∧ rd' = true
[] get[rd]: rd -> rd' = false
endofdesign

When a component design C is defined, it indicates that we have defined a component type C. So, C is not a concrete component and we should instantiate it before putting it into a configuration diagram. The concept of population manager will be introduced later on in this chapter to manage the creation and deletion of instances of a component type in the system. For example, when we declare c1: C in a subsystem, we are implicitly using such a create action of the population manager defined in the subsystem to create a live instance c1 of C.

For the user of a component or other components, which need to interact with this component, the main concern is the interface of the component, such as its public channels and the actions that can be seen from the outside. We will use the following graphical notation to represent a component along with its interface, and the signature of this component can be obtained from its graphical representation.

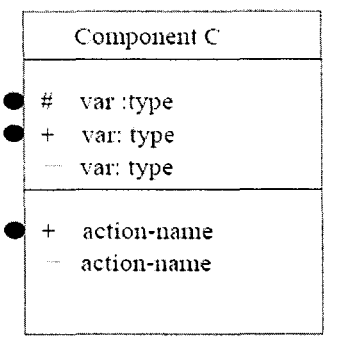


Figure 3. 1 Graphical notation of component in DynaComm

A component type is depicted as a box, with two sections for channels and actions. In the channels section, we use # to represent input channels and + or − refers to output or private channels, respectively. Similarly, + represents public actions and − means private actions in the section for actions. To define the interface of a component, filled circles are put next to the channels and actions exposed to the outside of the component. The interface of a DynaComm component will correspond to its input and output channels, and public actions. It is important to notice that a notion of hiding and interface, similar to the one we are using here, is already present in standard CommUnity. This is provided in CommUnity by the "local" keyword for channels, and "private" keyword for actions.

To support dynamic reconfiguration in the specification of systems, and possibly for other applications, we consider that the following extensions should be made to the definition of components in DynaComm.

- Parameters for actions (Schema actions)

As we specify the dynamic client-server system later on in this chapter, we find it will be convenient to have parameterized actions in DynaComm to specify reconfiguration actions, which is not supported by CommUnity. For example, we can define action g as follows:

g(index:int): L(g), U(g) -> R(g)

Instead of using the notion of parameters in actions, we call this kind of action a *schema action* or *indexed action*, and the formal definition is as follows:

Definition 3.1 A schema action is a collection of actions with different identifications (the index). Each member of this finite set of actions has the same behavior, which can only be distinguished by the unique index. This index can appear in the enabling and progress guards, and the R(g) expression of the schema action g.

We need this extension for the specification of multiple interfaces of the same body of a component, which is essential for dynamic reconfiguration in DynaComm (especially for the design of interface manager in section 3.4.3). For

example, in the previous example of component server, we need to specify multiple interfaces of the send and accept actions for a dynamic number of clients.

For any schema action in a design, the corresponding set of actions should be finite, so that the signature of this design will be finite. This restriction is necessary and related to the finitely cocompleteness property of the category REG we work in. To ensure that the colimit will exist for a well-formed configuration diagram in REG, the signature of each design within the diagram must be finite. The finiteness restriction also explains the reason for not using the notion of parameterized actions in DynaComm, because a parameter can have a type from S, such as integer, so the set of actions could be infinite and the finiteness of the signature cannot be guaranteed.

To make the semantics of DynaComm consistent with CommUnity, the normalization procedure for indexed actions will be introduced in section 4.1 to replace them by "standard" actions, such that the semantic model of standard CommUnity designs can be applied to derive the semantics of DynaComm specifications with indexed actions.

- The actions of a component manager

We need a component manager M to create and delete instances of a component type C. Whether we need separate component managers for different component types within a system, or one population manager that can be put at the subsystem level to manage all the instances of different classes is a design choice, which will be discussed in the population manager section. In both cases, parameters will be required in its actions to indicate the instances being created or deleted, and predicates to indicate the status of instances (alive or dead) should also be introduced.

For example, we want to define an action to delete an instance of component type server in its component manager. In order to indicate the status of the instance, we will use the component type as a predicate, e.g. ¬Server'(s1); this primed predicate means that s1 will not be a live instance of server after the execution of delete_server action.

Then the delete action can be defined as follows, and the corresponding propositions for predicates Server and Server' will be defined in section 4.1.1.

delete_server(y:NAME): Server(y), false -> ¬Server'(y)

However, it will be a bad design choice to put component creation and

deletion actions inside the component. The reason is that it will cause the self-reference problem when we use an action of a component to create an instance of it, and it is difficult to give the semantics of this component using the CommUnity categorical approach.

3.2.2 Connector

Connectors are components specialized in implementing interactions, which separate the computational part of a component from its interaction with its environment. A component interaction pattern is encapsulated into the connector, so that the same pattern can be applied in different contexts. CommUnity does not provide the syntax for connectors, which is essential for the specification of complex systems in DynaComm and ADLs generally. However, it proposes the concept of connector type defined by a set of roles that can be instantiated with specific components of the system under construction, and a glue specification that describes how the activities of the role instances are to be coordinated. Based on this concept, we define the syntax of a connector as follows:

design connector connector-name (parameters)
[**refines** connector-name]
[**attributes [prv | in | out]** var: type]
[**constraints** C-expr]
glue component-name (parameters) | subsystem-name (parameters)
[**refines** component-name | subsystem-name
connections
< component-name.var | subsystem-name.var **to**
component-name.var | subsystem-name.var >
< component-name.action | subsystem-name.action **to**
component-name.action | subsystem-name.action >]
< role component-name(parameters) | subsystem-name(parameters)
[**refines** component-name | subsystem-name
connections
< component-name.var | subsystem-name.var **to**
component-name.var | subsystem-name.var >
< component-name.action | subsystem-name.action **to**
component-name.action | subsystem-name.action >]
connections
< component-name.var | subsystem-name.var **to**

component-name.var | subsystem-name.var >
< component-name.action | subsystem-name.action **to**
component-name.action | subsystem-name.action >
>
init init(V)
actions
< [**prv**] action-name (parameters): [L-expr,U-expr] -> R-expr >
endofdesign

We use notation < > to indicate that the definition inside it can be repeated any number of times, and [ ] to indicate that the corresponding declaration is optional. The definition of connectors can be related to association classes in the Unified Modeling Language.

As an optional choice, attributes and constraints can be defined in a connector specification to enable it to specify a general architectural pattern, which will be inherited by the subsystem instantiating the connector.

An attribute can be private, input or output, as the definition of channels in a component. C-expr is a formula in first-order temporal logic [27], which constrains the architectural evolution of the connector's configuration. A connector contains a glue, a finite set of roles and the connections between the glue and roles, where the connection between each role and the glue is given by the synchronization of channels and actions between them, for which we can define a "middle" component (a cable) and corresponding regulative superpositions from it to the glue and the role. Roles define a minimum requirement on components (to instantiate the roles) and their behaviors to be plugged into the connector. Each role can refine another already defined component or subsystem, in which the refinement morphism is given by the mapping of the channels and actions between the components or subsystems.

When we define the connector, it is assumed that a set of predicates is built into it to decide if the roles have been instantiated by the corresponding role instances. (We will discuss these predicates in section 4.1.2.1.) ROLE-NAMES is the name space for the roles of a connector, in which each role must have a distinct name even if they have the same type. We have the predicates with the following form:

Role_connected (role-name: ROLE-NAMES, role-instance: role-type)
Role_disconnected (role-name: ROLE-NAMES, role-instance: role-type)

For example, assume we have defined components client, server and buffer, as well as the connector CBS which connects client and server (roles) through the buffer (glue). Then predicate Role_connected(client: ROLE-NAMES, c: Client) will indicate if the role client has been instantiated by the client instance c.

In the case that a dynamic number of instances of a component type will connect to the glue, which indicates that a set of roles of the same type should be defined, we need an interface manager to specify the connections to overcome the problem of incorrectly synchronized actions in CommUnity, which will be discussed in detail in section 3.4. This fixed set of roles will be defined as an array of the component type and we can distinguish them by the indices.

In the above discussion, we assume the connector has fixed glue and specification of roles, so that the dynamic reconfiguration will take place at the subsystem level by dynamically attaching or detaching role instances. If we also need to change the glue and roles of the connector, for example, the capacity of buffer should be enlarged in connector CBS, actions connect and disconnect should be defined to specify the corresponding change of connections between the glue and roles (see section 4.1.2.1). Again, they are indexed actions and can be transformed into "normal" actions by the procedure being introduced in section 4.1.2.

The parameters of a connector can have the same types as for component specification. To support the use of higher-order connectors proposed in [7] to specify *aspects* as architectural transformation patterns, we also allow connectors to act as parameters of a connector specification. For example, in [7], the higher-order connector Monitoring is defined as:

design connector Monitoring (AsyncSR(s,k), msg, s)
glue Observer-Buffer-Mpass (msg)
role Client
role Monitor(s)
role Server
endofdesign

Notice that AsyncSR(s,k) is a connector, which is passed to the glue Observer-Buffer-Mpass (msg) as a parameter:

design connector Observer-Buffer-Mpass(msg)
refines AsyncSR(s,k)
glue Buffer(s,k)
role Observer(msg) refines Sender(s)
role Mpass(msg) refines Receiver(s)
endofdesign

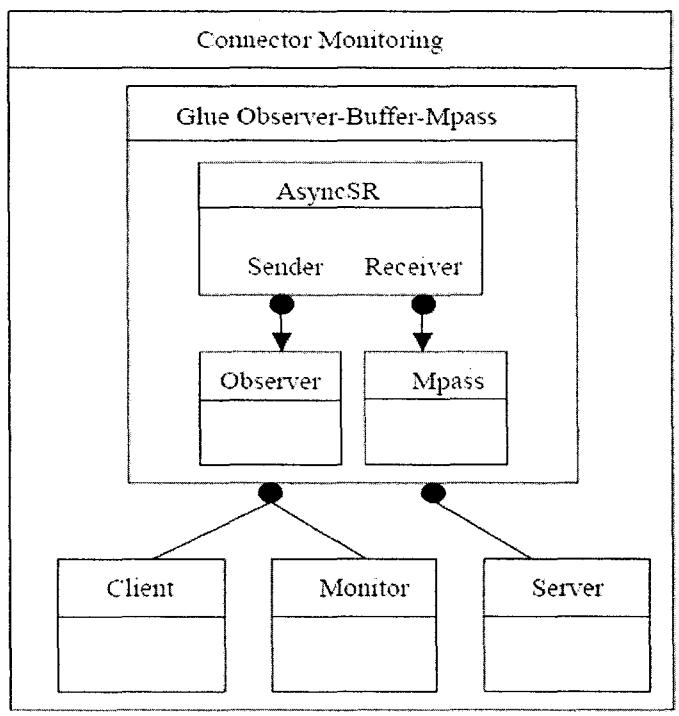


Figure 3. 2 Higher-order connector Monitoring

In our view, the glue is a component or subsystem, which specifies the interactions between the associated roles. Therefore, it has a tight relationship with the roles it will coordinate. However, we also can associate different sets of roles with the same glue for different interaction patterns, and thus define different connectors. Roles are explicit in a connector, which represent the interface of the connector. Meanwhile, the semantics of a connector can be obtained from the colimit of its configuration diagram.

However, a connector is not a concrete subsystem and its glue or roles must first be instantiated in the specification of systems. We consider it as the essential building block for building subsystems in a hierarchical way, organizing the complicated interactions between the components of a subsystem. As shown in the dynamic client server example of this chapter, we need connectors for the construction (or decomposition) of subsystems in an appropriate way.

3.2.3 Subsystem

In DynaComm, subsystems are coarse grained components, which are considered as the basic unit for the construction of systems. Intuitively, one might think of subsystems as configurations of simpler components, which due to reconfiguration can be dynamically modified. They can be used to combine the instances of components, connectors and other subsystems. When we define a subsystem, a schema of the subsystem is also defined, which means the instances of this subsystem can be used to construct other subsystems. The syntax of a subsystem specification is as follows:

design subsystem subsystem-name (parameters)
associations connector-name(parameters) |
    <**component** component-name | **subsystem** subsystem-name>
    morphisms
    < component-name | subsystem-name **to** component-name |
    subsystem-name
    connections
    < component-name.var | subsystem-name.var **to**
     component-name.var | subsystem-name.var
      component-name.action | subsystem-name.action **to**
     component-name.action | subsystem-name.action >
    >
participants < component-instance: component-name |
    subsystem-instance: subsystem-name >
[**attributes** [**prv** | **in** | **out**] var: type]
[**constraints** C-expr]
interface
    var **to** var. component-name | subsystem-name

action **to** action. component-name | subsystem-name
init init(V)
actions
< [**prv**] action-name (parameters): [L-expr,U-expr] -> R-expr >
endofdesign

The associations section in the subsystem definition will specify the possible relationships between the participants (components or subsystems). When the interaction between the components is complicated, we usually define the associations by a connector (which encapsulates the definition of the interaction). In the case that the relationships between the participants are simple and straightforward, we just list the components and subsystems as well as the morphisms between them explicitly. The morphisms are defined between components and subsystems, indicated by the **morphisms** key word, and the connections section will give the synchronization of channels and actions.

Since we use subsystem as a coarse grained component for the construction of a large, hierarchical system and it may contain a (dynamic) number of components and subsystems, we need a notation of *interface* to restrict the access to the subsystem from other components outside it. From our discussion of the parallel composition of designs in section 2.4, the coordinated actions of components and subsystems can be considered as joint actions of their pushout, which provides us the way to declare the public actions of interconnected subcomponents in the interface of the subsystem, through appropriate selection and renaming, where the input channels of subcomponents that have not been connected with channels of other subcomponents must be included in the interface. In addition, the public (input/output) channels and reconfiguration actions declared in the subsystem will be included in its interface too. The graphical notation for the interface of a subsystem is shown next, with an example illustrating how we select the joint actions.

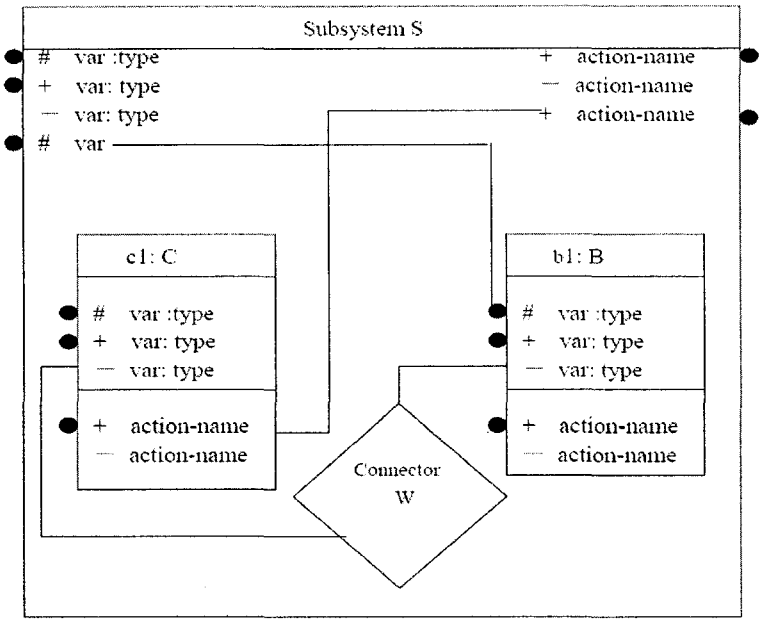


Figure 3. 3 Graphical notation of subsystem in DynaComm

In the first section of the subsystem diagram we include the attributes and reconfiguration actions, where input and output attributes along with the public actions specified in the subsystem are part of the interface. Within the subsystem the association is defined by a connector W, and the roles are instantiated by instance c1 of component type C and instance b1 of component type B. To export the interface of this configuration, we declare the interface containing one input channel and one action, which are linked to the corresponding input channel of b1 and public action of c1 (which might be synchronized with actions of other components in the system). If there are other live instances of a component type (say C) and we want to access one of them from outside the subsystem, a channel typed with C can also be declared here as a reference to one of the live instances with component type C. Again filled circles are placed next to the channels and actions to show the interface of the subsystem.

C can be used as a predicate to indicate whether this instance is a live instance of C in the subsystem. Actually, it will be appropriate to declare this kind of channel of type NAME in a subsystem, and indicate their status by using predicates such as C. Meanwhile, the semantics of a channel with type NAME can only be given for a certain state of the system, because it may represent the instance of different components or subsystems during the execution of this

system.

The parameters of an action can be of type NAME, which indicates that an instance of a component or subsystem is passed to this action to operate on. As we discussed before, this kind of action should be viewed as a schema action, which is indexed by the channels of type NAME. For example, the following action will create a server instance y with the desired mode, where two indices are used in this schema action. (Correspondingly, the mechanism to eliminate these indices will be discussed in detail in chapter 4, by means of which the semantics of CommUnity could be reused for DynaComm.)

create_server(y:NAME, mode:enum(main,backup)): ¬Server(y) -> Server'(y, mode)

Before the execution of this action, y is not a live instance of server; the primed predicate Server' means that y will be a live instance of server after the execution of the create_server action.

3.3 Population manager

If we want to incorporate dynamic reconfiguration mechanisms into the DynaComm language, it is necessary to have the constructs in the language to create and delete instances of components or subsystems dynamically. In other words, for a design in DynaComm, whether it is a component, connector or subsystem (we can consider it as a class), a class manager [12] will be generated to manage the instances of this class.

3.3.1 Design choice

The straightforward way for the design of class managers will be to generate independent class managers for different designs (classes), which take care of the creation and deletion of the corresponding instances. However, in spite of quite different implementations of various classes, the management actions are quite the same. For example, suppose we have multiple instances of client and server classes in our system, then the class manager for client will have actions to manage the name space while the same situation applies to class server. These two class managers will have the same set of actions for managing the name

space, which causes duplication.

Therefore, we may consider another design choice: have a component named M which has the common actions for managing populations of different classes and a concrete class C can be interconnected with M to generate the class manager of C. On the subsystem level, we introduce the population manager M to manage the name space of the subsystem, and for each class the actions to manage its live instances can be defined by using the common actions of M.

