Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - Nantes
1999

Generic Component Architecture Using Meta-Level Protocol Descriptions

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)

By: Maria Jose Presso

Promotor: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotors: Prof. Kris DeVolder (Vrije Universiteit Brussel)
Prof. Pierre Cointe (Ecole des Mines de Nantes)

Acknowledgments

I want to express my gratitude to all the people who helped me to carry out this thesis.

I thank my advisors, Kris De Volder and Pierre Cointe. To Pierre Cointe for initiating me in the field of reflection and broadening my horizon by pointing to many interesting topics. To Kris De Volder for giving me the right mixture of advice, pushing and support I needed to do this work.

I thank my promotor Theo D'Hondt for encouraging me when I needed it most.

Thanks all the people who made the EMOOSE program possible, and very specially to Annya Romanczuk, whose concern went really far beyond duty.

Thanks to all the people at EMN and PROG who was ready to help in many different ways, in academic aspects as well as in solving the numerous quotidian problems that come up when being far from home.

This wouldn't have been possible without the unconditional support from my family. Their love and concern were and will always be essential for any undertaking.

I want to thank the enfermeros for their for their day-to-day company and support, and for the fun we had together which was often just the needed help. Their friendship made this rather than possible, enjoyable.

To all the people who shared gratifiyng moments in Brussel and Nantes. Meeting them was an important part of this being a rich experience.

Abstract

The purpose of component technology is to build applications by composing reusable of software pieces without programming.

This dissertation addressed the problem of describing protocols for composing components in such a way that they are explicit and become independent from the composition tool. This allows to build generic tools capable of handling different kinds of composition protocols.

In order to be able to build applications out of components, it is necessary to have descriptions of components that tell how to deploy them, and how to connect them to others. Component models standardize these composition mechanisms, giving raise to composition protocols. Those protocols are described in specifications and embedded into builder tools that allow to describe an application built up from connected components and generate the code.

We propose the use logic meta programming to describe the components, describe the protocols for composition and the code that realizes them, specify an application built up from connected components and generate the code for the application. We show by constructing a prototype tool, that these descriptions make the protocols and the tool independent.

Contents

List of Figures

Chapter 1

Introduction

In the latest years there is a growing interest in technologies to build applications by assembling reusable software pieces or *software components*. The availability of this kind of technology lowers development effort and allows to build customized software in a cost effective way. It also improves reliability, as components become thoroughly tested by successive uses.

Components are encapsulated entities that are meant to be assembled to make up applications. A component is not used in isolation, it needs to interact with others to form a system. To be able to interact with others, components have well defined interfaces that specify what they provide and what they require from other components. As a reusable entity, a component can be used or deployed in different applications and it can be configured or adapted to fit the needs of a particular application.

Using components, applications are assembled rather than programmed. The development of an application based on components consists in deploying a set of components, configuring them and connecting them. Deployment, configuration and connection are thus the basic mechanisms necessary to build applications out of components.

A component model defines a set of standard mechanisms covering the different steps in component-based development. In the present there are already several industrial component models, as for example Java Beans, Enterprise Java Beans, Corba components, ActiveX controls and DCOM. The standards defined by component models allow for the interoperation of components that are independently produced, for example by different vendors.

A component model defines how components have to be described and how the are composed. Describing how components are composed involves specifying how to create a composite application, how to deploy components into the composite application and how to connect them.

The description of how to connect the parts, or *connection protocol*, involves on one side to specify when two parts can be connected, when they are "plug compatible". On th other side it involves defining how the connection is realized, if there is some glue code that needs to be produced, or some step of actions that have to be performed.

In the existing industrial component models, the mechanisms or protocols for composition are described by means of natural language specifications. This descriptions are verbose, ambiguous and of course non-executable.

A key point for the success of component technology is the availability of tools for building applications out of components. Such tools should assist the user in describing a composite application by assembling components and generate the code for the application. For example, a visual builder tool lets the user pick a set of components from a palette and place them in some kind of container. The user can configure these components by editing some attributes. Then the tool allows to connect the components by drawing lines between them. In this way the user produces a kind of description of the application. In the end, the tool generates the code for the described application.

The builder tool assists the user in the creation of the composite applications so that he builds well-formed component assemblies. It does so by restricting the constructs the user can make to those that are valid. In the example of the visual builder tool, where the connections are created by drawing lines, the tool only allows to draw lines between components that are plug-compatible. To guide the user in this way the tool needs to have knowledge that allows to determine whether a construction is valid.

The tool also generates the code for the composite application, so it also needs to know how to map the description of a component assembly into code.

Typically, a builder tool embeds knowledge about the composition mechanisms from a certain component model. It embeds the notions of what is a valid construct and a specific way of realizing the constructs. Thus the mechanisms available and their mapping into code are fixed, imposed by the builder tool. Unless the implementation of the tool is modified, it is not possible to change how the code is generated, to add a new kind of connection or even less, to use the tool with a different component model.

The goal of this thesis is to make the mechanisms for composition explicit and independent from the builder tool. Separating the mechanisms for composition from the builder tool allows to build generic tools capable of handling different composition mechanisms. It is then possible to modify the composition mechanism or define new ones and continue to use the same tool.

To achieve this goal we use meta-level descriptions of the components, the mechanisms for compositions and the composite applications. These descriptions support the operation of a builder tool in its two main functionalities: assisting the user in specifying component assemblies and the generating the code for the application.

To provide for the assistance to the user, the description of the components and component models must allow to determine whether a certain construction is valid, as for example whether a component can be used in a certain assembly, or if a pair of components can be connected.

To provide for the generation of the application code, the description of the component model must allow to interpret a description of a composite application and map it into code.

We use a logic programming language as a formalism to describe components, component models and composite applications. Components and component models are represented as sets of facts and rules. The kind of questions that need to be answered from the representation, such as whether a certain connection is valid, or what are the valid connections are very much like logic queries. A logic programming language provides good support for this kind of reasoning. A builder tool makes appropriate queries to the representation to determine what are valid compositions and guide the user to produce the description of a composite application.

The description of the composite application is a declarative specification, saying what are the components that make up the application, how they are configured and how they are connected. This declarative description can also be appropriately described using a logic language as a set of facts stating what are the components of the application and how they are configured and connected.

To generate the code we again use the technique of logic meta programming. In logic meta programming a program is represented as a set of facts. In our component based development, the facts that represent the code are deduced from the facts that describe the composite application and the facts and rules that describe the components and the component model. The description of the component model provides rules that specify how to generate the code for the application.

We defined an interface for the interaction between a builder tool and the logical representation of components and component models. This makes the tool generic, as it can work with any model that is represented according to the defined interface. It is then possible to modify the model, as for example add new ways of connecting components or modify how the code is generated, and continue to use the same tool. It is also possible to define a new component model and use the same tool to build applications in this new model.

To validate the approach, we developed a prototype builder tool that interacts with the logic representation of component models.

We show how to represent component models, components and composite applications using logic in such a way that it supports the operation of the builder tool. To illustrate the representation, we represent the Java Beans component model. We use this model because it is accessible and clearly defined.

We conducted some experiments to validate that the tool is independent from the component model. To show that the component model can be modified while continuing to use the same builder tool, we extended Java Beans with new connection protocols. To show that the tool can be used with a different component model, we represented a new component model with its own composition mechanisms. In both cases, the tool can be used without any change, thus providing evidence about the independence between the tool and the mechanisms for composition.

The report is structured as follows: in next chapter we survey research work related to component technology and identify a set of elements we consider the main features in a component model (from the point of view of the programming/composition language), setting up a conceptual framework for the work. Then we use this conceptual framework to analyze existing commercial component models. Chapter 3 presents the technique of logic meta programming. Chapters 4 and 5 present the contribution of this work. In chapter 4 we discuss the interface between a generic composition tool and present the prototype builder tool we built for validating the approach. We discuss the representation of components, component models and composite applications using a logic programming language to support the operation of a generic builder tool. We illustrate the representations by representing Java Beans. Chapter 5 discusses the experiments that show that the tool is independent from the composition mechanisms: the extension of Java Beans with new connection protocols and the representation of a new component model. Chapter 6 discusses some lines for future work and chapter 7 presents the conclusions.

Chapter 2

Components and Component Models

There are currently several industrial approaches to component technology. These approaches are very diverse and for this reason it difficult to analyze in a uniform way. In this chapter we will present a conceptual framework to be used to analyze existing component models. This framework will also be used as the basis for our representation of component models. We start by discussing research work in the area, then we present the features we use to characterize a component model and finally we analyze some existing component models using our conceptual framework. The last section discusses the Java Beans component in more detail, as it is going to be used as a reference for our representation of component models.

2.1 Related Research

In this section we will discuss research work on conceptual foundations for the area of software components, and addressing the problem of connecting components.

A compact definition of components is given by Nierstrasz and Dami [7] saying that

> *"a component is a static abstraction with plugs".*

Static means that the component is independent from its uses or instantiations, and can be stored in a repository. Abstraction means that a component is an encapsulated entity. Static abstractions can be very diverse in nature, as for example classes, mixins, functions, macros, procedures, templates and modules. The plugs are the points of interaction between the component and its environment. Examples of plugs are parameters, ports and messages. This definition of components is complemented by the definition of software composition which says that *"the process of constructing applications by interconnecting software components through their plugs"*. This definition points out one of the main problems in the area, which is how to connect the components.

Another definition of components by Szyperski[6] says that

> *"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 two definitions agree on components having well defined interaction points. Independence implies that the component is encapsulated, so encapsulation is also a common point. Composability is also a point present in both definitions. Although not present in the definition, the discussion in Szyperski's book also considers a component as the static software piece that is delivered independently of its uses or instances.