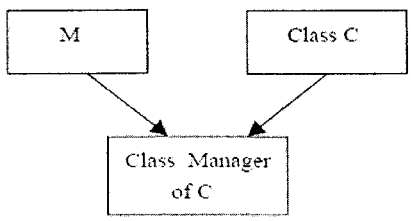


Figure 3. 4 Generating class manager

3.3.2 Our approach

We will define the name space of a subsystem, which consists of distinct names for instances of different classes in the system, because their names must be distinguished at the subsystem level. First we assume data type NAME has been defined, which contains all the distinct names that can be used in the system. Then we will include a channel in the subsystem, which is defined as a set of type NAME to represent the names of live instances of different classes in the system as follows:

prv S_NAME: set of NAME

Then the actions for managing the name space of the subsystem are defined and they are considered as common actions of the population manager, which will be called by population management actions of different classes in the subsystem. (The choose function will return an available name in the name space, which has not been included in S_NAME.)

assign(x: NAME): true, false -> x' = choose(NAME – S_NAME) $\wedge$ S_NAME' = S_NAME $\cup$ choose(NAME – S_NAME)

collect(x: NAME): true, false -> S_NAME' = S_NAME – {x}

Now, we can define the actions to manage live instances of each specific class in the subsystem. For example, a class S will have its population management actions as follows:

create_S(y: NAME): ¬S(y) , false -> assigned'(y) ∧ S'(y)
delete_S(y: NAME): S(y) , false -> ¬S'(y) ∧ collected'(y) ∧ ¬assigned'(y)

Notice that we use primed predicates assigned'(y) and collected'(y) in the above actions, which can be viewed as the short hand for the propositions specified in the assign and collect actions. In the delete_S action, the predicate ¬assigned'(y) is redundant because its meaning is equivalent to that of the predicate collected'(y).

The population management actions (create_S and delete_S) are schema actions indexed by the channels of type NAME. We claim that these indexed actions can be considered as a short hand for a set of "normal" actions, in the sense of CommUnity, and the "normalization" procedure will be described in detail in section 4.1 to eliminate the indices and transforming the predicates (such as S, S', assigned' and collected') into normal propositions, so that the corresponding set of standard actions in CommUnity can be derived. Because we use a finite set of channels (of type NAME) to eliminate the indices, the finiteness of the population management actions is guaranteed.

3.4 The dynamic client–server system

Having the basic building blocks and constructions in DynaComm defined above, let us look at how a complex system can be built from components, subsystems and connectors.

We want to model a client-server system, in which a dynamic number of clients will request the data from the server. The server simply keeps an integer as the data required by the clients, which is increased by one regularly. When the server is down, a new server will be created and attached to the configuration as an active service provider.

The dynamics of the system is explicitly described by the server's failure triggering the change of system topology during its execution. In order to achieve this effect, we must introduce operations that characterize dynamic changes in the architecture. First, we need the operations to create, delete, attach and detach

instances of components in the subsystem. Second, events that can trigger a reconfiguration should be defined. In this example, we use the output channels of components to indicate their state, which can be considered as events that will be observed by other components or the subsystem. Finally, we give the specification of what reconfigurations are triggered by each event. We achieve this by using the events as guards for reconfiguration operations, which describes how the corresponding event triggers reconfigurations. The overall architecture of the system is shown in the diagram below:

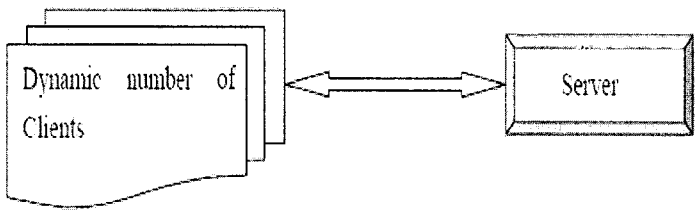


Figure 3. 5 Dynamic Client-Server system

3.4.1 Basic components

The basic components in this dynamic client-server system are client and server.

design component client
in data : int
prv stat: int;
req: bool
init req = false
actions
send_req [req]: ¬req, false -> req' = true
[] receive [stat,req]: req, false -> stat' = data ∧ req' = false
endofdesign

The client component simply generates the request to the server by means of the send_req action, which will be synchronized with some action of the server to accept the request. Private channel req is used to prohibit a client sending another request before it receives a response. Action receive will store the value of input channel data into its own channel stat, which stores the data requested by the client. The interface of the client component is the input

channel data, and the actions send_req and receive.

design component server
out res_data: int;
    down: bool //indicate if the server has gone down
prv stat: int;
    res: bool //indicate if the server can send data to the client
init stat = 0 ∧ res = false ∧ down = false
actions
    update[stat]: true ,false -> stat' = stat + 1
[] accept[res]: ¬res ,false -> res' = true
[] send[res_data,res]: res, false -> res_data' = stat ∧ res' = false
// this event will trigger reconfiguration operations in the subsystem
[] **prv** godown[down] : ¬down, false -> down'= true
endofdesign

Component server has actions update, accept and send, which update its own state and respond to the requests from the clients. Currently it is designed to serve clients sequentially, which means it will not accept the request from another client until requested data has been sent to the current client (by using guard res). Action godown will be triggered when the server is going down. Notice that we use the output channel down to indicate the state of the server at an abstract design level, and the implementation of this server design will provide the mechanism for the server's failure behavior. The change of this channel can be observed by the subsystem to trigger the reconfiguration operations.

3.4.2 Subsystem MCServer serving multiple clients

To enable the server to respond to requests from multiple clients, we want to add channels and actions into the server component to record the id of the current client making a request. We achieve this goal by superposing a regulator mc-reg, which performs the recording task, onto the server component, making the design clean and modularized.

// Specification of the regulator which records client id
design component mc-reg
in in_id: int
out out_id: int
prv client_id: int
actions
a1[client_id]: true -> client_id' = in_id
[] a2[out_id]: true -> out_id' = client_id
endofdesign

// Specification of the cable to interconnect the server and the regulator
design component cable2
actions
sync1: true -> skip
[] sync2: true -> skip
endofdesign

Then we can define the subsystem MCServer using the configuration diagram below:

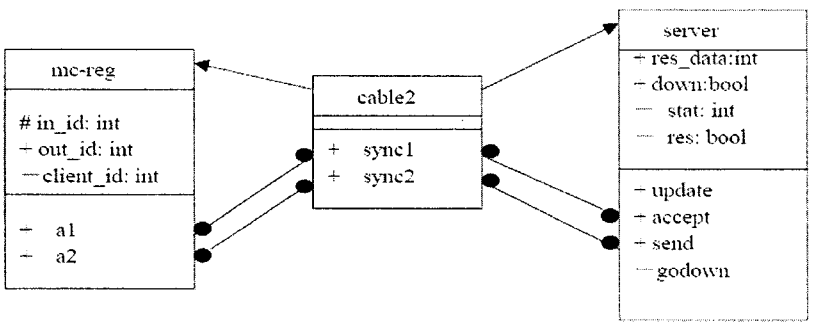


Figure 3. 6 Configuration diagram of subsystem MCServer

We use regular lines to indicate the mapping of actions and channels between the components, and the arrowed lines in the configuration diagram represent the regulative superposition morphisms defined by these associations. In subsystem MCServer, there are regulative superposition morphisms from cable2 to mc-reg and server, so we can compute the pushout of this diagram and derive its semantics. The specification of subsystem MCServer is as follows:

design subsystem MCServer
associations
component mc-reg, cable2, server
morphisms
cable2 **to** mc-reg
connections
sync1.cable2 **to** a1.mc-reg
sync2.cable2 **to** a2.mc-reg
cable2 **to** server
connections
sync1.cable2 **to** accept.server
sync2.cable2 **to** send.server
participants
r:mc-reg; c:cable2; s:server
interface
in
in_id **to** in_id.r
out
res_data **to** res_data.s
down **to** down.s
out_id **to** out_id.r
actions
update **to** update.s
accept **to** accept.s
send **to** send.s
init
Mc-reg(r) ∧ Server(s) ∧ Cable2(c) ∧ MCS(r,c,s)
endofdesign

Clearly the action a1 of regulator mc-reg will be synchronized with the server's action accept which will record the client's id when accepting the request of the client. Similarly when the server sends the data back to the client, action a2 of mc-reg will send the client id too. After the subsystem MCServer is defined, we can use the predicate MCS to indicate if the component instances in the subsystem are connected appropriately. Notice that in the interface part of the subsystem, we define the mapping between channels and actions declared in the interface to that of subcomponents inside the subsystem. If we want to refer to some participants of the subsystem, the mapping can also be declared in the

interface, such that the subcomponents can be seen outside the subsystem. A graph notation representing subsystem MCServer along with its interface and internal structure is shown as follows:

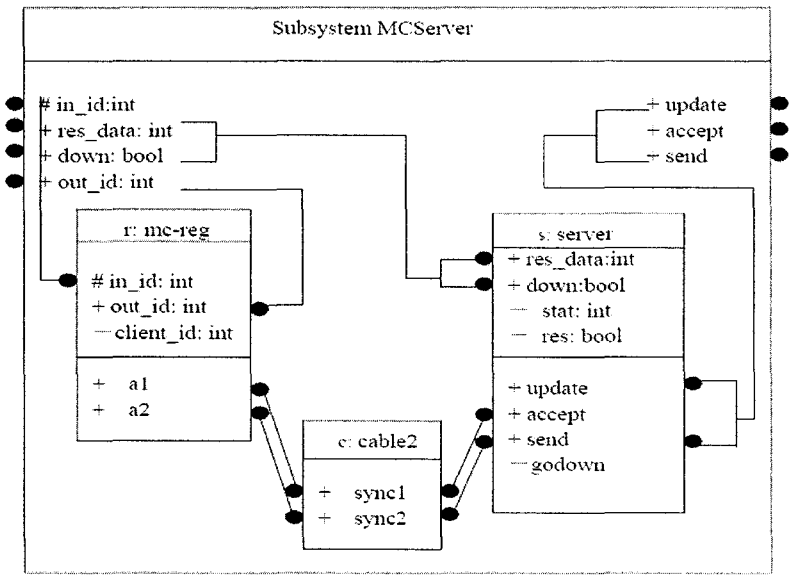


Figure 3. 7 Graphical notation of subsystem MCServer

Actually subsystem MCServer can be considered as a big component, which can be obtained by taking the pushout of the configuration diagram in Figure 3.6. The detailed calculation is described in section 2.4, when we review the categorical semantics of CommUnity. Here we show the result of the pushout operation, which is named as component MCServer1.

design component MCServer1
in in_id: int
out res_data: int;
down: bool;
out_id: int
prv stat: int;
res: bool;
client_id: int
init res' = false

actions

update[stat]: true, false -> stat' = stat + 1

[] accept[res,client_id]: ¬res, false -> res' = true ∧ client_id' = in_id

[] send[res_data,res,out_id]: res, false -> res_data' = stat ∧ res' = false ∧ out_id' = client_id

[] **prv** godown[down] : ¬down, false -> down'= true

endofdesign

3.4.3 Interface Manager: the regulator for MCServer

Having the ability to serve multiple clients with the mechanism to record the id of current client, another difficulty will appear when we try to connect a dynamic number of clients to subsystem MCServer by synchronizing send_req actions of different clients to the same action accept of MCServer. The problem of synchronized actions originates from the mechanism of action synchronization in CommUnity, where the synchronized actions must occur simultaneously. An example to show the negative effect of this problem is as follows:

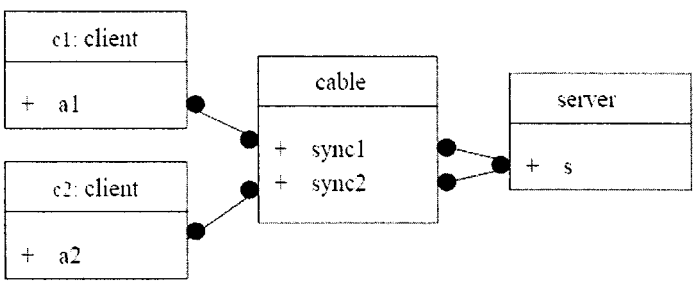


Figure 3. 8 The problem of synchronized actions in CommUnity

In the above diagram, action a1 of client1 and action a2 of client2 are synchronized with the same action s of server. Assume action a1 is executed, the server's action s will be synchronized, which will call action a2 of client2. Therefore, these two clients will not be able to communicate with the server independently. This is not what we expected, so a regulator should be introduced to provide multiple interfaces for action s of the server. We call this kind of regulator an *interface manager*, because it is designed to provide a set of actions (schema action) for each action of the service provider component, to

synchronize with the service request actions of a dynamic number of components, and we have determined that it can be derived in an automatic way from the example shown below. The idea of the interface manager is motivated by the object interface in [16], through which we are able to achieve independent synchronization between clients and server in this example.

If we consider the communication between a dynamic number of service request components (named as clients) and the component providing the service (named as server), there are two cases of the communication:

- The clients will send the requests to the server independently. The design of the interface manager is as follows:

design component client-interface
out out_id: int
prv req_id: int;
    ac: bool
init ac = false
actions
    C-accept(C-id:int)[ac,req_id]: ¬ac, false -> ac'=true ∧ req_id' = C-id
[] accept[out_id,ac]: ac, false -> out_id' = req_id ∧ ac' = false
endofdesign

When a client (which has C-id as its identity) sends a request to the interface manager, the corresponding schema action C-accept(C-id) will be synchronized and set the guard ac to false to disable all the interface actions (C-accept). They will not be enabled until the real accept action is executed, which sends the request to the server. Therefore, the clients can send the requests to the server independently, though sequentially, through the interface manager.

- The server will send the response to the corresponding client. We will design the interface manager as follows:

design component server-interface
in in_id: int
prv sg: bool
init sg = false
actions
C-send(C-id:int)[sg]: in_id' = C-id ∧ sg, false -> sg' = false
[] send[sg]: ¬sg, false -> sg' = true
endofdesign

The send action of the server is synchronized with the interface manager's send action, which will enable the guard sg for schema action C-send(C-id). Then the corresponding C-send action (with its index C-id equal to the in_id sent by the server) will inform the right client. Meanwhile, the interface manager will not respond to the server until the right C-send action is executed.

If both of the above communications will occur between the clients and server, we can combine the above designs and obtain the corresponding interface manager, which will be shown in the design of the server-reg component. The following diagram shows the connections between the clients, interface manager and the server.

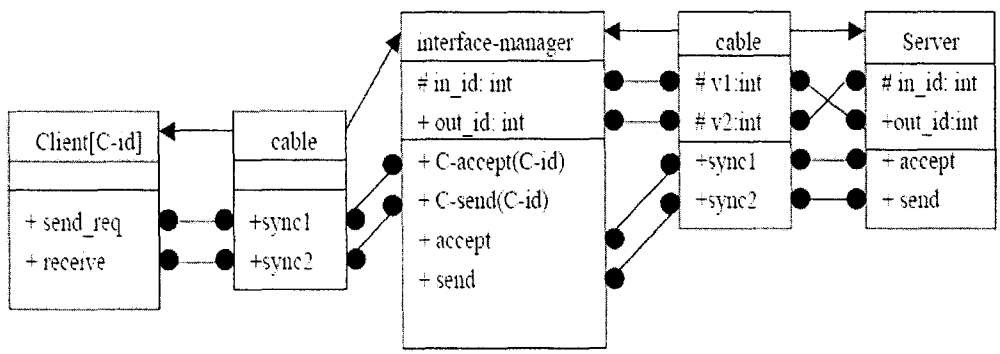


Figure 3. 9 The interface manager

Now we can define the regulator for MCServer as follows, which serves as the interface manager between the clients and MCServer.

design component server-reg
in in_id: int;
in_data :int
out out_id: int;
out_data: int;
cur_id: int // the recent assigned interface action id
prv req_id: int;
stat : int;
ac: bool;
sg: bool;
C-guard: array(bool)
init ac = false ∧ C-guard [] = false ∧ sg = false
actions
C-accept(C-id:int)[ac,req_id]: C-guard[C-id] ∧ ¬ac ,false -> ac'=true ∧ req_id' = C-id
[] C-send(C-id:int)[out_data,sg]: C-guard[C-id] ∧ in_id' = C-id ∧ sg, false -> out_data' = stat ∧ sg' = false
[] accept[out_id,ac]: ac, false -> out_id' = req_id ∧ ac' = false
[] send[stat,sg]: ¬sg, false -> stat' = in_data ∧ sg' = true
[] assign[cur_id,C-guard] : ¬C-guard.full, false -> cur_id' = C-guard.find ∧ C-guard'[C-guard.find] = true
[] collect(x: int)[C-guard]: C-guard[x], false -> C-guard'[x] = false
endofdesign

The graphical representation of the regulator server-reg is as follows:

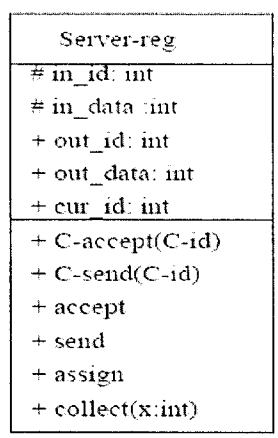


Figure 3. 10 The regulator for interface management

C-accept and C-send are schema actions, which represent a finite set of pairs of actions with the same behavior, one pair for each client connected to the server. We use array C-guard to store the guards for these actions and the index of the array corresponds to the index of the schema actions. C-guard is assumed to have a fixed number of entries, which is designated as *max_index*. Actions assign and collect are introduced as interface manager actions to assign or collect the available indices of C-guard, which is the C-id assigned to the client. C-guard[C-id] is initially set to false to indicate that the corresponding interface actions are disabled so that they can be assigned to new clients who want to connect to the server.

Channels req_id, in_id and out_id are used to record the client id so that the corresponding C-send(C-id) action will be called to send the data back to the right client. Output channel cur_id will store the recently assigned id of an interface action, by means of which the client will connect to the server.

Also we have actions accept and send in server-reg to synchronize with the corresponding actions of MCServer, which accept the request from the regulated server and send the requested data back to it. We assume the boolean array data type has the predicate full which indicates if all the entries are of value true, and the function find returns an index where the corresponding entry has value false, indicating that there is the capacity to accept a new client.

3.4.4 Connector DCS and subsystem DynamicCS

Having defined the interface manager server-reg to coordinate the interactions between a dynamic number of clients and the server, we are ready to specify a connector to incorporate this interaction pattern. The connector DCS is defined by the following configuration diagram, which has server-reg as the glue, a fixed set of clients and MCServer as roles. According to the requirement of the dynamic client-server system, a dynamic number of clients will be connected to the server. Therefore, we specify a fixed set of roles with type client in the connector, which will be instantiated dynamically when attaching or detaching the client instances.

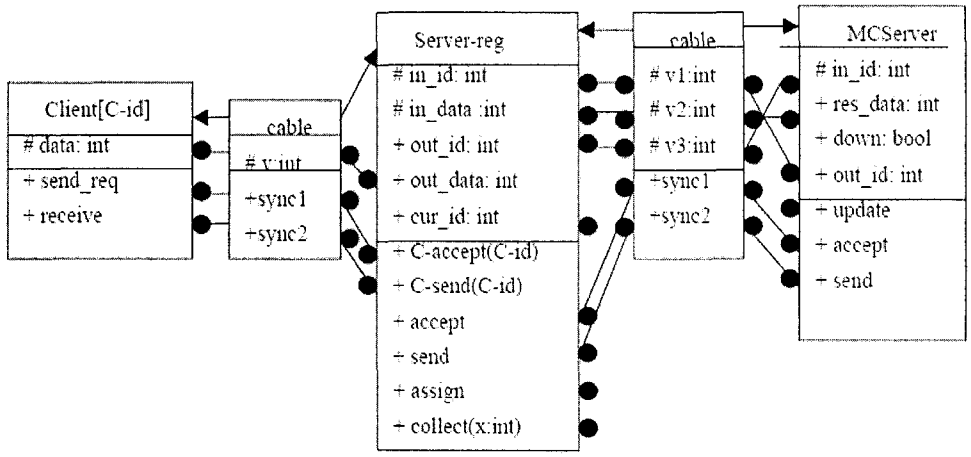


Figure 3. 11 Configuration diagram of connector DCS

To show the connections between the glue server-reg and its roles, we use an implicit cable to interconnect them, which can be generated automatically. Again, arrowed lines represent regulative superpositions from cable to the glue and roles, which is consistent with the definition of connector type in CommUnity, and the mapping of channels and actions are also given by the links between interfaces in the diagram. We show a schema of connections between a role client and the glue server-reg, where the same pattern will be applied to all the roles of type client. The corresponding specification in DynaComm is as follows:

design connector DCS
glue server-reg
role MCServer
connections
in_id.MCServer to out_id.server-reg
res_data.MCServer to in_data.server-reg
out_id.MCServer to in_id.server-reg
accept.MCServer to accept.server-reg
send.MCServer to send.server-reg
role client [max_index: nat]
connections
data.client[C-id] to out_data. server-reg
send_req.client[C-id] to C-accept(C-id). server-reg

receive.client[C-id] to C-send(C-id). server-reg
endofdesign

Having defined the interface of subsystem MCServer, we are ready to specify its connections with the glue in a straightforward way, where the subsystem is an encapsulation of its subcomponents and their interconnections and we can access its functionality through the interface. Clients are defined as a fixed set of roles (with max_index as the upper limit of its size) and the connections part specifies a schema for the morphisms between different clients with their corresponding interface actions. So, the configuration of connector DCS is fixed and dynamic reconfiguration is achieved by instantiating and uninstantiating the set of client roles dynamically. (We will give the static semantics for a certain state of the connector with dynamic reconfiguration capabilities in chapter 4.) It is easy to check that our design of the connector DCS has incorporated the dynamic interactions between the clients and the server, thus satisfying the requirements of the dynamic client-server system.

Finally, subsystem DynamicCS is defined by instantiating glue server-reg with its instance sr, role MCServer with its subsystem instance mcs and the set of client roles with a dynamic number of client instances.

design subsystem DynamicCS
associations DCS
participants
sr: server-reg; mcs: MCServer
attributes
prv
//the association between client instances and interface actions
R:list (<NAME,int>);
S_NAME: set of NAME;
new_mcs: NAME
interface
init
init Server-reg(sr) ∧ MCServer(mcs) ∧ DCS(sr, mcs)
actions
attach_client(x:NAME): Client(x), false-> sr.assigned' ∧
DCS.Role_connected'(client[sr.cur_id], x) ∧ R.inserted'(x, sr.cur_id)

[] detach_client(x:NAME): Client(x) ∧ DCS.Role_connected(client[R(x)], x),
false -> sr.collected'(R(x)) ∧ DCS.Role_disconnected'(client[R(x)], x) ∧

R.deleted'(x, R(x))

[]**prv** attach_mcserver(y:NAME): MCServer(y) ∧ DCS.Role_disconnected (MCServer, y), false -> DCS.Role_connected'(MCServer,y)

[]**prv** detach_mcserver(y:NAME): MCServer(y) ∧ DCS.Role_connected (MCServer,y) -> DCS.Role_disconnected'(MCServer,y)

[]**prv** change_mcserver(y:NAME): MCServer(y) ∧ y.s.down, false -> detached_mcserver'(y) ∧ deleted_mcserver'(y) ∧ created_mcserver' (new_mcs) ∧ attached_mcserver'(new_mcs)

[]**prv** create_mcserver(y:NAME) : ¬MCServer(y), false -> assigned'(y) ∧ MCServer'(y)

[]**prv** delete_mcserver(y:NAME) : MCServer(y), false -> ¬MCServer'(y) ∧ collected'(y)

[] create_client(x:NAME) : ¬Client(x), false -> assigned'(x) ∧ Client'(x)

[] delete_client(x:NAME): Client(x), false -> ¬Client'(x) ∧ collected'(x)

[]**prv** assign(x:NAME) : true, false -> x' = choose(NAME – S_NAME) ∧ S_NAME' = S_NAME ∪ choose(NAME – S_NAME)

[]**prv** collect(x: NAME) : true , false -> S_NAME' = S_NAME – x
endofdesign

In the above specification, we use list R(<NAME, int>) to record the relationship between connected clients and interface actions. Combined with the predicates for connectors we defined in section 3.2.2 and the interface management actions of server-reg, we are able to specify the reconfiguration operations for dynamically attaching and detaching client instances. For example, in the attach_client action, we first ask the interface manager sr to assign an available index of the schema actions (which has been connected to the client role of the same index), then the corresponding client role is instantiated by the client instance and the mapping between the index and the client instance is recorded in R.

Also dynamic reconfiguration actions (attach_mcserver, detach_mcserver

and change_mcserver) have been added to DynamicCS subsystem, which enables the subsystem MCServer to be replaced when it is going down. Notice that as population management actions, reconfiguration actions are also schema actions, where each live instance of the corresponding class will have this set of actions, and we can use the "normalization" procedure in section 4.1 to transform them into "normal" actions to derive the standard semantics as in CommUnity.

In the interface part of the subsystem specification, we declare an init action corresponding to the initialization condition of the subsystem. The behavior of adding a new client or removing an existing client can be achieved by calling a sequence of reconfiguration actions in the interface of the subsystem. (We will introduce the notion for sequencing of actions in section 4.1.3.) For example, if we want to add a new client into the system, actions create_client and attach_client can be called sequentially. The graphical notation for subsystem DynamicCS (for a certain state of the system) is as follows, with two clients connected to the connector DCS by instantiating the roles of clients through refinement morphisms, and the other role MCServer instantiated by a MCServer instance mcs.

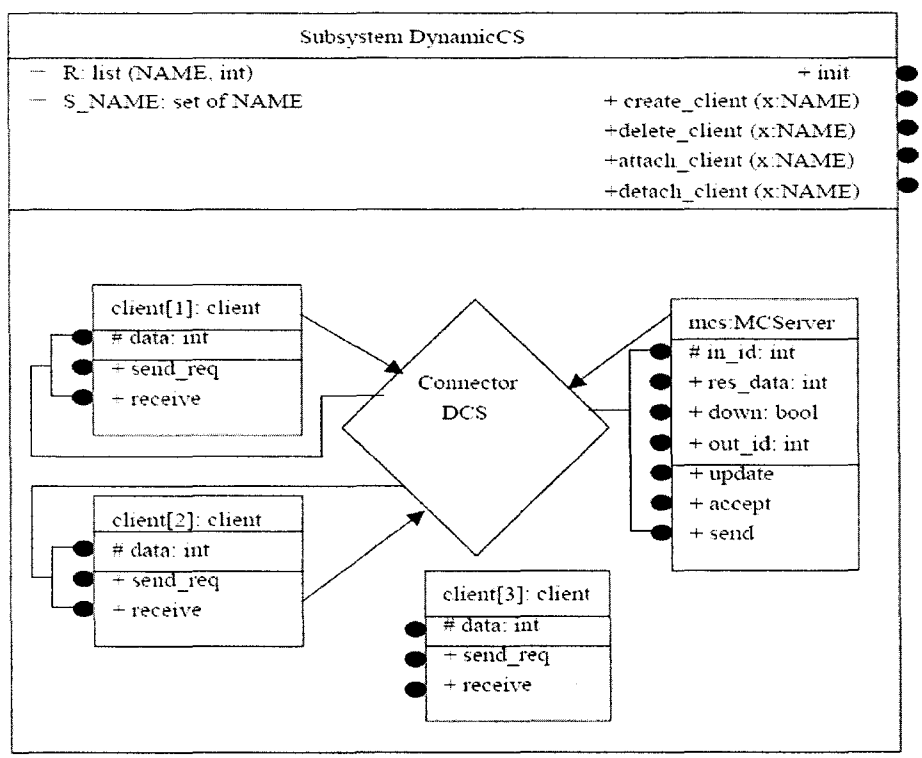


Figure 3. 12 Graphical notation of subsystem DynamicCS

From this example, we can view the construction of a system as applying connectors to components and subsystems in a hierarchical way so that it can be used to represent more general, hierarchically organized software architectures. When a system is decomposed, a connector will be at the highest level of its architecture.

3.4.5 Some Temporal Properties of DynamicCS

* Server-reg will keep the current client's req_id until the client's request, along with its id, has been sent to the server.

* The server will not accept another client's request until it has sent the requested data to server-reg.

* Server-reg will send the data back to the client (action C-send) before it receives the server's data again.

We will represent these properties in first-order temporal logic formula and translate DynaComm specifications into the corresponding logical specification so that they can be verified in a formal way, which will be investigated in our future work (chapter 6).

3.5 An improved dynamic client-server system

There is a problem with the above client-server system, where the data stored in the server (the channel stat) will get lost when it goes down and the newly created server may not be able to provide the correct data required by the clients. Therefore, we will design a backup server to synchronize its state with the main server and it will replace the broken main server without interrupting the service to the clients. Currently, we do not consider in our specification the case that the server and its backup both go down.

Since the backup server should be able to replace the main server and provide the same service to the clients, parameterized design will be required to achieve this effect. So, we propose a parameterized MCServer, which can switch its mode between main and backup, thus providing different interfaces

depending on the requirement. Again, we can design a regulator to incorporate the parameter (smode), and superpose it onto the original MCServer to obtain the parameterized MCServer.

design component mode-reg(smode: enum(main,backup))
in syn_in: int
out syn_out: int;
mode: enum(main,backup)
prv stat:int
init mode = smode ∧ stat = 0
actions
update[syn_out]: mode=main, false -> stat' = stat + 1 ∧ syn_out' = stat + 1
[] accept: mode=main, false -> skip
[] sync[stat]: mode = backup, false -> stat' = syn_in
[] switch[mode]: mode = backup, false -> mode' = main
endofdesign

We call this regulator mode-reg, which will provide a different interface according to its parameter smode. If smode has the value main, mode-reg will provide the update action to update the state of main server, and action accept to accept the request of the clients. It also has a private channel stat to be synchronized with the channel stat of MCServer through the synchronization of update actions, and the syn_out channel will output its stat to be read by backup server if the server is in main mode. On the other hand, sync action will set the server's state by the value of input channel syn_in when it is in backup mode, while the switch action will turn a backup server into a main server.

Then we can interconnect the regulator mode-reg with the MCServer, and obtain the parameterized MCServer (as a subsystem PMCServer) using the configuration diagram below:

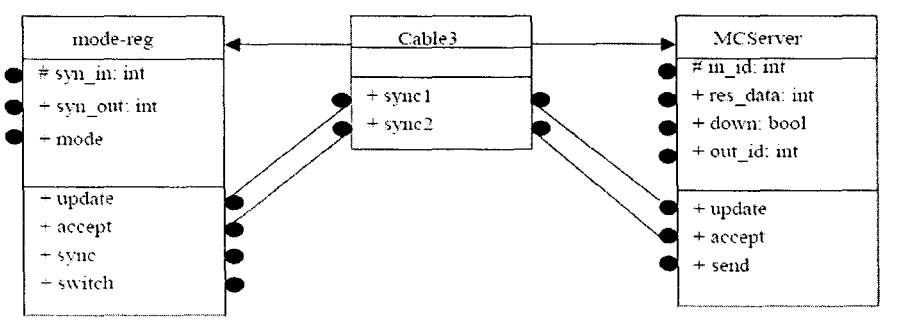


Figure 3. 13 Configuration diagram of subsystem PMCServer

The specification of subsystem PMCServer is as follows, where the mapping of channels and actions between the regulator and the MCServer is given by the associations section.

design subsystem PMCServer(smode: enum(main,backup))
associations
component mode-reg, cable3, MCServer
morphisms
cable3 **to** mode-reg
connections
sync1.cable3 to update.mode-reg
sync2.cable3 to accept.mode-reg
cable3 **to** MCServer
connections
sync1.cable3 to update.MCServer
sync2.cable3 to accept.MCServer
participants
mr:mode-reg; c:cable3; mcs:MCServer
interface
in
syn_in **to** syn_in.mr
in_id **to** in_id.mcs
out
syn_out **to** syn_out.mr
mode **to** mode.mr
res_data **to** res_data.mcs
down **to** down.mcs
out_id **to** out_id.mcs
actions
update **to** update.mcs
accept **to** accept.mcs
send **to** send.mcs
switch **to** switch.mr
sync **to** sync.mr
init Mode-reg(mr) ∧ MCServer(mcs) ∧Cable3(c) ∧ PMCS(mr,c,mcs)
endofdesign

The interface of the parameterized MCServer is declared as above and we

can figure out that the original interface of MCServer has been preserved when we set its mode to main, such that the clients of MCServer can use PMCServer (smode=main) instead, without perceiving any difference. The graphical notation for PMCServer is as follows, which illustrates the interface of this parameterized subsystem clearly: the interface of its subcomponent MCServer represents the interface of the main mode PMCServer, while if PMCServer is in backup mode, the channel syn_in and actions sync and switch of the regulator mode-reg will give its interface.

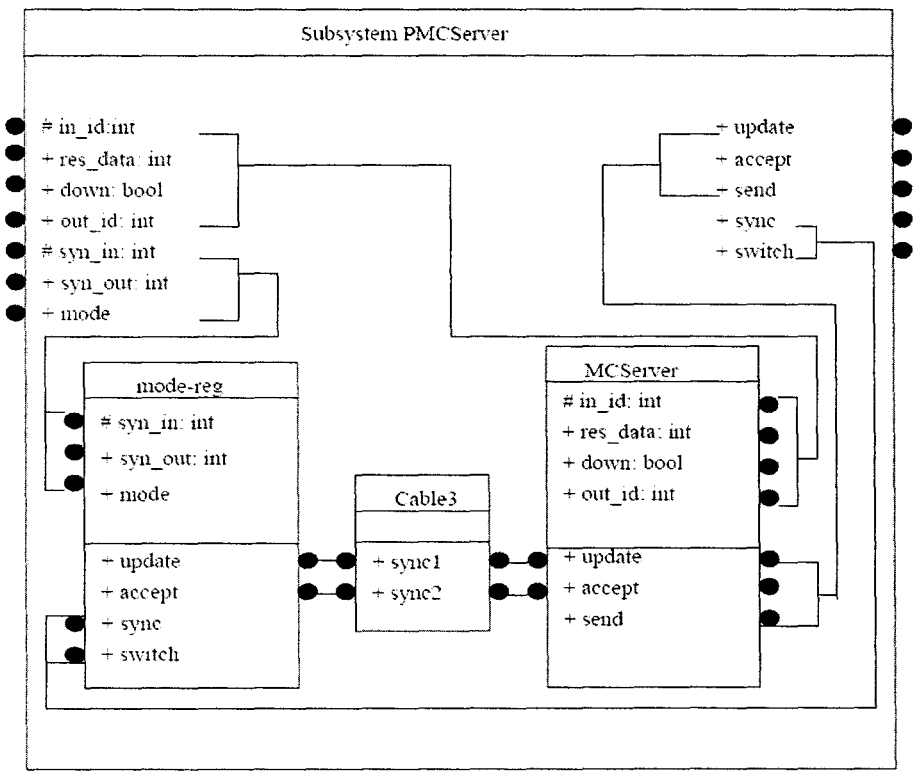


Figure 3. 14 Graphical notation of subsystem PMCServer

Having the parameterized MCServer, we can design the subsystem FT-MCServer with a fault tolerant capability by interconnecting a main mode MCServer with a backup mode MCServer through a cable, in order to make them synchronized on their internal states (the channel stat of main PMCServer and backup PMCServer). Meanwhile, we also specify the corresponding reconfiguration actions (change_mserver and change_bserver) to respond to the events such as main server is down or backup server is down. The configuration diagram of the fault tolerant server is as follows:

design component cable4
in v: int
actions
sync1: true -> skip
endofdesign

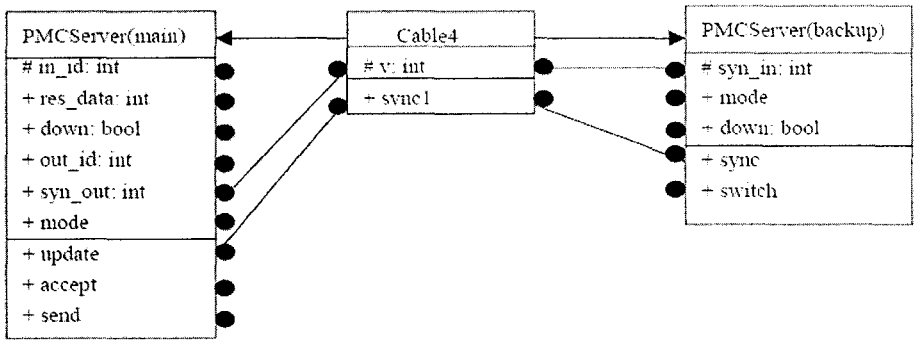


Figure 3. 15 Configuration diagram of subsystem FT-MCServer

The corresponding specification for the fault tolerant MCServer is given as below, and we notice that the interface part of this subsystem comes from the participant mcm, which is an instance of the main mode PMCServer.

design subsystem FT-MCServer
associations
component PMCServer(main), cable4, PMCServer(backup)
morphisms
cable4 **to** PMCServer(main)
connections
v.cable4 **to** syn_out.PMCServer(main)
sync1.cable4 **to** update.PMCServer(main)
cable4 **to** PMCServer(backup)
connections
v.cable4 **to** syn_in.PMCServer(backup)
sync1.cable4 **to** sync.PMCServer(backup)
participants
mcm:PMCServer; c:cable4; mcb:PMCServer
attributes
prv new_mcb: NAME
interface

init
input
in_id **to** in_id.mcm
output
res_data **to** res_data.mcm
out_id **to** out_id.mcm
actions
accept **to** accept.mcm
send **to** send.mcm
init
PMCServer(mcm, main) ∧ PMCServer(mcb,backup) ∧Cable4(c) ∧ FTMCS(mcm,c,mcb)
actions
// action to change the main MCServer
prv change_mserver(y:NAME): PMCServer(y,main) ∧ y.down, false -> detached_mserver'(y) ∧ deleted_pmcserver'(y) ∧ detached_bserver'(mcb) switched'(mcb) ∧ attached_mserver'(mcb) ∧ created_pmcserver'(new_mcb, backup) ∧ attached_bserver'(new_mcb)

// action to change the backup MCServer
[]**prv** change_bserver(y:NAME): PMCServer(y,backup) ∧ y.down, false -> detached_bserver'(y) ∧ deleted_pmcserver'(y) ∧ created_pmcserver'(mcb, ,backup) ∧ attached_bserver'(mcb)

[]**prv** attach_mserver(y:NAME): PMCServer(y,main), false -> FTMCS.Role_connected'(PMCServer(main),y)

[]**prv** detach_mserver(y:NAME): PMCServer(y,main) , false-> FTMCS. Role_disconnected'(PMCServer(main),y)

[]**prv** attach_bserver(y:NAME): PMCServer(y,backup), false -> FTMCS.Role_connected'(PMCServer(backup),y)

[]**prv** detach_bserver(y:NAME): PMCServer(y,backup) , false-> FTMCS. Role_disconnected'(PMCServer(backup),y)

[]**prv** delete_pmcserver(y:NAME) : PMCServer(y), false -> ¬PMCServer'(y) ∧ collected'(y)

[]**prv** create_pmcserver(y:NAME, mode:enum(main,backup)) : ¬PMCServer(y), false -> assigned'(y) ∧ PMCServer'(y,mode)

[]**prv** assign(x: NAME): true, false -> x = choose(NAME – S_NAME) ∧ S_NAME' = S_NAME ∪ choose(NAME – S_NAME)

[]**prv** collect(x: NAME): true , false -> S_NAME' = S_NAME – {x}
endofdesign

The population management actions for managing the live instances of subsystem PMCServer, and the reconfiguration actions to attach, detach or change the main and backup PMCServers are specified similarly as in the dynamic client-server system. Notice that all the actions are declared as private in the system, because the fault tolerant functionality is incorporated into the FT-MCServer subsystem and these actions should not be accessed from outside.