The notion of a component is also a main element in the area of software architectures. Software architecture addresses the description of the overall system structure. A software architecture is defined as a set of components and connectors. The term component is not used in the sense of a reusable software piece but in the sense of a computational entity, for example a filter or a server. The connectors describe the interaction between the components. A representative work in the area is the work by Allen and Garlan[10][11]. They consider the definition of a system in three parts: component and connector types, component and connector instances and configuration, and they define a language, Wright, to specify these parts. A component type describes a set of ports that represent the points of interaction between the component and its environment. A component type can be assimilated to the static components in the definitions discussed above and the ports to the plugs. The connector types define a set of roles, describing the obligations of the components that take part in the interaction and a glue, describing the coordination between the roles. The component and connector instances are the actual parts of the system, and the configuration describes the combination between the components and the connectors by mapping the ports in the components to compatible roles in the connectors. In Wright, the ports, roles and glue are defined using a variant of CSP.

This description allows to determine the compatibility between the ports and the roles they are mapped to, and in this way, to determine the validity of the connections between the components. A distinctive characteristic of this work is the representation of connectors as explicit entities, with the purpose of characterizing and reasoning about component interaction.

The representation of interactions as explicit entities has been applied in the area of software composition in the work on gluons[1]. Gluons are part of a framework for developing financial applications which allows for dynamic composition. The gluons are special objects whose responsibility is to mediate the cooperation between software components (in this case, objects). The framework is centered around the protocols and they are considered the primary reusable elements. The specification of a protocol consists in a set of roles, and interplay relation that specifies the compatibility between the roles, and a finite automaton that specifies the valid sequences of interactions between the participants. Each object in the environment may conform to a set of roles, and these roles together with the protocols will determine to which objects it can be connected. In this approach, most of the interaction code is factored out from the components into the gluons, the components keep the functionality necessary to hook the protocols such as exported values and event notification.

Scripting languages are a means of specifying software composition, by introducing, customizing and binding a set of components[2][13] . These languages are similar to programming languages, but with the special purpose of supporting the assembly of a particular kind of components. They also allow to program some additional, application-specific behaviour and to provide some glue code adapting components.

De Mey addressed the problem of building applications by visually assembling components[4]. She presents a framework for visual composition that can be specialized to specific application domains. The components have a set of ports, that define their composition interface. Components are connected between them by linking their ports. The validity of the connections between the ports is determined by a set of composition rules, that define the compatibility between port types. The framework is specialized for a certain domain by providing information about the components and the component rules specific to the domain. The information about the components include the composition interface and the implementation of the task the component performs, and a the visual presentation responsible for the component's visual display.

Vista is a prototype tool that implements this framework. It provides some abstract classes to be subclassed for implementing the component's behaviour and presentation. The compatibility rules are described using a textual notation that is interpreted by a composition model manager which is part of the framework. The composition model manager is like an oracle that maintains the compatibility rules and oversees the functioning of the framework.

This framework supports the definition of different composition models and in this sense, our approach is similar to this one. In De Mey's work the emphasis is put in the visual manipulation of components, and the specification of the composition rules is restricted to listing the port types and their compatibilities. It points out the necessity of alternative way for expressing composition models. This is the point addresses by this thesis, which concentrates on the definition of the composition rules, and proposes the use of a logic programming language for this purpose.

2.2 Conceptual framework

This section presents the conceptual framework we will use to analyze the existing component models, and that underlies the architecture we propose for representing component models. The conceptual framework is based on concepts from the work discussed in the previous section and on the study of existing component models. We will keep the discussion abstract in this section, then it will be illustrated by the analysis of the existing models according to this framework in next section. This gives an idea how well the models fit in the framework and thus is an informal validation of the choice of the elements in the framework.

The main characteristics of components we identify are

- components **can be composed** to build more complex components and applications.

- components are **encapsulated**: to use a component it is not necessary to access to its internals. Moreover, this access is not possible.

- components can **interact** with other components; as a component is not an application by itself, it is necessary to connect it and make it communicate with other components. The interaction with other components is done through **well defined interfaces**, the clearly specify what the component provides and what it requires from other components and from their environment. The points of interaction of a component with their environment are the *plugs*.

- a component is a static software piece. It can be used or **deployed** many times, in an application or in different application. This makes a distinction between the reusable component and its uses or instances. For the purpose of making the distinction clear, we will use the term *part* to refer to a particular use of a component in a determined application while the term *component* is used for the static entity that is delivered and can be stored in a component repository.

- components **can be configured** in a particular use by setting a set of parameters.

Deploying a component means linking it to a particular composite application. It is done when the developer is assembling the application. The deployment of a component creates a part, that is a description of how the component is used in the application. A part refers to the binding between a component and a container.

Given the above characteristics of components, the creation of an application by assembling components will involve deploying a set of component as parts of the application, configuring the parts and connecting the parts though their plugs. In order to use the components in this way we need descriptions of the components that specify how they must be deployed in a composite application and configured, and how they can be connected to other components.

A **component model** defines a standardized way for describing components, and mechanisms for deploying, configuring and connecting them. As we will see in the analysis of the existing component models, the component are not deployed "in the air", but hosted in a special entity which we call the *container* which provides an execution environment for the component instance. A component model defines

- a way of describing the components and their plugs

- a way to create a container to host the components

- a way to use a component as part of an application, and link the part to the container

- a way to configure the parts

- protocols to connect the parts through their plugs

The first mechanism will be used by the component developer, the other four will be used by the application assembler.

The main focus of this dissertation is on the mechanisms for connecting the parts and determining which are the valid connections. A component model may provide different kinds of connections. We call the description of one kind of connection a *protocol*.

In chapter 4 we will show how we can describe a component model using a logic programming language. We will make the descriptions of the component models explicit allowing to modify the protocols of a component model, or extend the model with new protocols.

2.3 Component Models

Existing component models are very diverse, and they have different particular goals. According to their goal, the notion of component is different and they stress a different aspect of the composition process. In this section we will analyze existing component models according to the main features identified in the conceptual framework: which are the kind of plugs the component may define, which is the container, how are the components instantiated and linked to the container, how they are configured and which are the connections. For each approach we will present the goal, how the elements in the conceptual framework match in it and which element is stressed.

In each case, we will present the goal of the approach, how the mechanisms map in the approach and what mechanisms are stressed in the approach. The purpose of this section is not to present each model in detail, but to characterize them within a unified framework and validate the framework by this characterization.

2.3.1 Component Models for Visual Composition

The aim of Java Beans[5] and ActiveX Controls[6] is to define components that can be manipulated visually by builder tools to build applications. This kind of models correspond closely to what we can expect as building an application by assembling components: the user has a tool box of components to pick and put into a kind of canvas, connects them by drawing lines between them and then the tool generates the code for the application.

In these models, plugs are methods, properties and events. Methods are services provided by the component. Properties are exported values, such as for example the contents of a text field. Events are a mechanism for notifying that something interesting has happened. For example, the selection of an item in a list raises an event. Objects willing to be notified of an event have to register with the object emitting the event.

Also properties can be used for configuring the instances of the components, such as for example the color of the background.

Both kinds of components are integrated into a container, in the case of ActiveX Controls an ActiveX container and in the case of Beans an applet, an application or a normal class.

These components are configured by setting their exported values or properties. They are connected using events. A connection is realized by registering an instance of a component as willing to be notified from events from another.

ActiveX Controls evolved from Visual Basic controls. Their provided operations are defined as COM interfaces, which are pointers to function tables .

The Java Beans component model has been defined from scratch. It is described in a technical specification [5]. The specification defines an API provided by a set of classes and a set of programming conventions that the component developer must follow when writing his beans and that will be used by the builder tools to manipulate them. This model is simple and accessible, yet includes the main features of a component model. For this reason, we use it as a reference for the definition of the component model architecture (chapter 4). We discuss it in more detail in section2.4.

2.3.2 Distributed Systems

In distributed systems the accent is put in the communications of objects through a network. In this kind of models a component allows to create an object that provides services over the network.

The main mechanism is the connection between components. Connection in this models is the communication by the invocation of operations in a remote component.

The communication infrastructure acts as a container for the components. The components are required to register with the infrastructure the services they provide. The deployment of the component involves creating the object that provides the service and registering it to the container.

The connection between components involves obtaining a reference to the object and then invoking the operation. The reference is obtained by interacting with a name service that allows to find an object based on a symbolic name, or a yellow-pages service that allows to find an object based on its characteristics. DCOM, CORBA and Jini belong to this kind of models; they all share the general interpretation of the features in a component model described above. Next we will discuss aspects more specific to each model.

DCOM

COM and DCOM are the component models from Microsoft. The goal of COM is the communication between applications. DCOM extends the COM model with communication between different machines. DCOM and COM do not specify what a component is. It is not necessarily an object, and it can be implemented in any language.

The plugs of COM/DCOM components are interfaces, each interface defines a set of operations or services provided by the component. An interface is a pointer of a table of function pointers. Another kind of plugs are outgoing interfaces, that are interfaces for event notification. Associated to each outgoing interface, the component has to provide for the registration of objects that implement the outgoing interface and are willing to be notified of the events.

CORBA

The goal of Corba is to allow the communication between distributed objects implemented in different languages and executing in different platforms. CORBA defines a language for specifying the services an object provides: the Interface Definition Language (IDL). An interface consist of a set of operations and attributes. The interfaces are the plugs of the components. Every service invocation is mediated by the ORB (Object Request Broker), which is responsible for locating the object that implements the service and communicate the request. The ORB provides an execution environment that hides the diversity of languages and execution platforms[6] [12]. The ORB corresponds to the container in our characterization of component models. The components providing services must be connected to the ORB, publish their references through the naming service and put themselves to the listening of service requests. This process corresponds to mechanism of adding a part to a container in our characterization.