The graphical representation of subsystem FT-MCServer is as follows. As we mentioned before, the interface of this subsystem comes from an instance of main mode PMCServer. We have also explained that the interface of the main mode PMCServer will preserve the interface of MCServer, thus the clients of subsystem MCServer can connect to subsystem FT-MCServer and obtain the same service.

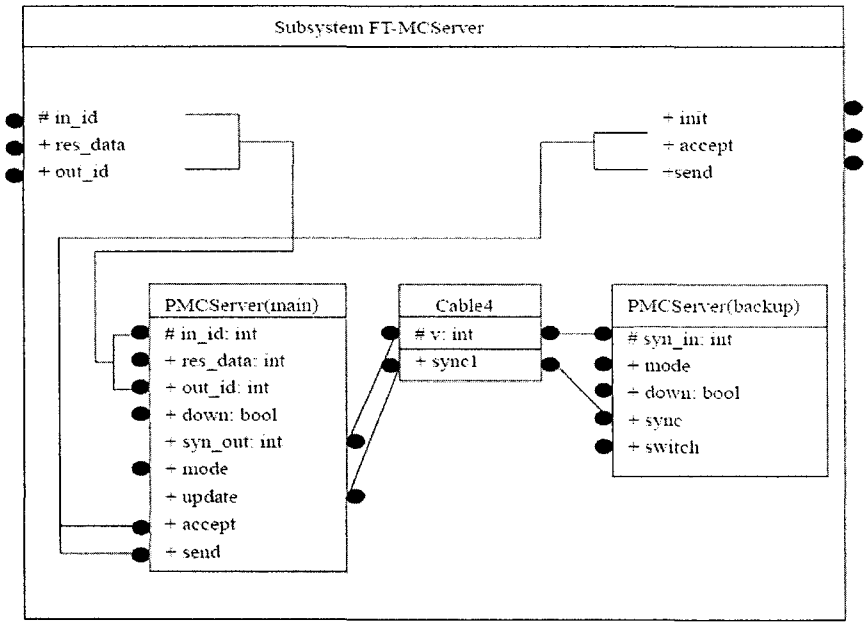


Figure 3. 16 Graphical notation of subsystem FT-MCServer

Now we will show that the role MCServer of connector DCS can be regulated to obtain the role FT-MCServer, which preserves the interface of MCServer. First, we superpose a regulator mode-reg(smode=main) onto MCServer to obtain subsystem PMCServer(main), so there is a regulative superposition morphism from MCServer to PMCServer(main). Then we create another subsystem PMCServer(backup) and interconnect it with subsystem PMCServer(main) to obtain subsystem FT-MCServer. From Figure 3.16, we can see that FT-MCServer provides the same interface as MCServer to the glue of DCS, so that it can replace the role MCServer in connector DCS. The corresponding change to the specification of connector DCS is shown as follows, where we only need to replace the occurrence of MCServer by FT-MCServer.

design connector DCS
glue server-reg
role FT-MCServer
connections
in_id.FT-MCServer to out_id.server-reg
res_data.FT-MCServer to in_data.server-reg
out_id.FT-MCServer to in_id.server-reg
accept.FT-MCServer to accept.server-reg
send.FT-MCServer to send.server-reg
role client [max_index: nat]
connections
data.client[C-id] to out_data. server-reg
send_req.client[C-id] to C-accept(C-id). server-reg
receive.client[C-id] to C-send(C-id). server-reg
endofdesign

Now we can derive the fault-tolerant dynamic client-server system with a small modification (actually simplification, since the reconfiguration actions to achieve the fault tolerant capabilities have been incorporated into the FT-MCServer subsystem) to the old specification, where only the population management actions and the attach and detach actions for the client are kept.

design subsystem FT-DynamicCS
associations DCS

participants
sr: server-reg; ftmcs: FT-MCServer
attributes
prv
R: list (<NAME, int>);
S_NAME: set of NAME
interface
init
init Server-reg(sr) ∧ FT-MCServer(ftmcs) ∧ DCS(sr, ftmcs)
actions

attach_client(x:NAME): Client(x), false -> sr.assigned' ∧ DCS. Role_connected (client[sr.cur_id], x) ∧ R.inserted'(x, sr.cur_id)

[] detach_client(x:NAME): Client(x) ∧ DCS.Role_connected(client[R(x)],x), false -> sr.collected'(R(x)) ∧ DCS. Role_disconnected'(client[R(x)], x) ∧ R.deleted'(x, R(x))

[] create_client(x:NAME) : ¬Client(x), false -> assigned'(x) ∧ Client'(x)

[] delete_client(x:NAME): Client(x), false -> ¬Client'(x) ∧ collected'(x)

[]**prv** assign(x: NAME): true, false -> x' = choose(NAME – S_NAME) ∧ S_NAME' = S_NAME ∪ choose(NAME – S_NAME)

[]**prv** collect(x: NAME) : true , false -> S_NAME' = S_NAME – x
endofdesign

The colimit of subsystem MCServer and mode-reg(main) will give the subsystem PMCServer(main). It is easy to show that there is a regulative morphism from MCServer to the subsystem PMCServer(main). After we superpose subsystem PMCServer(backup) to it and get FT_MCServer, there will still be a regulative morphism from MCServer to this new subsystem (the composition law of the category REG). Hence, in connector DCS there is a regulative morphism between the cable and glue Server-reg, as well as between the cable and FT-MCServer, respectively, through which we can calculate the colimit of the configuration diagram. Meanwhile, FT-MCServer can be viewed as a regulated role of MCServer in connector DCS and will provide the same

interface to the glue server-reg.

3.6 Summary of this chapter

We have defined the basic language constructs of DynaComm in this chapter and shown its suitability for specifying dynamic reconfiguration mechanisms in a reasonably big system, which contains complicated interactions, by the dynamic client-server system example. From its extended version, a fault tolerant dynamic client-server with the mechanism to provide consistent service when the main server is down, we can figure out the flexibility to refine a connector (by refining a role of a connector, we can obtain a refined connector) or regulate a connector to add new functionality to existing systems, which supports the incremental design principle. In the above example, we regulate the role server of connector DCS and obtain a new role FT-MCServer which provides the same interface to glue Server-reg. Therefore, we can modify the specification of connector DCS and subsystem DynamicCS by simply replacing all the occurrences of MCServer by FT-MCServer and derive the improved version of the old system in a structured and incremental way.

Chapter 4

The Semantics of DynaComm

In this section we will start to define the semantics of the DynaComm language. To make the discussion easier at first, we will only consider a static system with a fixed configuration diagram. More generally, it can be viewed as the semantics of the configuration diagram corresponding to a certain state of a system containing dynamic reconfiguration mechanisms, which is specified by DynaComm with reconfiguration actions.

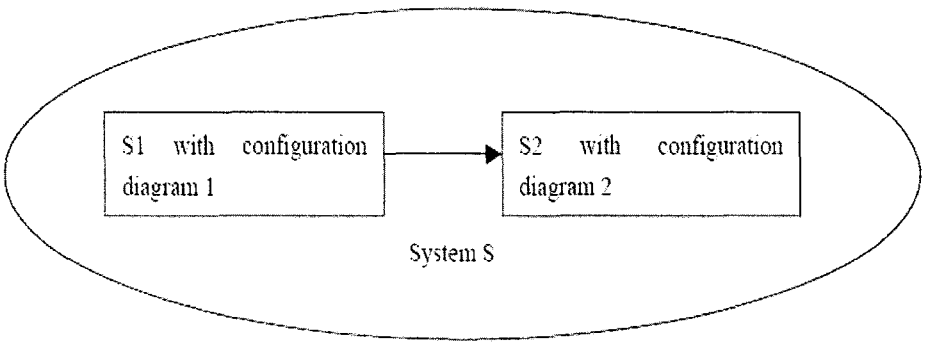


Figure 4. 1 The change of configuration in a dynamic system

For example, we have a system S with reconfiguration actions, which could change the configuration diagram of S. Therefore, it can be viewed as a transition system, which takes S1, S2, ... as worlds and reconfiguration actions as events to trigger the transition between the worlds. The configuration of the components and the connectors may be different in each of S1, S2, ... Currently we will consider the semantics of systems S1, S2, ... individually, instead of system S.

In the dynamic client-server example of chapter 3, we have introduced population management actions as well as other dynamic reconfiguration actions, which allow parameters (indices) in the actions to specify the system. This is a deviation from the philosophy of CommUnity, which enforces the action

coordination between the components by sharing the channels instead of through the parameters. In order to make DynaComm follow this principle, in section 4.1, we will show how to "normalize" these indexed actions and obtain the equivalent "pure" actions, in the sense of CommUnity, which do not have indices and only perform multiple assignments to their channels.

We have mentioned in section 2.4.4.3 that for any finite configuration diagram in REG, the colimit will exist and the semantics of the configuration is given by the colimit of its underlying diagram. Therefore, the semantics of a DynaComm specification (in a certain state) can be derived by transforming this fixed diagram into a flat configuration diagram (we will give this procedure in section 4.2), which consists of only basic components, cables and regulative superposition morphisms between them. Having the normalization technique to obtain CommUnity-like *designs* from the specifications of dynamic systems, we can consider the reconfiguration actions and related channels of the dynamic system specification as a high-level transition system, which defines new states and triggers transitions between different configuration diagrams. So, we are able to talk about the change in a dynamic system's configuration diagram within the DynaComm language, by using the syntax and semantic model of CommUnity. Meanwhile, in a certain state of the dynamic system, its configuration diagram is fixed and the semantics can be derived based on the above discussion.

4.1 The normalization of actions

We have defined the semantics of *designs* in CommUnity in section 2.4. Because we extend the syntax of actions in DynaComm, the "normalization" procedure will be introduced in this section to transform them into "pure" actions, in the style of CommUnity, such that the semantics of systems specified by DynaComm can then be derived.

4.1.1 The actions for population management

First we will look at the population management actions, which are used for the creation and deletion of the instances of components and subsystems in the system. For example, consider the instance creation and deletion actions of class C introduced in section 3.3.2, where a channel (index) of type NAME will be provided for the population management actions of C to indicate the reference to

the instance to be created or deleted. The general forms of the actions are:

create_C (x: NAME) : ¬C(x), false -> C'(x)
delete_C (x: NAME): C(x), false-> ¬C'(x)

To normalize these actions, first we need to give the explicit expressions corresponding to C(x) and C'(x). Then we will show how to remove the indices of these actions by introducing additional channels into the design of the subsystem.

When a subsystem contains a component or subsystem of type C, a finite set of objects with type C (objC: set of C) will be included in the subsystem, which represents the live instances of C. C(x) and C'(x) are the shorthand for the following expressions:

C(x): x in objC
C'(x): objC' = objC ∪ {x}
¬C(x): ¬(x in objC)
¬C'(x): objC' = objC – {x}

Suppose component type C is included in the subsystem S and type NAME has been defined for the name space of the live instances in S; the actions for managing the name space need to be introduced in S, as we have specified in section 3.3. In order to transform these indexed actions, we need to introduce private channels, such as C1 below. C1 will be used to record the reference to the newly created instance of C, by means of which we will be able to remove the indices of the creation and deletion actions of C.

prv C1: NAME
[] create_C1[C1, S_NAME]: true, false -> C1' = choose(NAME – S_NAME) ∧ S_NAME' = S_NAME ∪ {choose(NAME – S_NAME) } ∧ C'(choose(NAME – S_NAME))
[] delete_C1[S_NAME]: true, false -> ¬C'(C1) ∧ S_NAME' = S_NAME – {C1}

If there is at most one live instance of C in the subsystem at any moment, it is enough to have one channel C1 to record the reference to the live instance. However, when the class has multiple instances in the subsystem, a schema for

the creation and deletion actions (indexed by C-id) is defined below, which indicates a set of pairs of actions (schema actions) are defined in the system. The corresponding finite set of channels to refer to a dynamic number of live instances of class C should also be defined.

prv C : array(NAME);
[] create_C(C-id:int)[S_NAME]: true, false -> C[C-id]' = choose(NAME – S_NAME) ∧ S_NAME' = S_NAME ∪ {choose(NAME – S_NAME)} ∧ C'(choose(NAME – S_NAME))
[] delete_C(C-id:int)[S_NAME]: true, false -> ¬C'(C[C-id]) ∧ S_NAME' = S_NAME – {C[C-id]}

Notice that instead of using indices for the population management actions, we define a set of pairs of population management actions for the dynamic number of live instances of a class in the system we are designing. Correspondingly, a finite set of channels of type NAME is defined in the subsystem to refer to the live instances of this class. This finiteness restriction is necessary, because we must ensure that the signature of a design to be finite, as we discussed in section 3.2.1. The relationship between these channels and the name space of the subsystem is shown in the following diagram, where A1, A2, ... represent names of live instances in the subsystem.

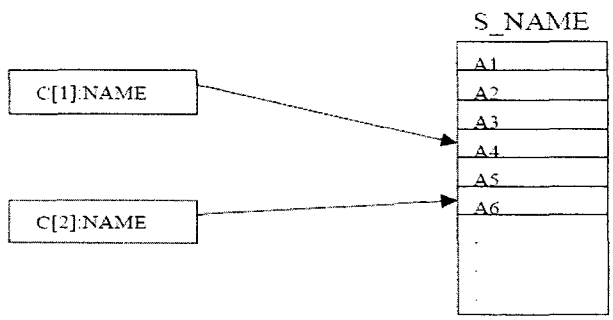


Figure 4. 2 The mapping between the name variables and the name space

4.1.2 Reconfiguration actions

Now we will consider the reconfiguration actions of the subsystem, which

change the configuration diagram of the system by putting new instances of components and subsystems into the diagram or removing the existing instances from the diagram. During the reconfiguration, the roles of the connector might be instantiated or uninstantiated. Therefore, the explicit actions corresponding to the predicates Role_connected and Role_disconnected should be given first. Then a general procedure for removing the indices of the reconfiguration actions will be introduced. As in section 4.1.1, we should distinguish between the two cases, where only one live instance of a class is in the system or multiple instances of a class will appear.

4.1.2.1 Predicates of the connector

We assume predicates DCS.Role_connected and DCS.Role_disconnected are defined within the connector DCS to manage the morphisms between the roles and their instances. When they appear in predicate R(g) associated with an action g, we should consider them as shorthand for a set of multiple assignments, which will change the connections between the roles and role instances. The types of the predicates are:

Role_connected (role-name: ROLE-NAMES, role-instance: role-type)
Role_disconnected (role-name: ROLE-NAMES, role-instance: role-type)

We will show how to define the corresponding multiple assignments to the channels in the subsystem containing the predicates. When the connector is defined, the set of morphisms (MPSet) between the roles and the role instances are also defined in the following form:

<role-name: ROLE-NAMES, set of <g1: G-NAMES, g2: G-NAMES> | <a1: A-NAMES, a2: A-NAMES>, role-instance: role-type, set of <g1: G-NAMES, g2: G-NAMES> | <a1: A-NAMES, a2: A-NAMES>>

G-NAMES and A-NAMES are the name spaces for the actions and the channels of the subsystem, respectively. The first set of tuples specifies the morphism between the glue and the role, while the second set of tuples represents the refinement morphism between the role and the role instance. When the connector is defined, the first set of tuples will be given, and the role-instance as well as the second set of tuples is set to NULL. We can see that MPSet contains all the information about the connection status between the glue,

the roles and the role instances during dynamic reconfigurations of the system, through which we will be able to construct the configuration diagram of the subsystem at any state. Meanwhile, the meaning of predicates Role_connected and Role_disconnected are given as follows, depending on where they appear in the actions.

- Role_connected (role-name: ROLE-NAMES, role-instance: role-type)

If it appears in the guards, we can derive the corresponding conditions by the description below:

Find the entry in MPSet by role-name and check if role-instance and the refinement morphism have been set.

If it appears in R(g), the corresponding multiple assignments can be obtained as follows:

Find the entry in MPSet by role-name and set role-instance as well as the refinement morphism by using the second parameter.

- Role_disconnected (role-name: ROLE-NAMES, role-instance: role-type)

If it appears in the guards, we can derive the corresponding conditions by the description below:

Find the entry in MPSet by role-name and check if role-instance and the refinement morphism are set to NULL.

If it appears in the R(g) expressions, the corresponding multiple assignments can be obtained as follows:

Find the entry in MPSet by role-name and set role-instance as well as the refinement morphism to NULL.

4.1.2.2 Attach and Detach actions

If there is only one live instance of class C in the subsystem, we know that it will correspond to at most one role of the connector (say CON), and we can refer to it by one name variable C1 in the subsystem. Therefore, the indices of the reconfiguration actions are not required. The attach and detach actions of class C can be defined as:

prv C1: NAME
attach_C: C(C1), false -> CON.Role_connected (C, C1)
detach_C: C(C1), false -> CON.Role_disconnected (C, C1)

Now we will consider the class C with a dynamic number of live instances in the system. Recall our discussion about the interface manager: a set of roles of type C will be defined in the connector CON and the glue sr: server-reg will contain a set of interface actions to connect with multiple instances, where a list R is introduced to record the mapping between the live instances of C and the indices of interface actions. For the dynamic reconfiguration actions attach and detach, we do not need to define them for every live instance of C. Instead, we assume these reconfiguration actions will happen serially so that two private channels are introduced to record the index (c_con) of C instance to attach to the connector, and the name of C instance to detach from the connector at a certain time, respectively.

prv c_con: int;
c_dcon: NAME;
R: list(<NAME , int>)

attach_C: C(C[c_con]), false -> sr.assigned' ∧ CON.Role_connected'
(C[sr.cur_id], C[c_con]) ∧ R.inserted'(C[c_con], sr.cur_id)

detach_C: C(c_dcon), false -> sr.collected'(R(c_dcon)) ∧ C[C.find]' = c_dcon ∧
CON.Role_disconnected'(C[R(c_dcon)], c_dcon) ∧ R.deleted'(c_dcon,
R(c_dcon))

In action detach_C, we assume C.find will return an index of array C, which is not referring to any live instance of C.