JINI

The goal of Jini [16][17]is to federate a set of resources into a single distributed system. Resources can be software of hardware devices or a combination of the two. Resources can join and leave the system dynamically. Jini is based on Java and assumes that Java is the programming language.

A Jini component implements a service; it may be a computation, a s-torage or a hardware device. A service is described as an interface in Java, which constitutes the plug of the component. Jini defines two protocols, Discover and Join, that allow a service to become a part of the system. They consist in finding a yellow-pages service and registering to it. Another protocol, Lookup, specifies how to obtain a reference to a service provider through the yellow-pages service. The obtainment of the reference followed by the invocation of the service constitute the connection between the components. In this model, the infrastructure for remote communication and the yellow-pages service constitute the container.

2.3.3 Server Components

The goal of server component models is to facilitate the development of server components. Server components are complex because they have to take into account non functional aspects as security, transaction management and persistence. These models specify a framework in which the component developer only concentrates on the functionality of the component, while non functional aspects are provided by a container in which the component instance executes. This models are Enterprise Java Beans[8] and the proposal for CORBA[9] components.

The main entity in server component models is the container which provides important services for the execution of the component instance. The container is complex and meant to be provided by a provider with system-level programming expertise.

A component in this models is a class implementing a set of business methods. In EJB the class must implement some Java interfaces defined by the EJB library.

In the EJB model the component specifies a set of requirements for the container, like the kind of transaction and security management it requires, by means of a "deployment descriptor". The the deployment of an EJB as a part of the container consists in setting the environment properties according to the requirements specified by the EJB in the deployment descriptor, generating additional classes for stubs and skeletons for distribution, generate some more classes to be used by the container to manage the EJB at runtime and register the service to the naming service.

The plugs of the EJB components are the business methods they define. The connection is the invocation of the business methods by the clients. To establish the connection, first the client locates a reference to the component by interacting with a naming service and then it invokes the methods. Method invocation is mediated by the container.

An EJB can be configured by adapting its business logic through wrapper methods. The EJB architecture provides hooks where to insert this wrapper methods. It can also be configured by modifying its deployment descriptor and setting the environment properties.

The proposal for CORBA components builds on top of the CORBA architecture for communicating distributed objects written in different languages and across different platforms. This proposal is said to be aligned to the EJB model. For the moment it is a proposal, and to our knowledge no containers exist so far.

In CORBA components, the plugs are provided and required interfaces (called dependencies), events and properties. The plugs are described using the CORBA IDL. The component also may specify some requirement from the container.

The component can define an interface for configuration.

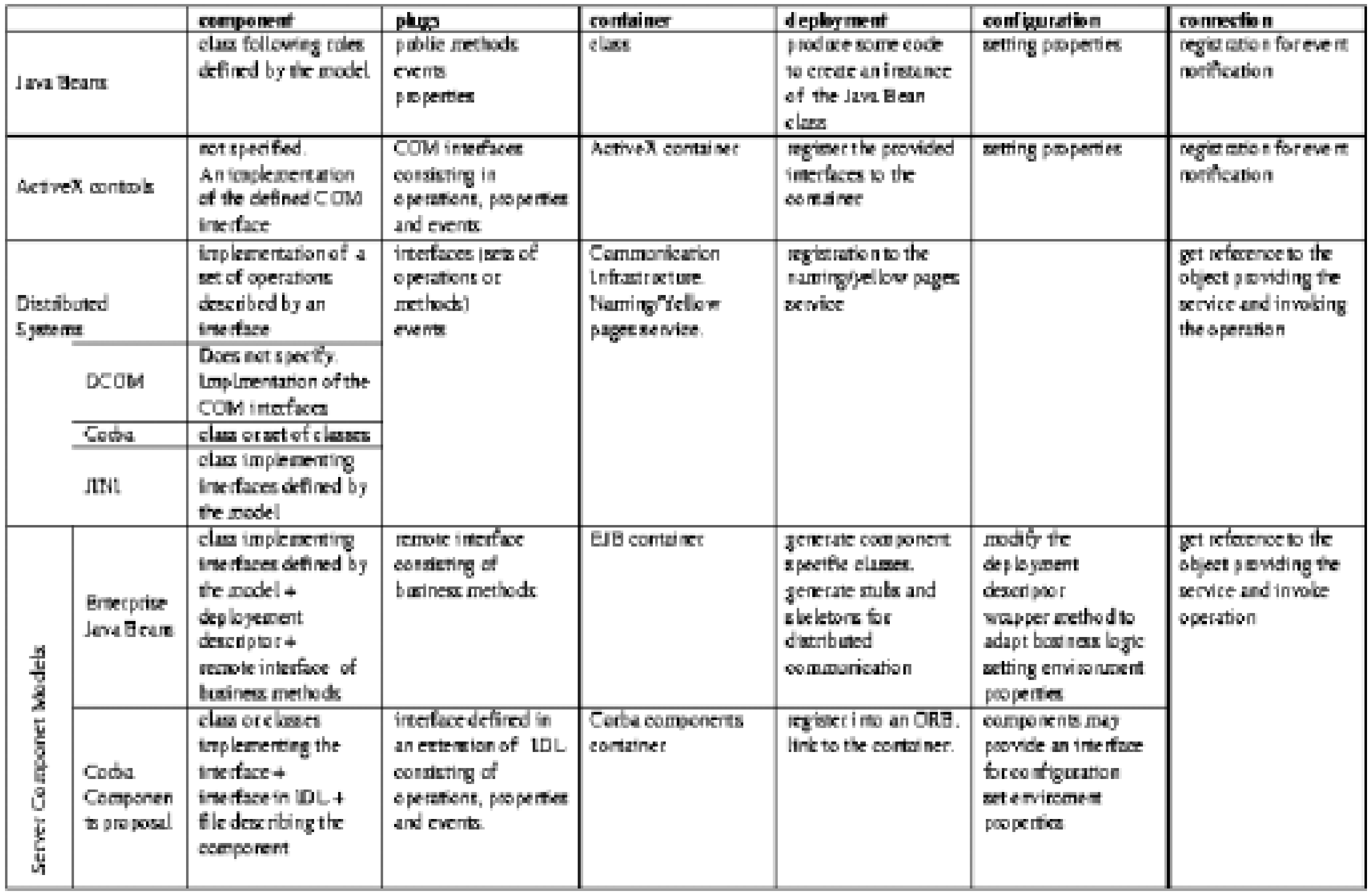
The deploying the component involves choosing a machine and a process for the component to execute, register to the CORBA ORB and configure the properties that the component requires from the container.

The connections in this model are of two kinds: linking required to provided interfaces and event emitters to event consumers and searching a service provider through a naming service and invoking the service.

2.3.4 Summary and Discussion

In this section we analyzed some existing commercial component models according to the main features identified in the previous section. Table 2.1 summarizes what is the mapping of the features in each of the models. Although sometimes very different in nature, the main features have a correspondence in each of the models. In most of the models, a component corresponds to a class. This class may sometimes be a facade for a set of classes. According to the model, this class has some conventions to fulfillentity, like implementing some interface in a library (EJB) following certain programming conventions (Java Beans) or providing a definition of the operations in terms of an interface definition language (Corba). In the Microsoft models (ActiveX and COM) the components do not necessarily correspond to a class, it is not defined what components are. It can be any implementation of an interface.

In all the models there are plugs defined as operations or methods. In



		component	plugs	container	deployment	configuration	connection
Java Beans		class following rules defined by the model	public methods events properties	class	produce some code to create an instance of the Java Bean class	setting properties	registration for event notification
ActiveX controls		not specified. An implementation of the defined COM interface	COM interfaces consisting in operations, properties and events	ActiveX container	register the provided interfaces to the container	setting properties	registration for event notification
Distributed Systems		implementation of a set of operations described by an interface	interfaces (sets of operations or methods) events	Communication Infrastructure. Naming/Yellow pages service.	registration to the naming/yellow pages service		get reference to the object providing the service and invoking the operation
	DCOM	Does not specify. Implmentation of the COM interfaces					
	Corba	class or set of classes					
	JINI	class implementing interfaces defined by the model					
Server Component Models	Enterprise Java Beans	class implementing interfaces defined by the model + deployement descriptor + remote interface of business methods	remote interface consisting of business methods	EJB container	generate component specific classes. generate stubs and skeletons for distributed communication	modify the deployment descriptor wrapper method to adapt business logic setting environment properties	get reference to the object providing the service and invoke operation
	Corba Components proposal	class or classes implementing the interface + interface in IDL + file describing the component	interface defined in an extension of IDL consisting of operations, properties and events.	Corba components container	register into an ORB. link to the container.	components may provide an interface for configuration set environment properties	

Table 2.1:

some of the models the operations are grouped in set of related operations or interface. Some other common kind of plugs are properties of exported values and events. Events are a means for decoupled communication, as the entity that emits the event does not need to know in advance the receiver of the event. For this reason they provide a good means for late connection, and are used in most of the models.

Connection involves following several steps or conditions, like registering for an event, or finding a reference to a service provider. In the end, the actual communication is done through operation invocation.

The main differences between the models concern the container and the deployment of the components. The container varies from just a class, to a complex communication infrastructure. All the models defined some way in which a component becomes a part of the composite application, but the mechanism can be very different. For example in Java Beans, the component is linked to the container by defining a variable that host the runtime instance of the component and providing some code that creates that running instance, in EJB the component is linked to the container by setting environment properties, creating a set of auxiliary classes and registering the component services to a name service.