4.1.3 The sequence of actions

In the above example, the sequence of the multiple assignments in some actions must be ordered to ensure the right result. For example, in action attach_C the index of interface actions should be assigned before the instance can be put into

the configuration diagram. Therefore, a guard will be introduced to ensure the right order of the assignments, which is initialized to be true.

prv g1: bool
attach_C_1: g1, false-> sr.assigned' ∧ g1' = false
attach_C_2: ¬g1, false -> CON.Role_connected'(C[sr.cur_id], C[c_con]) ∧ R.inserted'(C[c_con], sr.cur_id) ∧ g1' = true

To make the design concise, we will introduce some syntactic sugar into DynaComm to express the sequence of multiple assignments in R(g):
action-name: [L-expr,U-expr] -> R-expr1; R-expr2;…;R-exprn

It will correspond to the following sequence of actions (the set of boolean variables are all initialized to be true):

prv g: array(bool)
action-name_1: g[1], false -> R-expr1 ∧ g[1]' = false
action-name_2: ¬g[1] ∧ g[2], false -> R-expr2 ∧ g[2]' = false
.
.
.
action-name_n: ¬g[1] ∧ ¬g[2] ∧ … ∧ ¬g[n-1] -> R-exprn ∧ g[1]' = true ∧ g[2]' = true … ∧ g[n-1]' = true

4.1.4 An example

To illustrate how to implement the above methods in the normalization of indexed actions, let us take the dynamic client-server example and consider the subsystem DynamicCS.

First, we will normalize the population management actions. Based on the discussion in section 4.1.1, we will get the following actions for subsystem MCServer, which only has one live instance in the system.

create_MCServer[mcs, S_NAME]: true , false -> mcs' = choose(NAME – S_NAME) ∧ S_NAME' = S_NAME ∪ {choose(NAME–S_NAME)} ∧ MCServer'(choose(NAME–S_NAME))

delete_MCServer[S_NAME] : true, false -> ¬MCServer'(mcs) ∧ S_NAME' = S_NAME – {mcs}

For component type client, which has a dynamic number of instances in the subsystem, which will instantiate a set of roles (of type client) of the connector DCS. The corresponding schema of population management actions are derived as below:

prv client : array(NAME);

create_client(C-id:int)[S_NAME]: true, false -> client[C-id]' = choose(NAME–S_NAME) ∧ S_NAME' = S_NAME ∪ {choose(NAME–S_NAME)} ∧ client'(choose(NAME–S_NAME))

delete_client(C-id:int)[S_NAME] : true, false -> ¬client'(client[C-id]) ∧ S_NAME' = S_NAME – {client[C-id]}

Then we come to the reconfiguration actions. For subsystem MCServer, it is straightforward to get the attach and detach actions as follows:

prv mcs: NAME

attach_MCServer: MCServer(mcs), false -> DCS.Role_connected'(MCServer, mcs)

detach_MCServer: MCServer(mcs), false -> DCS.Role_disconnected' (MCServer, mcs)

The action to change the server when it is going down can be specified as:

change_MCServer: mcs.down, false -> detached_MCServer' ∧
deleted_MCServer' ; created_MCServer' ∧ attached_MCServer'

For component client, the reconfiguration actions can be normalized following the second case in section 4.1.2.2:

prv c_con: int;
c_dcon: NAME;
R: list(<NAME , int>)

attach_client: Client(client[c_con]), false-> sr.assigned' ; DCS. Role_connected' (client[sr.cur_id], client[c_con]) ∧ R.inserted'(client[c_con], sr.cur_id)

detach_client: Client(c_dcon), false -> sr.collected'(R(c_dcon)) ∧ DCS. Role_disconnected'(client[R(c_dcon)], c_dcon) ; R.deleted'(c_dcon, R(c_dcon)) ∧ client[C.find]' = c_dcon

Finally, we can update the design of subsystem DynamicCS by introducing additional channels into the system specification, and eliminating the indices of the population management and reconfiguration actions. Notice that the channel MPSet is not modified explicitly by the actions in the specification. Actually the operations on MPSet are incorporated into the predicates of connector DCS (as we discussed in 4.1.2.1), and these predicates can be replaced by a sequence of standard actions. In this sense, the following specification of subsystem DynamicCS is partial and we use schema actions, the syntactic sugar for sequence of actions and the above predicates to reduce the length of the specification.

design subsystem DynamicCS
associations DCS
participants
sr: server-reg; mcs: MCServer
attributes
prv g: array(bool); //for the sequence of actions
client: array (NAME);

c_con: int;
c_dcon: NAME;
S_NAME: set of NAME;
MPSet: set of (<ROLE-NAMES, set of (<G-NAMES, G-NAMES> | <A-NAMES, A-NAMES>), NAME, set of (< G-NAMES, G-NAMES> | <A-NAMES, A-NAMES>) >);
R: list (<NAME, int>)

interface init

init

Server-reg(sr) ∧ MCServer(mcs) ∧ DCS(sr, mcs)

actions

attach_client: Client(client[c_con]), false-> sr.assigned' ; DCS. Role_connected' (client[sr.cur_id], client[c_con]) ∧ R.inserted'(client[c_con], sr.cur_id)

[] detach_client: Client(c_dcon), false -> sr.collected'(R(c_dcon)) ∧ DCS. Role_disconnected'(client[R(c_dcon)], c_dcon) ; R.deleted(c_dcon, R(c_dcon)) ∧ client[C.find]' = c_dcon

[]**prv** attach_MCServer: MCServer(mcs), false -> DCS.Role_connected' (MCServer, mcs)

[]**prv** detach_MCServer: MCServer(mcs), false -> DCS. Role_disconnected' (MCServer, mcs)

[]**prv** change_MCServer: mcs.s.down, false -> detached_MCServer' ∧ deleted_MCServer' ; created_MCServer' ∧ attached_MCServer'

[] create_client(C-id:int)[S_NAME]: true, false -> client[C-id]' = choose(NAME–S_NAME) ∧ S_NAME' = S_NAME ∪ {choose(NAME–S_NAME)} ∧ client'(choose(NAME–S_NAME))

[] delete_client(C-id:int)[S_NAME] : true, false -> ¬client'(client[C-id]) ∧ S_NAME' = S_NAME – {client[C-id]}

[]**prv** create_MCServer[mcs, S_NAME]: true , false -> mcs' = choose(NAME – S_NAME) ∧ S_NAME' = S_NAME ∪ {choose(NAME–S_NAME)} ∧ MCServer'(choose(NAME–S_NAME))

[]**prv** delete_MCServer[S_NAME] : true, false -> ¬MCServer'(mcs) ∧
S_NAME' = S_NAME – {mcs}
endofdesign

Compared with the original specification of subsystem DynamicCS, the sequence of actions are specified in some actions (e.g. attach_client) of the updated specification, and a boolean array g is included to transform the action sequence into a set of standard actions. The indices of the reconfiguration actions (attach_client, detach_client, attach_mcserver, detach_mcserver and change_mcserver) have been eliminated by the method introduced in section 4.1.2. For the population management actions of the client, the mechanism of schema actions has been applied to remove the indices of these actions, and provide a set of actions to create or delete the live instances of the client independently. In addition, the private actions assign and collect for managing the name space of the subsystem have been removed and incorporated into the population management actions of different classes, to show the complete specification of separate class manager actions.

It is worth mentioning that the length of the above specification will be increased quickly (n^2), when we try to transform it into a standard CommUnity design. The main factors contributing to this increase are the schema actions (population management) and the sequence of actions. However, since our work focuses on the theoretical aspect of the DynaComm language, we will not take the complexity issue into consideration at this stage and this problem will be investigated when we try to implement this language (as a development tool) and use it to model, develop and analyze real systems in some industrial setting.

4.1.5 Regulator for subsystem DynamicCS

In the above example, the creation and deletion actions of the class can be called arbitrarily because we do not put any guards for these actions. The reason for leaving the guards empty is to make the actions general so that the reusability and extensibility of the design is kept.

From the requirements of the Dynamic Client-Server system, a dynamic number of clients will exist in the system and the client can connect or disconnect from the server depending on its status. Therefore, the live instance of the client has three states: con, dcon, and del. The first state means that the client has been connected to the server and put into the configuration diagram of

the system. The second state represents that the client has just been created or has disconnected from the server. If the client is deleted, it will be set to the third state. We will design a regulator for the subsystem DynamicCS to record the states of the clients and regulate the creation and deletion actions of the client class in the subsystem. The following diagram describes the relationship between the actions of subsystem DynamicCS and the regulator DCS-reg we will design.

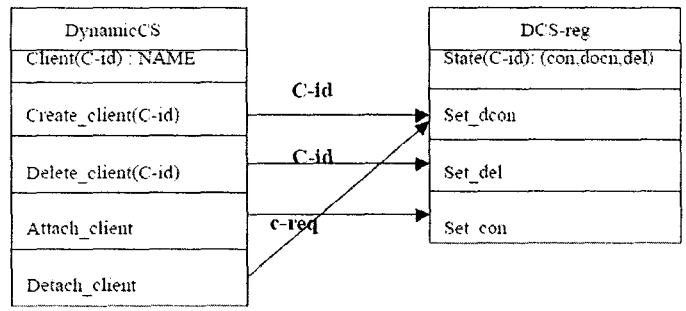


Figure 4. 3 The regulator for subsystem DynamicCS

The graphical notation for subsystem DynamicCS is as follows, and we add one channel out_id into it to record the current client id, which will be synchronized with the corresponding channel of the regulator to modify the client's state.

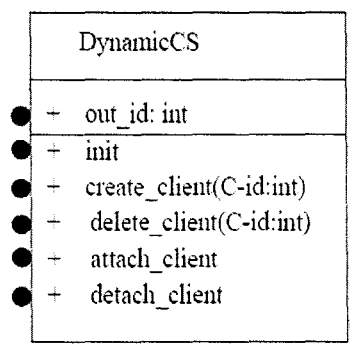


Figure 4. 4 Graphical notation of subsystem DynamicCS

At the same time, we need to modify the actions of DynamicCS to set the value of out_id to the current client's id in the action. So, in actions create_client

and delete_client, we will add out_id' = C-id, and assignments out_id' = c_con, out_id' = C.find should be added to actions attach_client, detach_client, respectively. Because all these assignments only write the new channel out_id, there exists a regulative superposition morphism from the "old" DynamicCS to this new subsystem. However, since the change is minor, we will still use the name DynamicCS to represent this new subsystem.

Then we define the regulator DCS-reg for DynamicCS as follows:

design component DCS-reg
in c_id: int
out state: array(enum(con,dcon,del))
actions
set_dcon1(C-id:int): state[c-id] <> dcon, false -> state[c_id]' = dcon
[] set_dcon2: true, false -> state[c_id]' = dcon
[] set_del(C-id:int): state[c-id] = dcon, false -> state[c_id]' = del
[] set_con: state[c_id] = dcon , false -> state[c_id]' = con
endofdesign

In Figure 4.3, we notice that actions create_client and detach_client will connect to the same action set_dcon of the regulator. Due to the problem of action synchronization of CommUnity, we replace action set_dcon by two actions to connect different actions of DynamicCS. We should also keep in mind that since create_client and delete_client are schema actions, their synchronized actions set_dcon1 and set_del in the regulator would be schema actions too, which means that a set of actions (with index C-id) for each of them should be defined in DCS-reg. The graphical representation of component DCS-reg is given below:

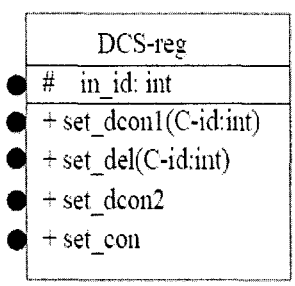


Figure 4. 5 Graphical notation of regulator DCS-reg

Then we can interconnect subsystem DynamicCS with the regulator DCS-reg with a cable and obtain the following specification of new system RDynamicCS. Notice that we need schema actions in the cable to interconnect systems having schema actions.

design component cable5
in v: int
actions
sync1(C-id:int): true -> skip
[] sync2(C-id:int): true -> skip
[] sync3: true -> skip
[] sync4: true -> skip
endofdesign

design subsystem RDynamicCS
associations
component cable5, DCS-reg
subsystem DynamicCS
morphisms
cable5 **to** DynamicCS
connections
v.cable5 **to** out_id. DynamicCS
sync1(C-id).cable5 **to** create_client(C-id).DynamicCS
sync2(C-id).cable5 **to** delete_client(C-id).DynamicCS
sync3.cable5 **to** attach_client. DynamicCS
sync4.cable5 **to** detach_client. DynamicCS
cable5 **to** DCS-reg
connections
v.cable5 **to** c_id. DCS-reg
sync1(C-id).cable5 **to** set_dcon1(C-id).DCS-reg
sync2(C-id).cable5 **to** set_del(C-id).DCS-reg
sync3.cable5 **to** set_con. DCS-reg
sync4.cable5 **to** set_dcon2. DCS-reg
participants
dyncs: DynamicCS; c:cable5; drg:DCS-reg

interface
init
actions
create_client(C-id) **to** create_client(C-id).dyncs
delete_client(C-id) **to** delete_client(C-id).dyncs
attach_client **to** attach_client.dyncs
detach_client **to** detach_client.dyncs
init DynamicCS(dyncs) ∧ Cable5(c) ∧ DCS-reg(drg) ∧ RDCS(dyncs,c,drg)
endofdesign

The graphical representation of subsystem RDynamicCS is as follows, which has the same interface as subsystem DynamicCS and provides the control for dynamic reconfiguration actions.

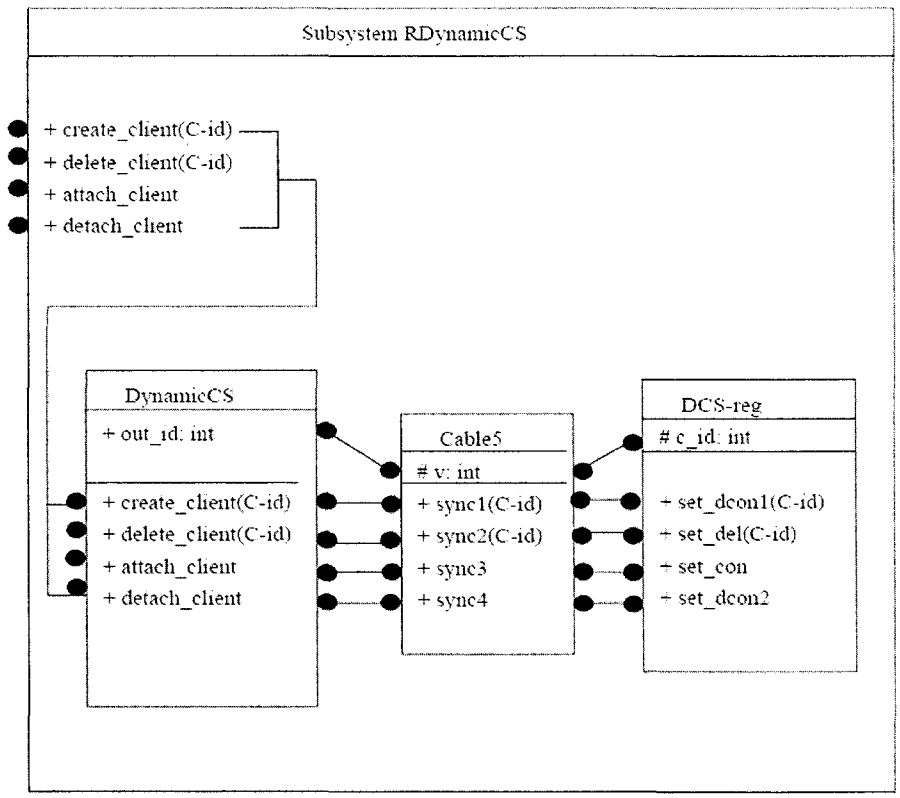


Figure 4. 6 Graphical notation of subsystem RDynamicCS

4.2 The transformation procedure

The purpose of this procedure is to take the system specified by DynaComm (with a fixed configuration diagram, i.e. a certain state of the system) as input and reduce it to the flat configuration diagram, which consists only of components, cables and regulative superposition morphisms. According to the proposition in section 2.4.4.3, the colimit of a flat configuration diagram will exist and the semantics of the configuration can be derived from the colimit directly based on the semantics of CommUnity we introduced in chapter 2. We call this procedure *TransComm* with the input *CompDg* (complicated diagram in DynaComm) and the output *FlatDg* (flat diagram in CommUnity).

Behind the DynaComm specification of any system there is a complicated configuration diagram built from components, connectors, subsystems and the corresponding morphisms, which we call *CompDg*. For components, they are the basic building blocks of *FlatDg* and nothing needs to be reduced. The main concerns are the connector, which serves as the crucial mechanism to interconnect the components, and the subsystem that contains subcomponents connected by morphisms between them. From the definition of subsystem in section 3.2.3, there will be two cases when we try to decompose *CompDg*.

- The association of the subsystem is specified by the components, subsystems and the morphisms among them.

We will look at the subcomponents of the system, which could be a component or a subsystem. Assume the configuration diagram of system S is as the following diagram, which consists of component instance c1 and subsystem instance s1 as subcomponents interconnected by cable c2.

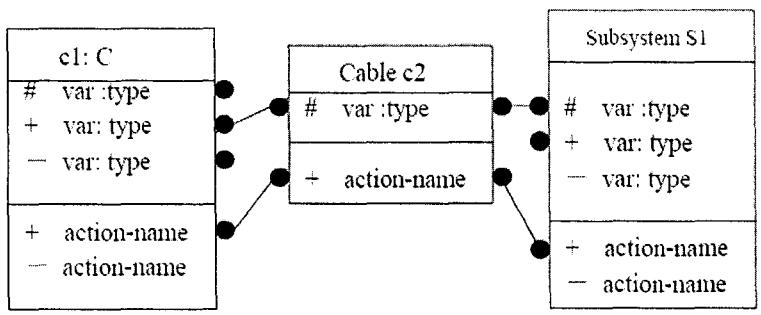


Figure 4. 7 Configuration diagram of system S

* If the subcomponent is a component, such as C1 in the diagram, then it need not be reduced.

* When the subcomponent is a subsystem, say S1 in system S, we will apply the *TransComm* procedure to the configuration diagram of S1 to get the flat diagram of subsystem S1. Then we can compute the colimit of the flat diagram, which is considered as a big component. The flat diagram of S will be obtained after we apply the *TransComm* procedure to all the subsystem subcomponents of S.