From the analysis in this section we conclude that the identified main features from component models provide a first step towards a characterization of component models. A deeper study is needed to achieve a general foundation of component models.

Most of the component models are strongly based on object oriented programming. A component usually corresponds to a class. This identification blurs the distinction between object models and component models. The shift to components puts the focus on connection and configuration mechanisms. Connection is based on communication through message passing or operation invocation, but with some structure or protocols that allows to decouple the caller and the callee. Events is one of this protocols that structure on message passing. Another is access through naming or yellow pages services. Configuration or adaptation in component models is done by setting values for properties or exported attributes, in contrast to the adaptation through inheritance typical from object-oriented programming.

2.4 Java Beans

The Java Beans component model [5] is clearly specified and accessible. At the same time it contains the main features we identified in a component model. For this reason we use it as a reference in the representation of component models. In this section we present in more detail the aspects of Java Beans that are relevant for this representation.

According to the specification of Java Beans [5], "a Java Beans is a reusable software component that can be manipulated visually in a builder tool". Beans can be simple graphical widgets such as buttons or more complex components such as database viewers or chart drawers. The Java Beans component model defines a set of rules that must be followed for writing a component so it can be manipulated by builder tools. This kind of tools allow to build an application by placing a set of beans in a container, configuring then and connecting them.

The plugs that allow to connect beans are methods, events and properties. Methods are just normal, public methods. Events are a mechanism to notify that something relevant has happened. Properties are exported values that can be read or written by invoking appropriate bean's methods. Properties are used for connection and for configuration. A bean can be configured by providing values for its properties.

We begin by discussing an example of the construction of an application out of beans, using a builder tool. Then we explain how beans are written and how they are manipulated in the application code.

2.4.1 Building an application by assembling beans

As an illustration of how an application using beans can be constructed and how the tool manipulates the components, let's consider building a small application using the BeanBox. The BeanBox is a simple tool provided by Sun that allows to create applications using beans. The purpose of the tool is to illustrate how to compose beans rather than to build real applications.

The application we are going to build is a directory browser that allows to walk through the directory hierarchy. It's interface is shown in figure 2.1. The application consists of four parts: a text label, a list box , a button and a non-visual component, directory explorer that explores the directory hierarchy. The text label will show the path of the current directory. The list box shows the contents of the current directory and allows to chose a directory to explore. The button allows to go up in the directory hierarchy. The directory explorer maintains the current directory and knows its contents, and can go up and down in the hierarchy.

To build this application we have to pick up the different parts, configure them and connect them. The BeanBox lets us perform all these steps and in the end it produces the code for the application.

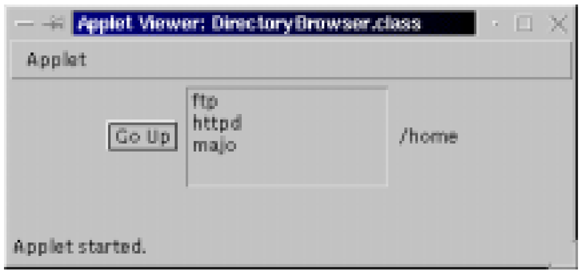


Figure 2.1: An application built from beans: a Directory Browser

When the BeanBox is started it shows a container to put the beans, a tool box with the available beans and a property sheet.

As a first step, we create the parts choosing the beans from the tool box and placing them in the container.

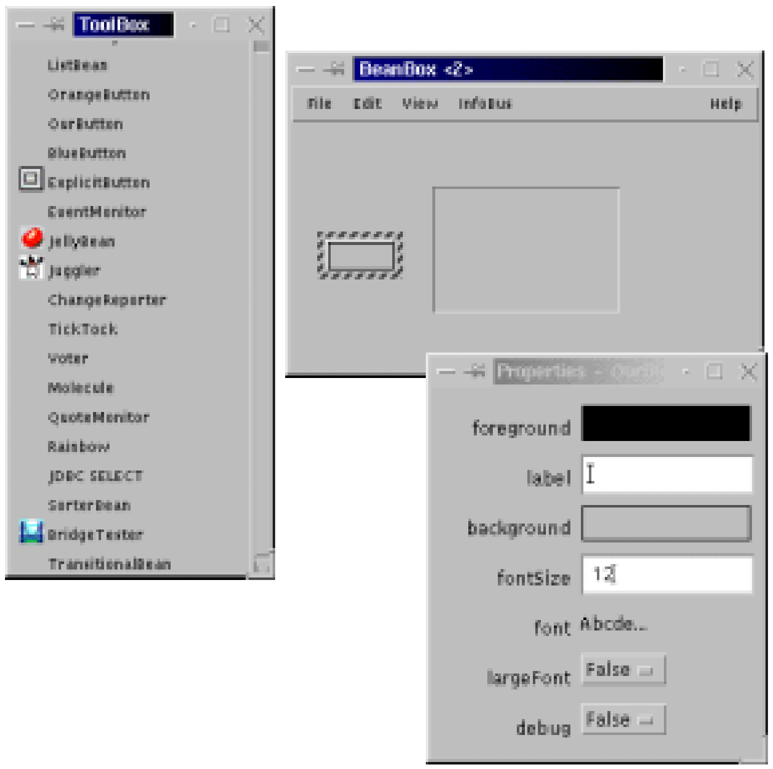
Next, we configure the parts. A bean can be configured by providing values for its properties. When a bean instance in the container is selected, the property sheet shows its properties and allows to configure them by editing their values. For example, we select the button in our example and configure it by providing text “Go Up” for the label property. (Figure 2.2)

Having created and configured the parts, we create the connections between them. The BeanBox allows to create two kinds of connections between the parts. The first kind of connection links an event in a part to a method in another part. An event is a notification that something interesting has happened. A method is just a normal method. Linking an event to a method means that the method will be invoked when the event occurs. The other kind of connection links two properties or exported values: a “source” and a “target” property. The meaning of the link is that whenever the value of the source property changes, this change will be reflected in the target property.

In our example, the Directory Explorer has a property `currentDirectory` whose value is the path of the directory being explored. We want to link this property to the text property of the label, so that the label shows the path of the current directory. To create this connection in the BeanBox we have to follow several steps:

1. Select the part that is the source of the link. In our example the source part is the DirectoryExplorer.

2. Choose from a menu what kind of connection we want to build. In this case we select properties.



foreground	
label	
background	
fontSize	12
font	Abcde...
largeFont	False
debug	False

Figure 2.2: Building an application using beans: configuring a bean

3. Then we choose the source property from the list of properties of the selected part. In this case we chose the property currentDirectory.

4. Draw a line to a target part. In our example the target part is the label

5. Choose the target property from the list of properties of the part. In this case we choose the text property.

In the same way we create the other two connections between properties. We link the contents property in the DirectoryExplorer, which holds the list of the files contained in the current directory, to the items property in the List, which holds the elements that are shown by the list. Finnaly, we link the selectedItem property from the list bean, which holds the selection made by the user, to the currentDirectory property in the DirectoryExplorer, so when the user selects a directory the DirectoryExplorer goes to explore that directory.

The last connection links the button to the directory explorer, so that when the button is pressed, the directory explorer goes one level up in the directory hierarchy. The directory explorer has a method goUp that makes it browse the upper directory. Pressing the button generates an event called actionPerformed. We want to link the occurrence of this event to the invocation of the goUp method in the directory browser. This connection is created using the BeanBox in an analogous way as the connection of properties: first we select the button and we choose to connect it trough events and select the actionPerformed as a source of the link; then we draw a line to the directory explorer and choose the goUp method from the list of the directory explorer's methods.

After completing the connections, we ask the BeanBox to generate the code for the assembled application. The BeanBox generates a class. The class is an applet with the layout of the BeanBox. This class acts as the container for the beans, and contains the code necessary for deploying, configuring and connecting the beans.

2.4.2 Java Bean Components

As a component, a bean provides a way of being deployed as a part of a composite application, a way of being configured and a way of being linked to other components. The Java Beans component model standardized this mechanisms, so the components do not define them all by themselves, but they define the information necessary for the standard mechanisms to operate. Builder tools use this information an mechanisms to assemble the beans.

The Java Beans specification establishes a set of rules or conventions that beans conform to so that they can be manipulated by builder tools.

A Java Bean is implemented by a class that follows a set of rules defined by the Java Bean specification. Alternatively, a bean could consist of a set of classes, but in this case there is a class that acts as a facade for the rest and complies to the rules.

A bean must provide support for instantiation by defining a no argument constructor. Connection is supported by providing a set of methods, events and properties. Methods are normal public methods. Events and properties are implemented using methods that follow special rules. Properties also support configuration.

Events

Events are a mechanism for notifying state change notifications from a source object to one or more target objects. Events provide a means for connecting bean instances, by linking events sources to event targets or *event listeners*.

The event mechanism consist in a publish and subscribe policy. An event source provides a mechanism that allows objects interested in being notified of the occurrence of the event to subscribe to it. The emission of the event consists in the event source invoking a specific method in each of its registered subscribers.

Each kind of event notification corresponds to a different method. Related groups of notification methods are grouped in event listener interfaces. For example, the methods for notifying mouse events are grouped in the `java.awt.event.MouseListener` interface which consists in the methods `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`.

Event listener interfaces are identified as such by extending the `java.util.EventListener` interface. This is an empty interface whose purpose is to serve as a marker or annotation to distinguish listener interfaces.

A class of objects that are meant to be targets of a certain kind of events implement the corresponding listener interface.

A class of objects that act as event sources are identified because they provide methods for registering and unregistering event listeners. These methods must follow some programming conventions that allow builders to detect them. An event source is identified by the presence of a pair of methods of the form:

```
public void addEventListenerType(EventListenerType a)
public void removeEventListenerType(EventListenerType a)
```

EventListenerType is an event listener interface. It must inherit from java.util.EventListener and its name must end with “Listener”.

The event notification may convey some information. For example, in an event that notifies of a mouse click, we are interested to know the coordinates where the mouse was clicked. The information associated to the event is encapsulated in an event state object which is the sole argument to the notification method. The class of an event object inherits from java.util.EventObject.

The following table summarizes the rules for defining an event Foo:

element	convention
class of the event object	FooEvent extends java.util.EventObject
event listener interface	FooListener extens java.util.EventListener
method in the listener interface	public void methodname(FooEvent e)
method for registering a listener	public void addFooListener(FooListener l)
method for unregistering a listener	public void removeFooListener(FooListener l)

Properties

Properties are exported attributes. The bean class defines special methods for reading or writing a property. Depending on the methods provided for accessing a property, it may be read-only, write-only or read-write: a readable property provides a getter method for reading its value and a writable property provides a setter method for updating the value. The conventions for setter and getter methods are (considering a property Bar of type PropertyType):

```
void setBar(PropertyType value);
PropertyType getBar();
```

Properties are used for configuration and connection. Configuration is done by providing values for writable properties. The mechanism for connection using properties is based on events: a bean may provide notification of the change of the value of some of its properties. The change of a property's value produces an event. Objects interested in being notified of property changes register as listeners of this event. These properties whose changes produce events are called bound properties.

Java Beans defines an event and a listener interface of property changes. The listener interface is PropertyChangeListener. As in the general case of events, a bean which has bound properties an so emits the PropertyChangeEvent must provide a pair of methods to register and unregister event listeners.

A single event is used for notifying changes of all the properties in a bean. To distinguish which is the property that actually changed, an event object is passed as an argument. This event object carries information about the source of the event, the name of the updated property and its old and new values.

The property change event and the event object allow to build the connection between properties described in the previous section. In the following section we will explain how the connections can be implemented.

The following table depicts the programming conventions for properties:

element	convention
getter method	PropertyType getFoo()
setter method	void setFoo(PropertyType value)
class of event object for notifying property changes	PropertyChangeEvent
change listener interface	PropertyChangeListener
notification method	void propertyChange(PropertyChangeEvent e)
method for registering a listener	void addPropertyChangeListener(PropertyChangeListener l)
method for unregistering a listener	void removePropertyChangeListener(PropertyChangeListener e)

2.4.3 Assembling beans

Beans are meant to be assembled using a builder tool which generates the application code. A normal user of a builder tool is not concerned about how this code implements his application. However, to make a tool or fine tune an existing tool (which allows to be modified), it is necessary to know how to implement the deployment, configuration and connection of the beans.

A composite application built out of beans is implemented as a class that hosts beans instances. This class acts as a container, and provides for the creation of bean instances, configuration and connection.

If a bean is used as a part of an application then an instance of the bean corresponding to that part has to be created. An instance of a bean is created by invoking the method instantiate in a support class, Beans, that is provided by the Java library. For example, the instantiation of the directory explorer in our example is done as follows:

```
myExplorer = (DirectoryExplorer)Beans.instantiate(cl,
                                  “DirectoryExplorer”);
```

In the previous code, `cl` is the class loader, which is obtained using Java introspection mechanisms.

The configuration of the beans is achieved by invoking the setter methods of its writable properties. For example, the configuration of the button in our example that provides a label for the button is implemented as

```
upButton.setLabel(“Go Up”);
```

As discussed in the previous section, support for connection is the event notification mechanism. Making up connections involves subscribing event listeners to event sources. This registration is done by invoking the method for registering listeners of a certain kind of event in the event source. For example, the registration of an object interested in being notified when the go up button is pressed, is done as follows:

```
upButton.addActionListener(aListener);
```

The definition of events and properties provide the plugs or hooks for building the kind of connections we used in the construction of the example application. This kind of connections are protocols provided by the tools on top of the basic plugs defined in the specification. The implementation of these protocols involves some glue code that realizes the links.

One of these connection links event to a method so that the method is called when the event occurs. In this connection, the target part may not implement the listener interface corresponding to the event, and the method may not be the notification method for the event. A possible way of making this connection is creating an adaptor class that implements the listener interface, and where the notification method for the event we are connecting invokes the target method. The connection is realized by creating an adaptor object and registering it as an event listener in the source.

The other connection we considered links a source property to a target property so that the changes of the value of the first property are propagated to the target property. This connection is built using the property change event and the associated event object. One way of implementing this connection is creating an adaptor object that listens to property change event and updates the target property when the source property is modified. When it is notified of a property change, the event the adaptor object consults the event object to determine if the modified property is the property it is interested in. If the modified property is the source of the connection, then the adaptor object invokes the setter method of the target property with the new value of the source property, which is contained in the event object.

The implementation of the connections we have just discussed is only one of the possible ways in which they can be made. Different tools may provide different ways of building the connections.

2.5 Summary

In this chapter we set up the conceptual framework that underlies our representation of component models. We started by surveying some research work in the area that addresses the conceptual foundations of component technology and the problem of connecting components. Based on this study and on the study of existing component models, we identified what we consider the main elements and mechanisms in a component model. Then we used this features to analyze the existing component models to see how well they fit in the conceptual framework. We could characterize the different models according to the chosen features, thus informally validating the choice. Finally, we explained in more detail the Java Beans component model which we use as a reference for the representation of component models. We chose this model because it is an industrial, well accepted component model which is accessible and clearly specified and presents all the main features of component models.

Chapter 3

Logic Meta Programming

This dissertation proposes the use of logic meta programming for describing component models, components and composite applications and to generate application code. We adopt the view of logic meta programming of [14]. In this chapter we present the approach and the logic meta programming system we used for implementing the experiments.

3.1 Logic Meta Programming

A meta program or meta-level program is a program that manipulates programs. The manipulated programs are called base programs or base level programs. The language in which the meta program is written is called the meta language, and the language in which the base programs are written is called the base language.

The meta program "reasons" about the base programs. To do this, it is necessary to represent the base language programs using data structures of the meta language.

In logic meta programming the meta language is a logic programming language. The base-language programs are represented by a set of logic facts. A function called representational mapping associates each program with its meta level representation. The representational mapping determines the set of facts that represent a base-level program.

A logic program indirectly represents a set of facts: the set of facts that can be proven from its facts and rules. Thus, the set of facts that represents a base program can be represented by a logic program that allows to deduce them. Therefore, base programs are indirectly represented by logic programs. This situation is depicted in 3.1. This representation is powerful because it allows to represent "patterns" of code with holes using rules, and then fill in the holes and deduce the facts that represent the actual code.

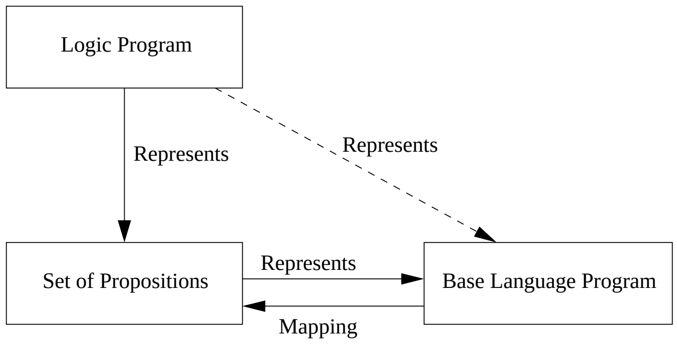


Figure 3.1: Base program representation

The representational mapping determines which aspects of the base program are made explicit and available for manipulation by the meta program. For this reason, the choice of the representational mapping is a central issue in the design of a logic meta programming system.

3.1.1 Logic Meta Programming System

The architecture of a logic meta programming system is shown in figure 3.2. The main parts of the system are a logic system and a code generator. The user of the meta-programming system interacts with a code generator which outputs the base language code. The code generator retrieves the information necessary to produce the base program by querying the logic system to consult the facts that represent the base program. The logic system consist on a database that stores the logic program and a logic inference engine. The facts that represent the base program are inferred by the logic engine from the rules and fact stored in the rule base. The code generator is a kind of inverse of the representational mapping. It produces a base-language programs from the set of facts that represent it. The code generator makes appropriate queries to the logic system and outputs the result as base-language code.