- The association of the subsystem is given by a connector C, which has glue g and roles R1, R2, … Rn.

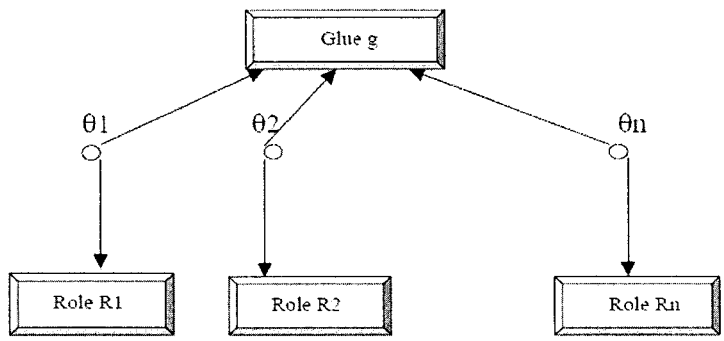


Figure 4. 8 The association of the system is a connector

In this case, we should consider the refinement morphisms between the roles and role instances in system S. According to our discussion in section 2.4, suppose role R1 is instantiated by an instance c1 with a refinement morphism, we can combine the regulative superposition from the cable to R1 with the refinement morphism to get a regulative superposition from the cable to c1, and the whole diagram can be updated similarly. After this step, we will be able to apply the *TransComm* procedure to the subcomponents of S recursively and get the flat diagram for system S, with the same method discussed in the first case.

4.3 Summary

To define the semantics of the DynaComm language, we have described a systematic approach to eliminate the parameters of actions, which are essential in specifying population management and reconfiguration actions in DynaComm. This approach has been applied to the dynamic client-server system we specified in chapter 3, to illustrate its effectiveness in transforming complex DynaComm specifications into CommUnity-like designs by introducing additional channels to record dynamic instances and morphisms between components and subsystems. Then we can obtain a high level semantic model by considering the dynamic subsystem's different configuration diagrams as states, and its reconfiguration actions as events to trigger the transition from one configuration diagram to another configuration diagram. Along with the method we have proposed to obtain a dynamic system's static semantics at a certain state, we believe that with further research in investigating their relationships, the semantics of the DynaComm language can be unified based on this solid foundation.

Chapter 5

Design with extension morphisms

It has been shown in [5] that higher-order connectors provide a very convenient basis for enhancing the behavior of an architecture of component designs, by the superimposition of aspects, such as fault tolerance, security, monitoring, compression, etc. Owing to the coordination mechanism of CommUnity, the coupling between the components has been reduced to a minimum so that we can superimpose aspects on existing systems through replacement, superposition and refinement of components. However, higher-order connectors are not powerful enough for defining various kinds of aspects, because some of them require extensions of the components and connectors [8]. Therefore, we will consider *extension morphism* (see definition 2.15 in chapter 2) as a candidate for defining extensions of components, which justifies the notion of substitutability arising in the context of object oriented design and programming, and provides a structuring principle for augmenting components by breaking encapsulation of the component in a controlled way [8].

This means that in a well-formed configuration diagram we should be able to replace component C by its valid extension, component C', and preserve the wellformedness of the diagram. We will prove this property in section 5.1. To illustrate the application of this principle in designing systems with the DynaComm language, a vending machine system example will be discussed in section 5.2 to show how we can combine regulative superpositions with extension morphims to derive an "augmented" version of the original system.

5.1 Combine regulative superpositions with extension morphisms

In this section we will consider the case that in a well-formed configuration diagram one component is extended by a design through extension morphism. Since we know that all the components are interconnected by cables through regulative superposition morphisms in a well-formed configuration diagram, the component to be replaced by the extended design is connected to a cable by the regulative superposition morphism, as shown in Figure 5.1. We will show that the regulative superposition can be combined with the extension morphism to obtain a new regulative superposition from the cable to the extended component.

Again, it is crucial to have the notion of cables to interconnect the components, to ensure that the composition of regulative superposition and extension morphism will give a new regulative superposition.

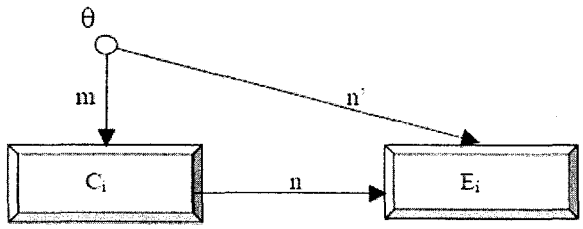


Figure 5. 1 Combine regulative superposition and extension morphism

Proposition 5.1 Suppose m is a regulative superposition morphism from cable θ to design C_i, and n is an extension morphism from design C_i to design E_i; there will exist a regulative superposition morphism n' from cable θ to design E_i.

Proof:
The morphism n' is defined as follows:
* n'_α is a total function: for every channel v in θ, $n'_\alpha(v) = n_\alpha(m_\alpha(v))$.
* n'_γ is a partial mapping: for every action g in E_i, if $n_\gamma(g)$ is defined and $m_\gamma(n_\gamma(g))$ is also defined, $n'_\gamma(g) = m_\gamma(n_\gamma(g))$; otherwise, it is undefined.

Since an extension morphism is also a signature morphism, we know n' is a signature morphism. To check if n' is a regulative superposition morphism, we need to check the following conditions:
* $I_{Ei} \Rightarrow n'(I_\theta)$.
Because n is an extension morphism, there exists a formula α, using only channels contained in $(V_{Ei} - n_\alpha(V_{Ci}))$, and α is satisfiable,
$\models I_{Ei} \Leftrightarrow n(I_{Ci}) \wedge \alpha$.
We have $I_{Ei} \Rightarrow n(I_{Ci})$, $I_{Ci} \Rightarrow m(I_\theta)$, so $n(I_{Ci}) \Rightarrow n(m(I_\theta)) \Leftrightarrow n'(I_\theta)$, and $I_{Ei} \Rightarrow n'(I_\theta)$.

* If $v \in loc(\theta)$, $g \in \Gamma_{Ei}$ and $n'_\alpha(v) \in D_{Ei}(g)$, then g is mapped to an action $n'_\gamma(g)$ and $v \in D_\theta(n'_\gamma(g))$.
* For every $g \in \Gamma_{Ei}$ where $n'_\gamma(g)$ is defined, if $v \in loc(\theta)$ and $g \in D_{Ei}(n'_\alpha(v))$,

then $R_{Ei}(g, n'_{\alpha}(v)) \Leftrightarrow n'_{\alpha}(R_{\theta}(n'_{\gamma}(g), v))$.
Because θ only contains input channels, $loc(\theta)$ is empty, so these two conditions hold.

* $L_{Ei}(g) \Rightarrow n'(L_{\theta}(n'_{\gamma}(g)))$.
* $U_{Ei}(g) \Rightarrow n'(U_{\theta}(n'_{\gamma}(g)))$.
From our definition of "middle" design, $L_{\theta}(n'_{\gamma}(g)) \Leftrightarrow$ true, $U_{\theta}(n'_{\gamma}(g)) \Leftrightarrow$ true, so these two conditions hold.

With this property, in a well-formed configuration diagram, we are able to replace each component by its extension component, by combining the regulative superposition from the cable to the old component with the extension morphism between the old component and its extension, to obtain a new regulative superposition from the cable to the extended component. Therefore, we reach the conclusion that in a well-formed configuration diagram of a system, we can extend any subcomponents of the system (through extension morphisms), and obtain an updated well-formed configuration diagram only containing regulative superpositions, through which the semantics of the new system can be derived from its colimit following the procedure introduced in section 2.4.4.3. Moreover, it can be shown that the colimit of the new configuration diagram is an extension of the colimit of the old configuration diagram [8].

From the proof of proposition 5.1, we can see that if θ_i is not a cable, the composition of a regulative superposition and an extension morphism may not give a regulative superposition. Therefore, it is necessary to enforce the designs to be interconnected by cables in a well-formed configuration diagram, so that the colimit will exist after extending any of the designs in the diagram through extension morphisms.

5.2 An example vending machine system

Now we want to model a system consisting of a customer and a vending machine with the DynaComm language. The requirement of this system is described as follows. The vending machine maintains a list of items, along with the price and amount of each item. The customer can place an order with the name of item and the payment to the vending machine. To simplify the example, currently we only allow the customer to order one item in a transaction, which will be extended later. The vending machine will check the price of the item and

decide if the order is accepted. If so, it will deliver the item along with the change to the customer; otherwise, the payment is returned to the customer. Currently, the vending machine will only accept payment comprised of nickels, dimes, quarters and 1 dollars, so it will refuse the order if the customer puts one cent in the payment. Meanwhile, if the vending machine is not able to make the change, it will also refuse the order and return the payment.

5.2.1 The design of the customer

Instead of modeling the customer with arbitrary behaviors, we choose to consider the machine's interface operated by the customer as the simulation of the customer's behavior. To make the system simple and general at first, the interface is divided into two parts: the buttons and the slot. The names of different items label the corresponding item buttons, and after the customer presses one of them, other item buttons will be disabled, so that he can only choose one item in an order. Then the customer can choose the "confirm" button to continue the order, where the slot will indicate to him to put the coins in and the complete order will be sent to the vending machine. If the customer chooses the "cancel" button, all the item buttons will be enabled and he can start another order.

The vending machine will check the price of this order and whether the ordered item is still available in its storage. If so, it will ask the slot to make the change. Then the vending machine will deliver the product to the slot and enable the item buttons, if the change can be made. Otherwise, the order will be refused and the payment is returned to the customer.

5.2.1.1 The interface controller

According to the above requirement, the customer places his order of an item through the buttons (including the item buttons and the command buttons: confirm and cancel) on the machine's interface, so we design an interface controller to model these buttons, as well as the customer's interaction with the interface of the machine. A finite set of actions for the item buttons and "confirm", "cancel" buttons are specified in the following design. The slot_get and slot_ret actions are designed to interact with the slot component to get the payment from the customer. Meanwhile, we use the order action to send the complete order to the vending machine, and after the order has been processed by the vending machine, the order_ret action will be called to reset the

controller.

design component controller
in
// the customer's payment in the slot
i_pay: int
prv
b_item: array (int);
bt_g: bool; //guard for item buttons
bt_confirm: bool; //guard for confirm/cancel buttons
slot_g:bool; // guard for slot get action
s_req:bool;
ord_g: bool; // guard for order action
o_req: bool
out
// order to vending machine
c_item: list (int);
c_pay: int
init ord_g = false ∧ o_req = false ∧ bt_g = true ∧ bt_confirm = false ∧ slot_g = false ∧ s_req = false ∧ c_item = NULL
actions
button_select (id: int) [bt_g,c_item,bt_confirm] : bt_g, false -> bt_g' = false ∧ c_item' = c_item * b_item [id] ∧ bt_confirm' = true
[] button_confirm [bt_confirm,slot_g] : bt_confirm, false -> bt_confirm' = false ∧ slot_g' = true
[] button_cancel [bt_confirm,bt_g,c_item] : bt_confirm, false -> bt_g' = true ∧ bt_confirm' = false ∧ c_item' = NULL
[] slot_get [slot_g,s_req]: slot_g, false -> slot_g' = false ∧ s_req' = true
[] slot_ret [c_pay, s_req, ord_g]: s_req, false -> c_pay' = i_pay ∧ s_req' = false ∧ ord_g' = true
[] order [o_req,ord_g]: ¬o_req ∧ ord_g, false -> o_req' = true ∧ ord_g' = false
// enable all the item buttons
[] order_ret [o_req, bt_g, c_item]: o_req, false -> o_req' = false ∧ bt_g' = true ∧ c_item' = NULL
endofdesign

The input channel i_pay indicates the payment received from the customer.

A finite set of item button actions (button_select) are defined, which correspond to the sequence of item buttons on the machine's interface. Again they are schema actions indexed by the id (in the above sequence) of the item buttons. We use a fixed size array b_item to store the item's index in the storage of the vending machine, and the index of array b_item will correspond to the id of the item button, e.g. the second item button b_item[2] may correspond to the item index 6 in the item list of the vending machine's storage.

The workflow of the controller component is described as follows. After one item button is selected, the guard bt_g is set to false to disable all the item buttons, so that the customer can only choose button confirm or cancel (as the enabling guards of other actions are disabled). If he chooses the confirm button, the guard slot_g is enabled and the slot_get action will be executed to request the customer's payment in the slot component. If the cancel option is selected, the controller will enable all the item buttons and wait for the customer's input from the beginning. After the payment is obtained from the slot, the order action will be called and it will send the order (c_item, c_pay) to the vending machine, then wait for the result of the order. After the vending machine processes the order and indicates the result to the order_ret action of the controller, the order_ret action will reset the item buttons and the c_item list, to be ready to accept another order. The graphical notation for the controller component is shown in Figure 5.2 (we ignore private channels and actions):

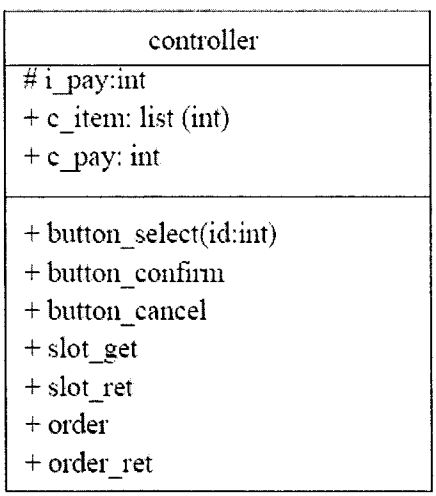


Figure 5. 2 Graphical representation of the controller component

Notice that we use a number of guards to control the sequence of actions in

the controller, and the correctness of our design can be ensured by controlling the right workflow of the component through the appropriate use of these guards. We also use the list data structure to record the ordered items, although currently only one item is allowed in the order. The reason is that in the different kinds of design morphisms we have discussed so far, the mapping of channels requires the types of channels to be preserved. If we use one channel of integer type to record the ordered item now and there is a new requirement to allow the customer to select multiple items in an order, we have to add new channels to the component and modify the corresponding actions as well, which seems awkward. Therefore, we choose the list data structure for the ordered items and the corresponding actions are designed to process the list of items.

We have also designed a pattern for a pair of actions of one component (e.g. slot_get and slot_ret), which sends a request to another component and waits for its response to proceed. The trick is to assign a guard (initialized to be false) to the callback action to make sure that it will not be called arbitrarily in an unexpected situation, and it will only be enabled in the request action.

5.2.1.2 The slot

The slot component takes care of the acceptance of the customer's payment and decides if the correct change can be made depending on its current storage of coins. When the interface controller requests the payment from the customer, the slot will distinguish the coins and it will refuse the payment and indicate this event to the controller if there exists one cent in the customer's input. Otherwise, it will store the coins and send the payment amount to the controller. Regarding the function for making the change, the slot is able to compute the composition of coins for the amount of change requested by the vending machine based on its current storage. If the computation is not successful, the vending machine will refuse the order and inform the slot to return the payment, which can certainly be made.

In the following design of component slot, a set of input channels such as i_dollar, i_quarter etc. represents the payment from the customer, a set of private channels is included as the coin storage of the slot, and we also use output channels o_nickel, o_dime, o_quarter and o_dollar to represent the change made by the slot. The get_pay action stores the coins in the payment and the send_pay action puts the amount of payment in the output channel o_pay. According to the amount of change that should be made in the input channel r_change, the comp_change action will compute the composition of coins, and the

send_change action will send the result of the computation (change_res) and update the storage of coins if needed. While the ordered item is accepted by the action rec_item, and the rec_return action receives the returned payment amount and returns the coins to the customer.

design component slot
in
// input coins from customer
i_cent: int;
i_nickle: int;
i_dime: int;
i_quarter: int;
i_dollar: int;
// received change amount and items from vending machine
r_change : int;
r_item: list (ITEM)
prv
// coins storage
s_nickle: int;
s_dime: int;
s_quarter: int;
s_dollar: int;
// guards for action sequence
get_g: bool;
change_g:bool;
item_g: bool
out
// changes made by the slot
o_nickle: int;
o_dime: int;
o_quarter: int;
o_dollar: int;
s_item: list (ITEM); // items to slot
o_pay: int; // payment amount to the controller
change_res: bool
init get_g = true ∧ change_g = true ∧ change_res = false ∧ item_g = false
actions
get_pay [get_g, s_nickle, s_dime, s_quarter, s_dollar]: get_g ∧ i_cent = 0,

false -> get_g' = false ∧ s_nickle' = s_nickle + i_nickle ∧ s_dime' = s_dime + i_dime ∧ s_quarter' = s_quarter + i_quarter ∧ s_dollar' = s_dollar + i_dollar
[] send_pay [get_g, o_pay]: ¬ get_g , false -> get_g' = true ∧ o_pay' = 100*i_dollar + 25*i_quarter + 10*i_dime + 5*i_nickle
[] comp_change [change_g, change_res]:change_g, false -> get_changed ∧ change_g' = false
[] send_change [change_g]: ¬change_g, false -> change_g' = true ∧ (change_res = true ⇒ item_g' = true ∧ s_nickle' = s_nickle - o_nickle ∧ s_dime' = s_dime - o_dime ∧ s_quarter' = s_quarter - o_quarter ∧ s_dollar' = s_dollar - o_dollar)
[] rec_item [s_item, item_g]: item_g, false -> s_item' = r_item ∧ item_g' = false
[] rec_return [ret_g, s_item,] : true, false -> s_item' = NULL ∧ get_changed
endofdesign

In the above design, we assume the function to compute the composition of change, namely get_change, has already been defined, which takes r_change as input and computes the number of nickels, dimes, quarters and dollars. If the computation is successful, it will set change_res to be true and the output channels for the change. Otherwise, change_res is set to false and this event is sent to the vending machine. Actually, get_change solves a linear programming problem, which takes s_nickel, s_dime, s_quarter, s_dollar and r_change as parameters. To simplify the specification of slot component, we do not describe the detailed procedure here.

The workflow of the slot component is described as below. When the interface controller requests the payment from the customer, the get_pay and send_pay actions will be executed to provide the payment amount to the controller. After the vending machine receives the order and recognizes that the payment is enough, it will ask the slot to compute the change. So, the comp_change action is called and the result of computation (change_res) is sent to the vending machine by the send_change action. If the result is successful, the change is given to the customer by the slot and the vending machine will send the product to the slot by means of the rec_item action. Otherwise, the rec_return action will get the amount of payment from the vending machine and give it back to the customer by calling the get_change function. The graphical notation for the slot component is as follows, where we ignore the private channels and actions.