3.2 TyRuBa

TyRuBa is a logic meta programming system based on a simplified Prolog extended to facilitate the manipulation of Java code. The TyRuBa language

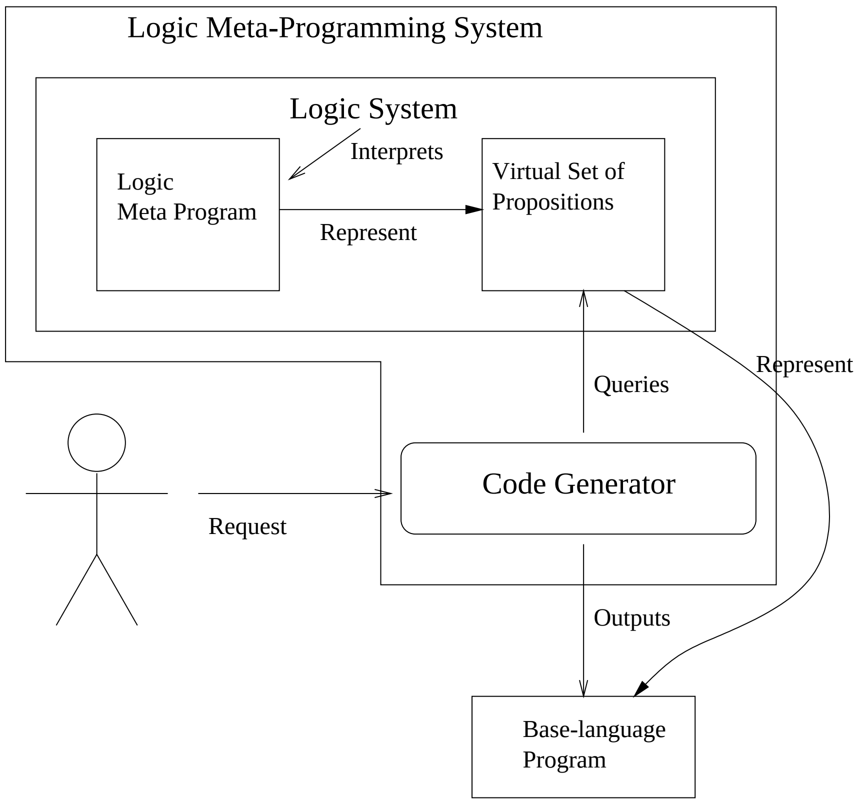


Figure 3.2: Architecture of a logic meta programming system

has a quoting mechanism that allows pieces of quoted Java code to occur as terms in logic programs. Conversely, logic variables and terms can also occur within a quoted piece of code. This last feature is very useful in the implementation of code generators, as variables can be used as place-holders to create code templates.

3.2.1 The TyRuBa Language

This section briefly presents the TyRuBa language. For more details on the language we refer to [14].

A TyRuBa program is a set of logic rules, facts, queries and directives.

```
TyRuBaFile :: = (Rule | Fact | Query)*
```

The syntax and semantics of rules and facts is the same as Prolog. A rule consist of a conclusion term followed by :- followed by a condition expression. A fact is a rule whose condition is always true and thus is omitted. Queries do not affect the generated code.

```
Rule ::= Predicate “:-“ Expression “.”
Fact ::= Predicate “.”
Query:: “:-“ Expression “.”
```

Predicates and expressions are also syntactically and semantically similar to Prolog predicates and expressions.

```
Predicate ::= Constant[“(“ TermList”)”]
```

The version of TyRuBa used in our experiments is an extension of the one presented in [14]. In this version, a predicate is any term.

Expressions are disjunctions and conjunctions of predicates:

```
Expression ::=Disjunction
Disjunction ::= Conjunction (“;” Conjunction )*
Conjunction ::= SimpleExpression (“,” SimpleExpression)*
SimpleExpression::= Predicate
                    |(“Expression”)”
```

Terms in TyRuBa are a bit different from Prolog. The main difference is a special kind of term that allows pieces of Java code to appear as data in logic programs. Variables, constants and Compound terms follow syntactic rules that allow to avoid conflicts when they appear inside quoted code.

```
Term ::= Variable | Constant | CompoundTerm |
         QuotedCode | List
```

Variables and constants are identifiers. Variables are distinguished from constants by a leading "?". `?pepe`, `?X` are variables; `pepe`, `12`, `X` are constants.

Compound terms are written using "<" and ">", instead of using "(" and ")" as in Prolog. This allows to distinguish compound terms occurring in quoted Java code from function calls. `Array<String>`, `?term<?bar, foo,<11,2>>` are examples of compound terms. We found compound terms very useful for expressing programming conventions as the ones in Java Beans.
List in TyRuBa are as in Prolog:

```
List     ::="[" ListRest
ListRest ::= "]" | Term ListCdr
ListCdr  ::= "," Term ListCdr
           | "|" Term "]"
           | "]"
```

Examples of lists are `[a,b,c]`, `[a|[b,c]`, `[a|[b|[c|[]]]]`, which all represent the list whose elements are `a`, `b` and `c`.

QuotedCode is a special kind of term that represents a piece of quoted Java code. It is a string between a pair of balanced "{" ,"}". It is composed of a list of Java tokens, variables, constant terms and compound terms. Nested "{" and "}" are just another kind of tokens and do not introduce nested quoting.

TyRuBa also provides some meta predicates, i.e. predicates about terms or expressions

`FIRST(`*expression*`)` evaluates the expression and retains only the first solution.

`FINDALL(`*expression*`,`*term*`,`*variable*`)` finds instances of term for which expressions is true and collect this instances in a list that is bound to variable.

`NODUP(`*term*`,`*expression*`)` produces all the solutions to expression, eliminating those where the instantiation of term is the same as the instantiation of term in a previous solution. The `NODUP` predicate is useful for code generation.

`NOT(`*expression*`)` fails when expression succeeds.

3.2.2 Code Generation

The TyRuBa language described so far is a logic programming system. To complete a logic meta programming system we need a code generator that generates the base programs from their logic representation. A code generator is associated with a representational mapping: it generates the code for programs represented using the representational mapping for which it is implemented.

In TyRuBa the code generator invokes the logic system to evaluate a query to obtain the code to be output. To obtain the declaration of a class or interface X, the code generator performs the query

```
:-generate(X,?code).
```

The solutions to this query bind the generated code to ?code. There may be zero, one or more solutions to this query. No solutions means that the meta program does not specify an entity of name X. In the case of many solutions, the system takes the first one.

It is possible to define different representational mappings and implement their corresponding code generator. A code generator is implemented by defining the generate predicate in terms of the facts in the representational mapping. The generate predicate is defined as rules and facts in TyRuBa itself. The system loads the code generator from an initialization file. In next section we will present a specific representational mapping and the corresponding code generator. This representational mapping is the one used for the experiments in this dissertation.

3.2.3 Representational Mapping

The representational mapping associates a base program with a set of facts that are its logic representation. The representational mapping determines which aspects of the base program are made explicit (reified) in the representation, and can thus be easily consulted, manipulated and generated by meta programs. The representational mapping we use in this dissertation reifies information about methods, instance variables and constructors in classes and interfaces, and the relationships between classes and interface.

The presence of a class declaration is represented by the fact

```
class_M(?C).
```

The representation of an interface is similar

```
interface_M(?I).
```

The presence of interface ?I in the implements clause of class ?C is asserted by the fact

```
implements_M(?C,?I).
```

The presence of type ?T1 in the extends clause of type?T2 is asserted by the fact

```
extends_M(?T2,?T1).
```

If class ?C is abstract, this is represented as

```
abstract_M(?C).
```

If type ?T is public, this is represented as

```
public_M(?T).
```

The presence of a method ?m with return type ?R and argument types ?A1,?A2,...,?An in type ?T is represented by

```
method_M(?T,?R,?m,[?A1,?A2,...,?An],{...declaration...}).
```

The last argument is the code of the method declaration as it will appear in the generated code for the class or the interface.

A constructor in class ? C with argument types ?A1,?A2,....?An is represented as

```
constructor_M(?C,[?A1,?A2,....?An],{...declaration...}).
```

The declaration of a variable ?n of type ?T in class ?C with initializer ?i is expressed as

```
var_M(?C,?T,?n,?i).
```

Figure 3.3 shows an example of a class and its representation using this mapping.

The code generator is implemented by defining the generate predicate in terms of the the predicates of this representational mapping. The implementation of the code generation defines a rule for generating the code of a class and a rule for generating the code of an interface. The code for a type is generated by querying the rule base for the different pieces that make up the type.

The following rule generates the code of a class:

```
class OurButton extends Component{
    OurButton(){
       super();}
    void setText(String t){
         this.text = t;}
   String getText(){
        return this.text;}
      :
      :
}
```

```
class_M(OurButton).
extends_M(OurButton,Component).
constructor(OurButton,[],{
      OurButton(){
         super();}
   }).
method_M(OurButton,void,setText,[String]),{
      void setText(String t){
          this.text = t;}
    }).
method_M(OurButton,String,getText,[],{...}).
:
:
```

Figure 3.3: A Java program and its logic representation

```
generate(?class,{
?public ?abstract class ?class
  ?extendsClause
  ?implClause
  { ?features }
}) :-
  class_M(?class),
  CG_public(?class,?public),
  CG_abstract(?class,?abstract),
  CG_extendsclause(?class,?extendsClause),
  CG_implementsclause(?class,?implClause),
  FINDALL(NODUP([?class|?feature],
                CG_feature(?class,?feature,?implem)),
          ?implem,
          ?features).
```

The predicate `CG_public` queries the rule base to determine if `?class` is asserted to be public or not and generates the keyword "`public`" or an empty String. The predicate `CG_abstract` is similar to `CG_public`. The predicate `CG_extendsclause` generates the extends clause from all the names in the database that are in an extend relationship with the class.

The definition of the predicate generate for interfaces is much the same as the definition for classes.

3.2.4 Derived Information

The representational mapping reifies the information about the features that are directly declared in a class or interface and not about the features that are inherited from its super types. Similarly, it only reifies the relation between a class and its direct superclass and extended interface, and not the general subtype relation. Often when a meta program needs to have more general information, as for example if a class has a certain method no matter if the method is declared in the class or inherited from a superclass. This kind of information can be deduced from the information reified in the representational mapping. Below we discuss some predicates providing higher level information. See [14] for their implementation.

The predicates `extends` and `implements` extend the `extends_M` and `implement_M` taking into account the inheritance relations. `extends(?subclass, ?superclass)` determines if `?subclass` inherits directly or indirectly from `?superclass`. `implements(?class, ?interface)` determines if `?class` implements `?interface` either because it implements it directly or it implements an interface that extends `?interface`, or if one of its direct or indirect superclasses implements `?interface` or and interface that extends `?interface`.

The predicate `subtype` implements the subtype relation between classes and interfaces as defined in Java.

The predicates `feature` and `feature1` allow to ask if a class or interface has a certain feature, i.e. a variable, a method or a constructor. The predicate `feature1` determines if the feature is directly declared in the class, while feature takes inheritance into account. The following query determines the presence of a feature in a class or interface, either by explicit declaration or inheritance:

```
:-feature(?type,?feature).
```

The features are represented as TyRuBa terms. A variable named `?n` of type `?t` is represented by the term

```
var<?T,?n>
```

A method `?m` with return type `?R` and argument types `?A1,...,?An` is represented as

```
method<?R,?m,[?A1,...,?An]>
```

A constructor with arguments `?A1,...,?An` is reprinted as

```
constructor<[?A1,...,?An]>
```

3.3 Summary

In this chapter we described the technique of logic meta programming as approached in [14] and the logic meta programming system TyRuBa. A base level program is represented by a set of logic facts as determined by a representational mapping. In a logic meta programming system the set of facts that represent the base program is indirectly represent by a logic meta program and inferred using an inference engine. A code generator queries the logic system to generate the base level programs. TyRuBa is a logic meta programming system with special features for dealing with Java code. We also presented a representational mapping and its code generator. The TyRuBa system and this representational mapping will be used for representing the applications assembled from components and to generate the application code.

Chapter 4

Representing component models for generic builder tools.

The goal of this dissertation is to make the knowledge about a component model independent from the builder tool. We propose to use a logic programming language as a uniform way of describing the components, the mechanism for assembling them and the composite applications. The idea is that components and components models are represented as sets of facts and rules that are stored in a rule base which is consulted by the builder tool to help the user to produce a description of the composite application. The application is also represented by a set of facts, which are added into the rule base. The application code is indirectly represented by the rules and facts that describe the composite application, the components and the component model. The code is generating by querying this set of facts, following the logic meta programming approach presented in chapter 3.

The interface between the builder tool and the component model is well defined in terms of a set of queries the tool asks the rule base and a set of facts the tool adds to the rule base. Thus, the tool is generic as it can be used with any model whose representation conforms to that interface.

In this chapter we explain how a generic builder tool interacts with a logic representation of a component model and present a prototype tool based on this interaction. We discussed the representation of components, composite applications and component models in terms of facts and rules that support the operation of the tool and the generation of the application code. We illustrate the representation by representing Java Beans.

4.1 General Picture

This section discusses the interaction between the logic representation of a component model and a generic builder tool.

In section 2.4.1 we described how a composite application is created using a builder tool. The tool guides the user to perform the different steps to build the composite application:

1. create a container for parts
2. deploy a set of components as parts of the application
3. configure the parts
4. connect the parts

The creation of the connection involves several sub steps:

1. select the source of the link
2. select the protocol of connection
3. select a plug among the plugs in the source part
4. select the target part
5. select a plug among the plugs in the target part

The tool guides the user when performing each of the steps so that only correct assemblies are built (correct in the sense that the composition is valid). It does so by placing constrains on what the user can build. For example, when creating the parts, it allows to select only components that are valid for the current container. When creating a connection, it allows to choose source and target plugs that are compatible. In order to help the user in this way, the tool needs to have knowledge about the component model and the components that allows it to determine what is valid. In a tool like the BeanBox this knowledge is hard-coded. This knowledge is not explicit, and we can not modify it unless we modify the implementation of the tool. It is not possible to add modify a protocol, modify the way the code is generated, or add new protocols.

We propose an architecture for a generic builder tool where the knowledge about the component models is represented separately from the tool. (See figure 4.1)The component model is described by a set of facts and rules stored in a rule base. The components are also described by logic facts in the rule base. The user interacts with the tool to create the description of the composite application. The tool queries the rule base to find out what is valid and help the user to create a correct assembly. The description of the composite application is also represented by a set of logic facts. These facts specify which is the container, which are the parts and how the parts are connected. The tool inserts these facts into the rule base.

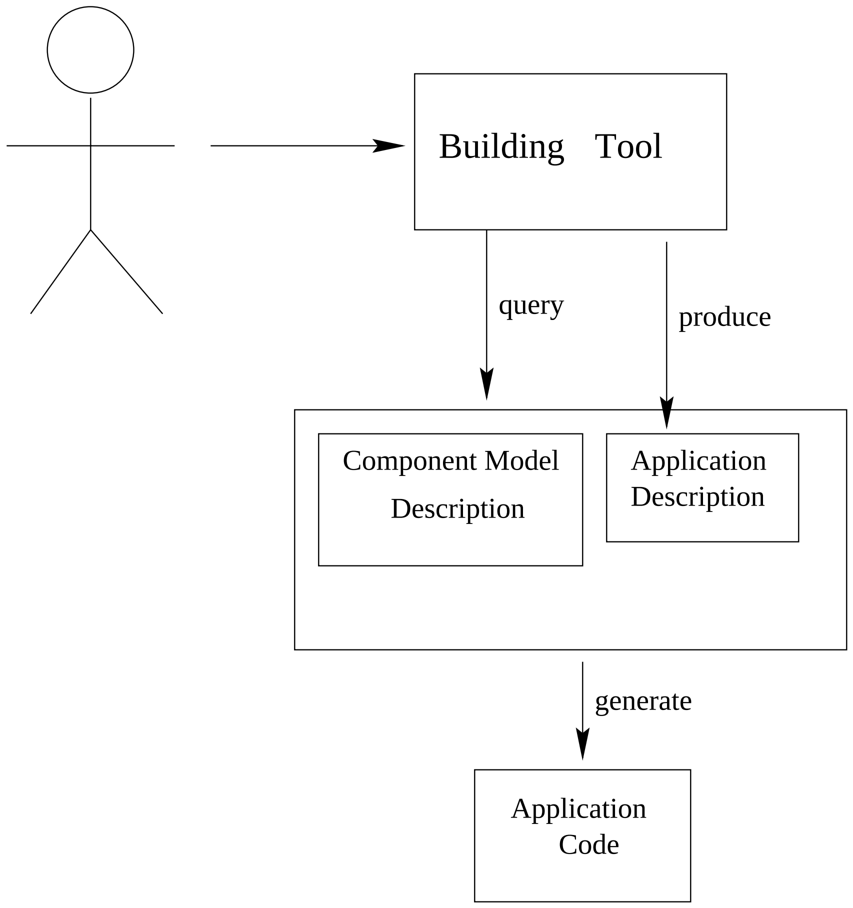


Figure 4.1: Architecture of a generic composition tool

The facts describing a composite application are:

`composite(?containertype,?compositename).` This fact declares a composite application named `?compositename` using the container `?containertype`.

`part(?compositename,?component,?partname).` This fact links `?component` as part of the composite application `?compositename`.

`configure(?compositename,?partname,?propertyname,?value).` This fact configures the part `?partname` by providing a the value `?value` for `?propertyname`.

`connect(?protocol,?compositename,?part1,?plug1,?part2,?plug2). -` This fact specifies the linking of `?plug1` in `?part1` to `?plug2` in `?part2` using the protocol `?protocol` within `?compositename`.

The tool guides the user to make valid application descriptions. For this purpose, it queries the rule base to determine whether a certain declaration is valid. For each of the possible declarations described above, there is a corresponding query the tool asks the rule base to constrain the description that can be made to the valid ones:

`:-validcontainer(?containertype,?compositename).` This query determines which are the containers we can use to create a composite application in the component model we are using.

`:-validpart(?compositename,?component,?partname).` This query determines which components can be used as parts of the application

`:-validconfiguration(?compositename,?partname,`

`?propertyname,?value).` This query determines which are the possible configuration for the parts of the application.

`:-validconnection(?protocol,?compositename,`

`?part1,?plug1,?part2,?plug2).` This query determines which plugs can be connected and under which protocol.

Finally, the tool queries the rule base to generate the code for the composite application. The code of the application is represented by a set of facts using the representational mapping presented in section 3.2.3. This facts are deduced from the facts that describe the composite application and the facts and rules that describe the component model and the components. The following query obtains the code for the application:

```
:-generate(?composite,?code).
```

A prototype builder tool

To validate our approach we implemented a prototype builder tool. We use this tool to build applications in different component models whose representation is explained later on this section. This tool validates the approach as it shows that the representation of component models we propose can effectively be used by a builder tool to construct applications and that we can use the same tool with different component models. Being a proof-of-concept implementation, the tool has the minimal functionality needed for the purpose of validation and provides only a very simple user interface.

The tool allows to build a composite application by performing each of the necessary steps: create a container, add parts to it, connect them and generate the code for application. It queries the rule base containing the knowledge about the component model using the interface described in the previous section to allow creating only valid compositions.

Figure 4.2 shows the interface to create a part. The component pop-up menu allows to choose between the components that can be used to create valid parts for the current container. Its contents are obtained by executing the query

```
:-validpart(thiscomposite,?component,?part)
```

and taking the binding of the variable component from all the solutions.

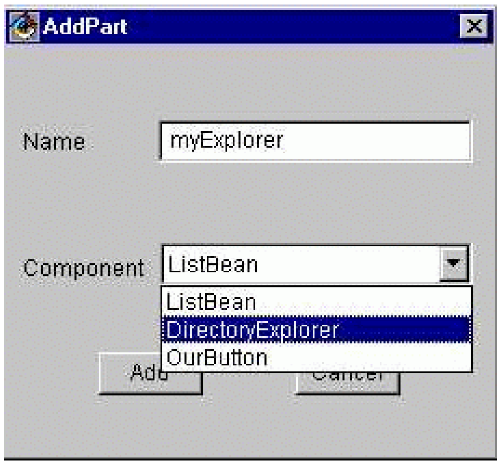


Figure 4.2: Interface for adding a part to a composite application

When the user presses the Add button, the tool inserts the following fact into the rule base:

```
part(thiscomposite, chosencomponent,partname).
```

Figure 4.3 shows the interface to create a connection. It allows to create a connection that is valid in the current application by choosing a protocol, a plug in one part and a plug in another part. The pop-up menu corresponding to each element shows which are the valid choices according to what is already chosen for the other elements. For example, if the source and target parts have been chosen, then the protocol can be selected among the protocols for which those parts have compatible plugs.