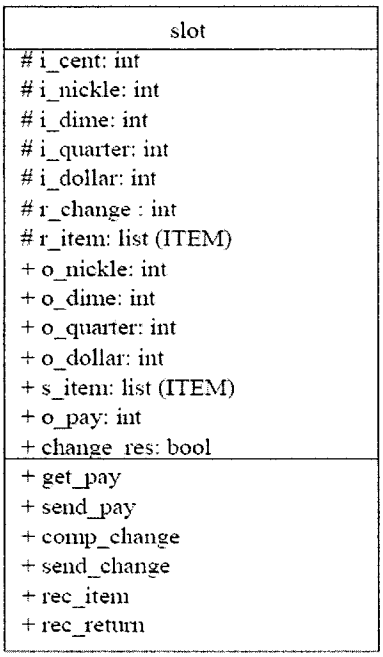


Figure 5. 3 Graphical representation of the slot component

5.2.2 The design of the vending machine

Based on the functional requirement of the vending machine, we will divide it into two components: vender and inventory, where the vender is in charge of the interaction with the customer interface (controller and slot), and the inventory serves as a database for storing the actual products (items) and maintaining the price and amount of each item.

5.2.2.1 The vender

The duty of the vender is to accept the order from the customer (the accept action), ask the inventory to the check the price and amount the the ordered item(s) (actions check_inv and check_ret), send the amount of change to the slot and ask if the change can be made (actions change and change_ret), request the item(s) from the inventory (actions req_item and req_return), deliver the item(s) to the customer (the delivery action) or return the payment (the return_ord action), and inform the interface controller to be reset to start a new order (the

reset_controller action). The design of the vender component is as follows, and the meaning of the channels is explained as the comments.

```
design component vender
in
  // the ordered item(s) and payment from the controller
  in_item: list (int);
  in_pay: int;
  // the price of the ordered item(s) from the inventory
  inv_price: int;
  inv_item: list(ITEM);
  // the result of checking whether the change can be made from the slot
  chg_res: bool
prv
  // the set of guards to control the sequence of actions
  ac: bool;
  ck: bool;
  cg: bool;
  rt: bool;
  rq:bool;
  rc: bool;
  dl:bool;
  // stores the requested item(s) from the inventory
  v_item: list(ITEM);
  // stores the order and payment from the customer
  ord_item: list(int);
  ord_pay: int
out
  // the order and payment to be sent to the inventory
  ck_item: list(int);
  ck_pay: int;
  // the amount of change to be sent to the slot
  chg_amt: int;
  // the ordered item(s) sent to the customer
  out_item: list(ITEM);
  // the returned amount of payment to be sent to the slot
  ret_amt: int
init
```

ac' = false ∧ ck' = false ∧ cg' = false ∧ rt' = false ∧ dl' = false ∧ rq' = false ∧ rc' =false

actions

accept [ac, ord_item, ord_pay, ck]: ¬ac, false -> ac' = true ∧ ord_item' = in_item ∧ ord_pay' = in_pay ∧ ck' = true

[] check_inv [ck, ck_item, ck_pay]: ck, false -> ck_item' = ord_item ∧ ck_pay' = ord_pay ∧ ck' = false

[] check_ret [cg, rt, v_item]: true, false -> (inv_price >= 0 ⇒ cg' = true) ∨ (inv_price = 0 ⇒ rt' = true)

[] change [cg, chg_amt]: cg, false -> chg_amt' = ord_pay — inv_price ∧ cg' = false

[] change_ret[rq, rt]: true, false -> (chg_res = true ⇒ rq' = true) ∨ (chg_res = false ⇒ rt' = true)

[] req_item [rq, ck_item]: rq, false -> ck_item' = ord_item ∧ rq' = false

[] req_return [v_item, dl]:true, false -> v_item' = inv_item ∧ dl' = true

[] return_ord [rt, ret_amt, out_item, ac, rc] : rt, false -> rt' = false ∧ ret_amt' = ord_pay ∧ out_item' = NULL ∧ rc' = true

[] delivery [dl, ac, out_item] : dl, false -> dl' = false ∧ out_item' = v_item ∧ rc' = true

// inform the controller to accept another order

[] reset_controller [rc]: rc, false -> rc' = false ∧ ac' = false

endofdesign

According to the initialization condition of this design, only the accept action is enabled and it is synchronized with the order action of controller to accept the order of the customer. It also sets the guard ck to be true, so that the check_inv action will be executed to ask the inventory to check the price and amount of the ordered item(s). The check_ret action waits for the response from the inventory: if inv_price>=0, it means that the transaction can continue and this action sets the guard cg to be true, to call the slot to check if the change can be made; otherwise, it enables the guard rt to call the return_ord action, if any item is not available or the payment is not enough.

If the order can continue, the change action is synchronized with the comp_change action of the slot to make the appropriate change to the customer. Then the change_ret action will wait for the response from the slot indicated by the input channel chg_res: if the change can be made, the vender will request the item(s) from the inventory by the req_item action, which is synchronized with the rec_req action of the inventory; otherwise, the return_ord action is called to

return the payment. After the vender receives the requested item(s) from the inventory by the req_return action, the delivery action will be called, which is synchronized with the rec_item action of the slot to deliver the item(s). Otherwise, the action return_ord will be executed and the slot's action rec_return will be synchronized to return the payment to the customer. Finally, the vender will call the reset_controller action to synchronize with the order_ret action of the controller to inform it that the next order can be taken now.

The graphical notation for the vender component is as follows (we ignore private channels and actions):

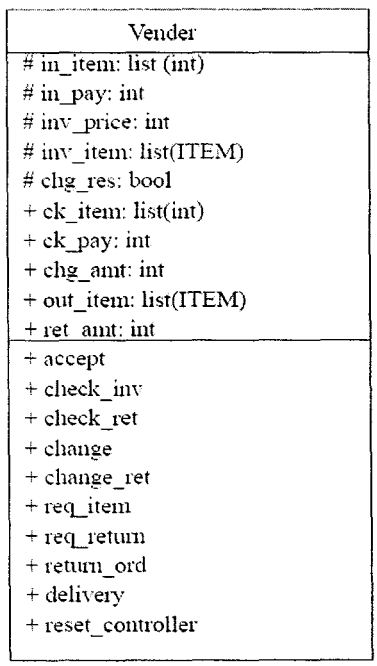


Figure 5. 4 Graphical representation of the vender component

Again, we use a set of guards to control the sequence of actions in the vender component, and in the above explanation of the component's work mechanism, we are able to control the right workflow of the design through the appropriate use of these guards, so that the correctness of our design can be ensured.

5.2.2.2 The inventory

The design of component inventory can be derived from its functional requirement described at the beginning of section 5.2.2. The inventory component maintains a list of items along with their price and remaining amount: (item_id:int, item:ITEM, price:int, amount:int), where item_id is the item's index in the storage and item represents the real item product. We use an array db (with a fixed size) to store this list of items, and the index of this array corresponds to item_id. Meanwhile, we assume functions first, second and third have been defined to return the first, second and third member of db, respectively.

The private action count_item calculates the amount of each ordered item and stores it in the channel s_item. It also computes the total price of the order. The check_price action goes through the inventory database and compares the amount of each ordered item with the amount of that item in the storage. If the storage is not enough or the payment is less than the price of the order, the output channel will be set to 0; otherwise, it will set to the value in p_price. The get_item action will retrieve the items from the storage according to the order and update the db channel. The specification of the inventory component is as follows:

```
design component inventory
in
  // the ordered item(s) and payment from the vender
  i_item: list (int);
  i_pay: int
prv
  // stores the ordered item(s)
  p_item: list (int);
  r_item: list (int);
  p_price: int;
  // array index is item id
  db: array (ITEM, int, int);
  // stores the amount of each ordered item, all the entries are initialized to be 0.
  s_item: array (int);
  j :int;
  // the guards to control the sequence of actions
  price_g: bool;
```

amt_g: bool;
ret_g: bool;
send_g : bool
out
o_item: list (ITEM);
// the price of the order sent to the vender
o_price: int
init p_item = NULL ∧ price_g = false ∧ amt_g = false ∧ ret_g = false ∧ o_price = 0 ∧ o_item = NULL ∧ r_item = NULL ∧ send_g = false
actions
check []: true, false -> p_item' = i_item
[] **prv** count_item []: p_item != NULL, false -> s_item [head(p_item)]' = s_item [head(p_item)] + 1 ∧ p_price' = p_price + second (db [head(p_item)]) ∧ p_item' = tail(p_item) ∧ (tail(p_item) = NULL ⇒ price_g' = true)
[] **prv** check_price [] : price_g, false -> price_g' = false ∧ ((i_pay >= p_price ⇒ amt_g' = true ∧ j' = 1) ∨ (i_pay < p_price ⇒ ret_g' = true ∧ o_price' = 0))
[] **prv** check_amt[] : amt_g ∧ (j <= sizeof(db)) , false -> ((s_item[j] <= third(db[j]) ⇒ j' = j + 1 ∧ (j =sizeof(db) ⇒ ret_g' = true ∧ o_price' = p_price)) ∨ (s_item[j] > third(db[j]) ⇒ amt_g' = false ∧ o_price' = 0 ∧ ret_g' = true))
[] inv_ret [] : ret_g, false -> ret_g' = false
[] rec_req []: true, false -> r_item' = i_item
[] **prv** get_item [] : r_item != NULL, false -> o_item ' = o_item * first(db[head(r_item)]) ∧ third(db[head(r_item)])' = third(db[head(r_item)])-1 ∧ r_item' = tail(r_item) ∧ (tail(r_item) = NULL ⇒ send_g' = true)
[] send_item [] : send_g, false -> send_g' = false
endofdesign

The workflow of this component is as follows. First, the check action is called to enable the guard of the count_item action. Then the action check_price is called to decide if the total price is less than ck_pay. If so, the inv_ret action will be enabled to return the result (inv_price) to the vender. Otherwise, the check_amt action is executed to check if the amount of each ordered item in the inventory is greater than the number of this item requested in the order. If so, it will call action inv_ret to return inv_price > 0 (the total price of the items in ck_item); otherwise, it will return inv_price = 0 in the inv_ret action. After the vender verifies that the change can be made, it will call the req_item action, which is synchronized with the rec_req action of the inventory, to get the ordered items and update the storage, and the inventory has the send_item action

to send the ordered items back to the vender.

Notice that in the count_item action we use the guard p_item != NULL to iterate through the list of ordered items. It can be generalized as a mechanism to implement the loop structures in the DynaComm language. (See future work.)

5.2.2.3 The vending machine subsystem

According to our design of the vender and inventory components and the discussion of their interactions, we can put them together by interconnecting the vender and the inventory through a cable. The configuration diagram of the vending machine subsystem is as follows:

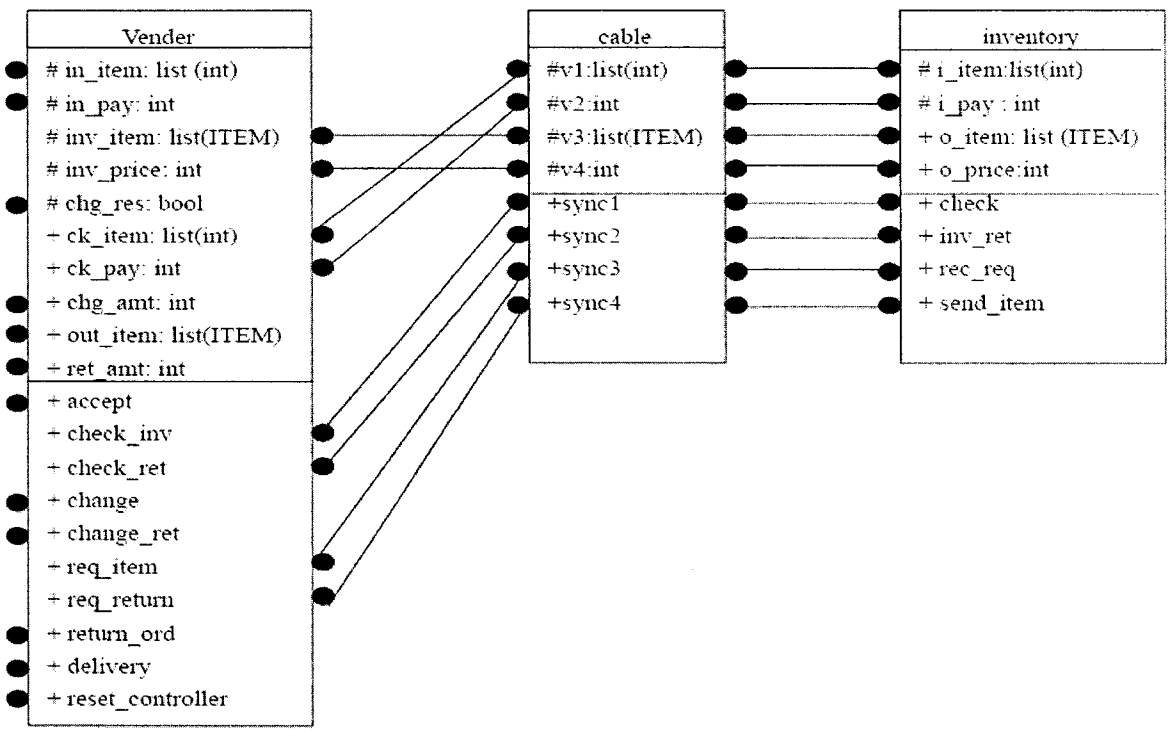


Figure 5. 5 Configuration diagram of the vending machine subsystem

The specification of the subsystem vending machine can be obtained easily from the above configuration diagram and we do not describe it in detail here. We can also determine the interface of this subsystem by looking at the left part

interface of the vender component in the diagram, which will interact with the interface controller and the slot.

Now we can put the vending machine subsystem together with the interface part (the controller and slot) to obtain the required vending machine system, which satisfies the design requirements, and the morphims between them are described in the following configuration diagram.

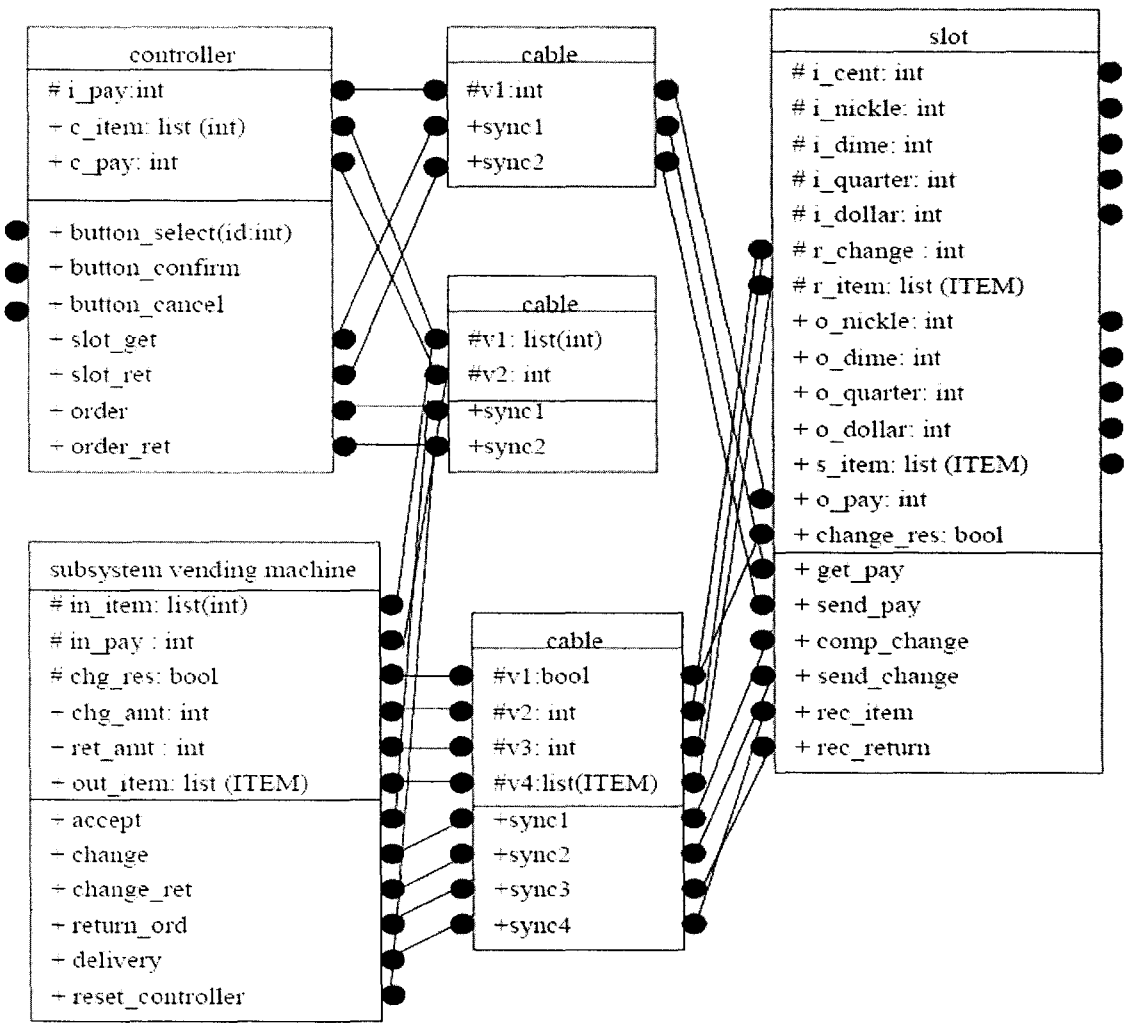


Figure 5. 6 Configuration diagram of the vending machine system

The interface of the system is shown in the left interface section of the controller component and the right interface section of the slot component, in

which the controller provides the buttons for the customer to select his favorite item and confirm or cancel the order, and the slot indicates to the customer to put the coins and get his ordered item and change.

5.2.3 The extended vending machine system

Now we want to add more behaviors to the vending machine system to improve the quality of its service. There are two extensions to be made, and we will show that they can only be achieved by the usage of extension morphisms.

- The extension allowing multiple items in an order

One extension we want to make is to allow the customer to select more than one item in an order, which should be done in the controller component. We must modify the actions of item buttons to achieve this effect. First, we will extend the controller component and show there is an extension morphism from the old controller to this extended new component. Then a proof is given to justify that it is impossible to regulate or refine the controller to obtain the required functionality and the extension morphism is necessary for our purpose.

We introduce a new channel ac: bool (initialized to be true) and weaken the guards of item buttons actions by taking the disjunction of ac with bt_g. The modified actions of the controller are as follows:

.
.
.

init ord_g = false ∧ o_req = false ∧ bt_g = true ∧ bt_confirm = false ∧ slot_g = false ∧ s_req = false ∧ c_item = NULL ∧ ac = true

actions

button_select (id: int) [bt_g,c_item,bt_confirm] : bt_g ∨ ac, false -> bt_g' = false ∧ c_item' = c_item * b_item [id] ∧ bt_confirm' = true

[] button_confirm [bt_confirm,slot_g,ac] : bt_confirm, false -> bt_confirm' = false ∧ slot_g' = true ∧ ac' = false

[] button_cancel [bt_confirm,ac,bt_g,c_item] : bt_confirm, false -> bt_g' = true ∧ ac' = true ∧ bt_confirm' = false ∧ c_item' = NULL

[] order_ret [o_req,bt_g,c_item,ac]: o_req, false -> o_req' = false ∧ bt_g' = true ∧ c_item' = NULL ∧ ac' = true

.
.

We call the extended version of the controller component controller'. It is easy to determine that controller' satisfies the new requirement. After the customer selects an item button, the enabling guards of button_select actions will remain true because ac is true. They will not be disabled until the customer selects the confirm button, and after the vending machine subsystem informs the controller that the order has been processed by calling the order_ret action, all the item buttons will be reset.

We need to show that there exists an extension morphism from the old controller component (say P_1) to controller' (say P_2). The morphism σ is defined as follows: the mapping of the channels σ_α will map each channels of P_1 to the identical channel of P_2, and σ_γ defines the mapping of actions from each action in P_2, to the identical action in P_1. We assert that σ is an extension morphism from P_1 to P_2.

Proof:

First we will show that σ is a signature morphism. Since the mappings of channels and actions are the identity, it is easy to see that all the conditions of a signature morphism are satisfied, except the condition $\sigma_\alpha(D_1(\sigma_\gamma(g)) \subseteq D_2(g)$. Since the actions in P_2 keep the effect of assignment to the mapped channels of P_1, this condition also holds. Therefore, σ is a signature morphism.

Then we will check the conditions of extension morphism according to definition 2.15:

* Obviously σ_γ is surjective and σ_α is injective.
* There exists a formula α, which contains only channels from $(V_2 - \sigma_\alpha(V_1))$, and α is satisfiable, $\models I_2 \Leftrightarrow \sigma(I_1) \wedge \alpha$.

$\alpha \Leftrightarrow$ ac = true, this condition holds.

For every $g \in \Gamma_2$ where $\sigma_\gamma(g)$ is defined,

* If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then there exists a formula α, which contains only primed channels from $(V_2' - \sigma_\alpha(V_1)')$, and α is satisfiable, $\models \sigma(L_1(\sigma_\gamma(g))) \Rightarrow (R_2(g, \sigma_\alpha(v)) \Leftrightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)) \wedge \alpha)$.

For action button_confirm, $\alpha \Leftrightarrow$ ac' = false, this condition holds.

For action button_cancel, $\alpha \Leftrightarrow$ ac' = true, this condition holds.

For action order_ret, $\alpha \Leftrightarrow$ ac' = true, this condition holds.

* If $v \in loc(V_1)$, $g \in D_2(\sigma_\alpha(v))$, then $v \in D_1(\sigma_\gamma(g))$.

Since the mappings of channels and actions are the identity and the actions in P_2 keep the effect of assignment to the mapped channels of P_1, this condition will hold.

* $\models (\sigma(L_1(\sigma_\gamma(g))) \Rightarrow L_2(g))$.

For each action button_select (id: int), bt_g $\Rightarrow$ bt_g $\vee$ ac.

* $\models (\sigma(U_1(\sigma_\gamma(g))) \Rightarrow U_2(g))$.

The progress guards of each mapped action are the same.

Now we will prove that the new functional requirement cannot be achieved by regulating or refining the controller component.

Proof:

The enabling guards of these button_select actions cannot be strengthened because in that case, all the buttons will be disabled after the customer selects one item button. The justification for this statement is as follows:

Suppose we have regulated or refined the controller component, then in the target component the enabling guards of the button_select actions will be strengthened, say one of the actions is g, its enabling guard is f and f $\Rightarrow$ σ(bt_g) (bt_g must be translated). According to the definition of regulative superposition and refinement morphism, we have $R_2(g, \sigma(bt\_g)) \Rightarrow \sigma(R_1(\sigma_\gamma(g), bt\_g))$. Since bt_g is set to false after the button_select action is called in the old controller, we know that σ(bt_g) should also be set to false after the execution of g in the extended controller. Because we have f $\Rightarrow$ σ(bt_g) and it should hold all the time, if σ(bt_g)' is false, we know f' must be false. Therefore, after the button_select action is executed in the target component, this action will be blocked, which means this item button is disabled.

- The extension of payment options

We expect that instead of only accepting payment consisting of nickels, dimes, quarters and one dollars, the vending machine system can also accept the payment including one cents and make the correct change. It is clear that we cannot refine or regulate component slot to achieve this goal, because we must modify its action get_pay and relax its enabling guard, which is not allowed in regulative superpositions and refinement morphisms. Therefore, we have to apply an extension morphism to the slot by modifying the get_pay action as follows and obtain the extended slot component.

.

.

.

get_pay [get_g, s_nickle, s_dime, s_quarter, s_dollar, s_cent]: get_g, false -> get_g' = false $\wedge$ s_nickle' = s_nickle + i_nickle $\wedge$ s_dime' = s_dime + i_dime $\wedge$ s_quarter' = s_quarter + i_quarter $\wedge$ s_dollar' = s_dollar + i_dollar $\wedge$ s_cent' =

s_cent + i_cent

.

.

.

Notice that we need to add a new channel s_cent into the slot component to store the cents and make the corresponding assignment to this channel. However, based on the definition of extension morphism, there will exist an extension morphism from the old component to this extended component (the proof is similar to the first extension case). For the same reason, we can modify the following actions of the slot component as well (and add the channel o_cent):

.

.

.

send_pay [get_g, o_pay]: ¬ get_g , false -> get_g' = true ∧ o_pay' = 100*i_dollar + 25*i_quarter + 10*i_dime + 5*i_nickel + i_cent

send_change [change_g]: ¬change_g, false -> change_g' = true ∧ (change_res = true ⇒ item_g' = true ∧ s_nickle' = s_nickle − o_nickle ∧ s_dime' = s_dime − o_dime ∧ s_quarter' = s_quarter − o_quarter ∧ s_dollar' = s_dollar − o_dollar ∧ s_cent' = s_cent – o_cent)

.

.

.

Since we have divided the functionality of the system in an appropriate way, we can simply reuse the vending machine subsystem and the controller component.

5.3 Summary

In this chapter we have presented the approach of combining regulative superpositions and extension morphisms to add new behavior into the existing systems specified by DynaComm, which cannot be accomplished by refinement morphisms and regulative superpositions. The theoretical foundation of this approach has been established first by proving that regulative superpositions and extension morphisms can be merged into "new" regulative superpositions in the context of a well-formed configuration diagram, to ensure the effectiveness of colimit constructions to the diagram, and the colimit itself will be an extended version of the original system [8].

A vending machine system has been specified to illustrate the applicability of the above approach, by showing that extension morphisms are necessary for incorporating new features into the system, and we need to augment the existing system through the regulation and extension of its subcomponents to make them work together. The support for structural and incremental design principles of our approach has been demonstrated by this example.

Chapter 6

Conclusions and Future Work

This chapter contains a brief review of the DynaComm language proposed in this thesis and a summary of our contributions in the extension of CommUnity to specify dynamic software architectures. We discuss possible directions for future research on the semantics of DynaComm and relating DynaComm to temporal logic specifications to support reasoning about system properties. Some further exploration on the relationship between different design morphisms is also discussed.

6.1 Review of DynaComm and Contributions

We have presented the syntax and (partial) semantics of a new Architecture Description Language for the specification of component-based systems, providing special support for dynamic reconfiguration. To clarify the motivation of this language, several representative ADLs with support for dynamic reconfiguration are surveyed, with a focus on their language constructs, associated styles of specification and mechanisms to achieve dynamic reconfiguration. Then we provide a detailed review of the CommUnity ADL, focusing on its semantic model, well-formed mathematical foundations rooted in Category Theory and the clarification of different categories of morphisms between designs. Because we can treat designs as objects in a category and specify regulative superpositions as morphisms between designs, the category REG is defined such that we are able to apply pushout and colimit constructions to build large systems from interconnected components.

Given the nice properties of CommUnity designs, the language's effective support for compositionality, modularity, reusability and enforcement of design principles is explained to give the rationale for our work on DynaComm, the "refinement" and extension of CommUnity to incorporate dynamic reconfiguration mechanisms, containing good features of the ADLs we have surveyed. Specifically, DynaComm allows for the specification of reconfigurable component-based systems by defining:

- Components: the smallest computational and data storage units not composed of simpler components. They are equivalent to the concept of designs in CommUnity, which are also called basic components, through which we are able to build complex systems out of simpler ones.

- Connectors: the encapsulation of component interaction patterns that can organize the complicated interactions between the components of a system. We provide a systematic syntax of connector, which offers a clear definition of connections between the glue and roles, supports the refinement of a connector by refining its glue or roles, incorporates attributes and constraints on the connector level to enable the specification of architecture patterns, and for the purpose of specifying flexible reconfigurations, introduces actions into connector definitions to support dynamic connectors that can reconfigure themselves.

- Subsystems: the coarse grained components which contain basic components, subsystems and their interactions (usually represented by connectors) as well. Subsystems encapsulate data in the form of their attributes, internal component or subsystem instances and the attributes of interacting subcomponent instances. Subsystems also encapsulate behavior by the joint actions of interconnected subcomponents, population management actions and reconfiguration actions, which create or delete instances of subcomponents, and modify the way in which these subcomponents interact (dynamic connectors).

- Population manager: the attributes and actions defined on the subsystem level to manage the creation and deletion of subcomponent instances inside the subsystem. We provide the rationale to justify our design choice of population manager for managing the name space of the whole subsystem, instead of specifying class managers for individual component or subsystem classes.

DynaComm also has the notion of interface manager, which overcomes the problem of incorrectly synchronized actions, originated from the mechanism of action synchronization in CommUnity, where the synchronized actions must occur simultaneously. This scenario is very common in the context of dynamic reconfigurable systems, where multiple instances of a component will be connected to the other component. To follow the composition principle of CommUnity, we have designed the interface manager as a regulator to be

applied to the target component and derive the "augmented" component automatically.

With respect to the semantics of the DynaComm language, we have described a systematic approach to eliminate the indices of actions, since indices are necessary for defining reconfiguration actions in a dynamic architecture. We have also shown the effectiveness of this approach in transforming subsystem specifications of DynaComm into CommUnity-like designs, where a high level semantic model can be obtained by considering different configuration diagrams of the subsystem as states, and the reconfiguration actions as events.

We have been greatly motivated by the structural and incremental design principles based on the investigation of appropriate ways of combining regulative superpositions and refinement morphisms in designing a complex system. The soundness of the combination of these design morphisms has been proved. Through the development of a fault tolerant dynamic client-server system, we determined that connectors can be regulated by superposing regulators onto their roles while preserving the interfaces, to add new functionalities into existing systems, which supports hierarchical specification and incremental design principles.

Finally, we investigated a new notion of extension morphism, which enables us to establish extension relationships between the components, in the sense of inheritance in object orientation. Compared with regulative superposition and refinement morphisms, it is of a different nature and characterizes a structured use of invasive superpositions to allow the breaking of encapsulation while preserving certain properties. We have presented the approach of combining regulative superpositions and extension morphisms to add new behavior into the existing systems, which cannot be accomplished by refinement morphisms and regulative superpositions. The theoretical foundation of this approach has been established, by proving that regulative superpositions and extension morphisms can be merged into "new" regulative superpositions in the context of a well-formed configuration diagram, to ensure the effectiveness of colimit constructions for the diagram, and the colimit produced will be an extended version of the original system.

6.2 Future Work

We have shown the "static" semantics of the DynaComm language by transforming a certain state of the dynamic system's configuration diagram into a flat diagram and deriving its semantics by the use of systematic pushout and

colimit construction in CommUnity. With the high-level semantic model of subsystem specifications derived from the normalization procedure, some further investigation of their relationships needs to be conducted to find an appropriate way to combine them and obtain the unified semantics of the DynaComm language.

We notice that the property proved in section 5.1 might not hold when combining extension and refinement morphisms. Suppose component C' is a refinement of component C and there is an extension morphism from C' to another component C''. Because C' will strengthen the enabling guard of C while C'' can weaken the enabling guard of C', it might happen that the enabling guard of C'' is weaker than that of C, therefore, C'' is not a refinement of C. However, we expect the refinement of C can be preserved by C''. Therefore, in future research, we will try to find the conditions under which we are able to combine a refinement morphism with an extension morphism (or vice versa) to obtain a new refinement morphism (or extension morphism). Furthermore, we want to answer the question about how to make refinement, extension and regulative superposition morphisms work together in specifying a complex system, and study the corresponding design principles implied by their relationships.

With respect to the ADLs we reviewed in chapter 2, namely Dynamic Wright, Darwin and Dynamic Acme, which are able to modify the system's structure during run time, the internal mechanisms for reasoning about possible system evolution have not been incorporated into these languages, although they provide the definition of components, connectors and transformation operations to change architectures dynamically. Actually they each require some meta-language to perform reasoning about a system's properties and behaviors in some informal way. A temporal logic based formalism has been proposed in [1][2] for the specification of reconfigurable systems, where temporal logic is used as a formal basis for specifying software architectures and provides direct support for reasoning. Based on the understanding of the relationship between CommUnity designs and temporal logic specifications in [17], we will investigate the appropriate ways of transforming the specifications in DynaComm (with the new language constructs we introduced) into the corresponding temporal logic specifications, so that reasoning about system properties and evolution will be supported within the DynaComm language.

Bibliography

[1] Aguirre, N., and Maibaum, T., *A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems*, ASE 2002, pp. 271-274.

[2] Aguirre, N., and Maibaum, T., *A Logical Basis for the Specification of Reconfigurable Component-Based Systems*, FASE 2003, pp. 37-51.

[3] Aguirre, N., and Maibaum, T., *Some Institutional Requirements for Temporal Reasoning on Dynamic Reconfiguration of Component Based Systems*, Verification: Theory and Practice 2003, pp. 407-435.

[4] Aguirre, N., *A Logical Basis For the Specification of Reconfigurable Component Based Systems*, Ph.D. Thesis, King's College London, Department of Computer Science, 2004.

[5] Aguirre, N., Alencar, P., and Maibaum, T., *Designing with Aspects, or Why you always knew that aspect weaving was colimit construction*, submitted 2005.

[6] Aguirre, N., Maibaum, T., Alencar, P., and Regis, G., *Reasoning about Temporal Properties of CommUnity Designs*, submitted 2005.

[7] Aguirre, N., Maibaum, T., and Alencar, P., *Abstract Design with Aspects*, submitted, 2005.

[8] Aguirre, N., Maibaum, T., and Alencar, P., *Extension Morphisms for CommUnity*, Essays Dedicated to Joseph A. Goguen, Lecture Notes in Computer Science, vol. 4060/2006, pp. 173-193, 2006.

[9] Allen, R., and Garlan, D., *Formalizing Architectural Connections*, in Proceedings ICSE'94, Sorrento, Italy, 1994.

[10] Allen, R.J., *A Formal Approach to Software Architecture*, Ph.D. Thesis, Carnegie Mellon University, School of Computer Science, available as TR# CMU-CS-97-144, May 1997.

[11] Allen, R., Douence, R., and Garlan, D., *Specifying and Analyzing Dynamic Software Architectures*, in: Astesiano E, ed. Proc. of the Fundamental Approaches to Software Engineering. LNCS 1382, Berlin: Springer-Verlag, pp. 21-37, 1998.

[12] Bicarregui, J.C., Lano, K.C., and Maibaum, T., *Towards a Compositional Interpretation of Object Diagrams*, Algorithmic Languages and Calculi, Chapman & Hall, pp. 187-207, 1997.

[13] Bradbury, J.S., Cordy, J.R., Dingel, J., and Wermelinger, M., *A survey of self-management in dynamic software architecture specifications*, Proc. of the 2nd Workshop on Self-Healing Systems, ACM Digital Library, 2004.

[14] Corradini, A., and Hirsch, D., *An Operational Semantics of COMMUITY Based on Graph Transformation Systems*, Electr. Notes Theor. Comput. Sci. 109: pp. 111-124, 2004.

[15] Fiadeiro, J.L., and Maibaum, T., *Categorical Semantics of Parallel Program Design*, Technical Report, FCUL and Imperial College, 1995.

[16] Fiadeiro, J.L., and Maibaum, T., *Design Structures for Object Based System*, in Formal Methods and Object Technology, Springer-Verlag, pp. 183-204, 1996.

[17] Fiadeiro, J.L., and Maibaum, T., *Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality*, SIGSOFT FSE, pp. 72-80, 1995.

[18] Fiadeiro, J.L., *Categories for Software Engineering*, Spinger, 2005.

[19] Garlan, D., Monroe, R., and Wile, D., *ACME: An Architecture Description Interchange Language*, in Proceedings of CASCON'97, Toronto, Ontario, 1997.

[20] Garlan, D., *Software Architecture: A Roadmap*, in The Future of Software Engineering, Filkenstein A. (ed), ACM Press, 2000.

[21] Georgiadis, I., *Self-Organising Distributed Component Software Architectures*, Ph.D. Thesis, Imperial College of Science, Technology and Medicine, Department of Computing, 2002.

[22] Goguen, J., *Mathematical Representation of Hierarchically Organised Systems*, in Attimger, E. (ed), Global Systems Dynamics, Krager, pp. 112-128, 1971.

[23] Goguen, J., and Ginali, S., *A Categorical Approach to General Systems Theory*, in Klir, G. (ed), Applied General Systems Research, Plenum, pp. 257-270, 1978.

[24] Goguen, J., *Categorical Foundations for General Systems Theory*, in Pichler, F., and Trappl, R. (eds), Advances in Cybernetics and Systems Research, Transcripta Books, pp. 121-130, 1973.

[25] Larsen, K.G., and Skou, A., *Bisimulation through probabilistic testing*, Information and Computation, 94:1-28, 1991.

[26] Magee, J., and Kramer, J., *Dynamic Structure in Software Architectures*, in Proceedings of the ACM SIGSOFT '96: Fourth Symposium on the Foundations of Software Engineering, San Francisco, CA, pp. 24-32, Oct., 1996.

[27] Manna, Z., and Pnueli A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

[28] Medvidovic, N., and Taylor, R.N., *A classification and comparison framework for software architecture description languages*, IEEE Trans. on Software Engineering, 26(1): pp. 70-93, 2000.

[29] Perry, D.E., and Wolf, A.L., *Foundations for the study of software architectures*, SIGSOFT Software Eng. Notes, vol.17, no. 4, pp. 40-52, Oct.1992.

[30] Shaw, M., and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Apr. 1996.

[31] Szyperski, C., and Pfister, C., *Component–Oriented Programming: WCOP '96 Workshop Report*, Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP '96, Linz, pp. 127-130.

[32] Wermelinger, M., and Oliveira, C., *The CommUnity Workbench*, in Proc. of the 24th Intl. Conf. on Software Engineering, page 713. ACM Press, 2002.

[33] Wile, D.S., *Using Dynamic Acme*, in Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia. Dec. 2001.