The possible values for an element are obtained by querying the rule base for the valid connections, using a variable as the argument for the element we are interested in, an the chosen values of the rest of the elements.

For example, the choices for the source plug are obtained by executing the query

```
:-validconnection(chosenProtocol, thisComposite,
                  chosenPart1, ?plug1,
                  chosenPart2, chosenPlug2).
```

and taking the values for the variable `?plug1` from all the solutions. The value of `thisComposite` is the name of the application under construction. The other arguments to the query are the chosen value for the element, or a variable if there is no selected value for the element.

The pop-up menu for an element shows the valid values that are obtained from the query and extra option allowing to leave the value unchosen.

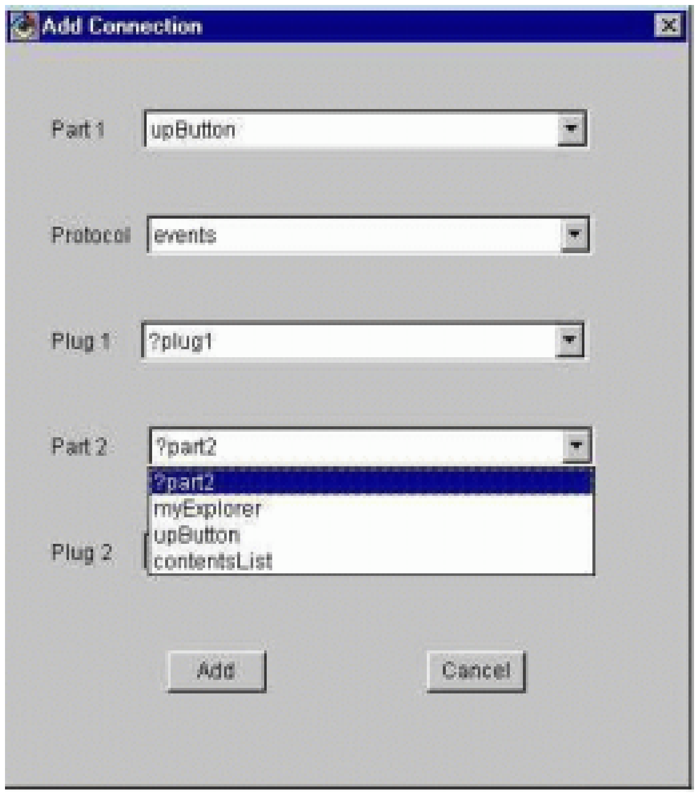
When the user presses the Add button, the following fact is inserted into the rule base:

```
connect(chosenProtocol, thisComposite,
        chosenPart1, chosenPlug1,
        chosenPart2, chosenPlug2).
```

The fact is only inserted when all the elements in the connection have been chosen. It is forbidden to add a connection where one of the elements is unchosen (i.e. one of the elements is a variable).

Configuration is done in a similar way. Figure shows the interface for configuring a part. The query that is used to provide the choices for the part and the property is

```
:-validconfiguration(thisComposite,?part,
                     ?property,?value).
```


Add Connection

Part 1 upButton

Protocol events

Plug 1 ?plug1

Part 2 ?part2

?part2
myExplorer
upButton
contentsList

Plug 2

Add Cancel

Figure 4.3: Interface for creating a connection between two parts

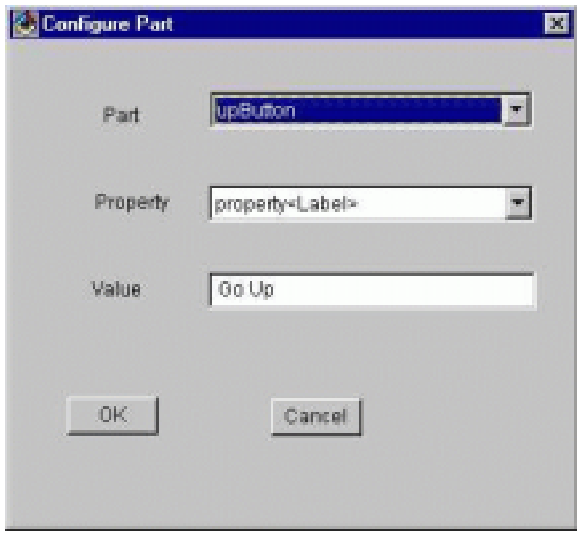


Figure 4.4: Interface for configuring the parts of the application

When the configuration is accepted by the user, pressing the OK button, the following fact is added to the rule base:

```
configure(thisComposite,chosenPart,
          chosenProperty,value).
```

In this section we discussed how a generic builder tool interacts with a logic representation of the component model to guide the user in assembling an application. The interaction consists in a set of queries that tool asks the rule base to find out what are valid compositions, a set of facts describing the composite application that the tool inserts into the rule base and a query that generates the application code. In the rest of the chapter we will present the representation of components, component models and applications that make this possible.

4.2 Describing an application

The description of an application specifies a container in which the parts are put, the parts and the connections between the parts. This description is a kind of script consisting of a set of logic facts. This description can be created by a builder tool, such as the one described in the previous section, but it can also be written down directly using the logic language as a kind of scripting language.

The description of a composite application contains exactly one fact that specifies the container to be used to host the parts of the composite application, a number of facts that specify which components are used as parts of the composite application and a number of facts that specify the links between the parts, describing which are the plugs that are connected and which is the protocol used to connect them.

In the rest of this section we will illustrate the description of an application by the representation of the directory browser application introduced in section 2.4.1. (Figure 4.5)

The application is built of four parts: a text label, a list box , a button and a non-visual component, directory explorer that explores the directory hierarchy. The text label will show the path of the current directory. The list box shows the contents of the current directory an allows to chose a directory to explore. The button allows to go up in the directory hierarchy. The directory explorer maintains the current directory and knows it contents, and can go up and down in the hierarchy.

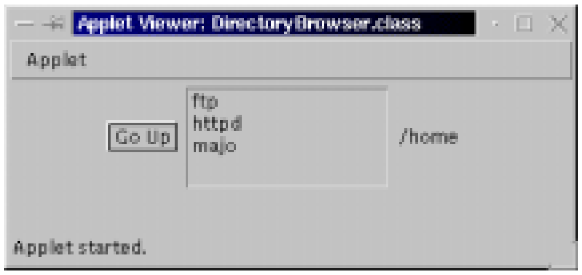


Figure 4.5: A composite application: Directory Browser

The composite application is built using a container for beans. This is expressed by the fact:

```
composite(beancontainer,DirectoryBrowser).
```

The following facts assert which are the parts of the application: a text label, a list box, a button and a directory explorer.

```
part(DirectoryBrowser,Label,path).
part(DirectoryBrowser,ListBean,contentsList).
part(DirectoryBrowser,Button,upButton).
part(DirectoryBrowser,DirectoryExplorer,myExplorer).
```

The upButton is configure by providing a text, and we also configure the directory explorer so it stars browsing the home directory:

```
configure(DirectoryBrowser,upButton,
         Text,”Go Up”).
configure(DirectoryBrowser,myExplorer
          currentDirectory,”/home”).
```

Next we need to represent the connections between these parts. The connection between the `currentDirectory` property in the directory explorer and the `text` property of the label, so that when the `currentDirectory` changes, the change is reflected by label showing the path of the new current directory is described by the following fact:

```
connect(synchronize, DirectoryBrowser,
        myExplorer, property<currentDirectory>,
        path, property<text>).
```

Synchronize is the name of the protocol that links to properties so that the change of the value in the first is reflected in the same change in the value of the second.

The `contents` property in the directory explorer is linked to the items property of the list box, so that the contents of the list box are updated when the contents of the directory explorer change to those of a different directory:

```
connect(synchronize, DirectoryBrowser,
        myExplorer, property<contents>,
        contentsList, property<items>).
```

The list box has a bound property, selectedItem, that is the current selection. This property is linked to the currentDirectory property of the directory explorer, so that when a directory is selected in the list box, the directory explorer goes to explore the selected directory.

```
connect(synchronize, DirectoryBrowser,
        contentsList, property<selectedItem>,
        myExplorer, property<currentDirectory>).
```

The directory explorer has a method goUp, that makes it go to explore the upper directory. The event action performed of the button is linked to this method, so that when the button is pressed, the goUp message is sent to the directory explorer. This connection is specified by the fact

```
connect(events, DirectoryBrowser,
        upButton, event<Action<Performed>>,
        myExplorer, method<goUp,[]>).
```

`events` is th protocol that links an event to a method, in such a way that the method is invoked when the event occurs. In the above description, the method is identified by its name and the list of the types of the arguments, which in this case is empty.

The logic facts presented in this section describe the composite application. In the next chapter we will show how the representation of the component model allows to generate the code for the application from this description.

4.3 Describing a Component Model

In this section we present how to represent a component model and illustrate it by discussing the representation of Java Beans.

The representation of the component model must support

- guiding the user in the construction of application description
- the generation of the code for the application from this description.

This representation provide rules and facts to determine what is a valid container and how to create a composite application, what are valid parts and how to link them to the container, and what are valid connections and how to create them. To determine what are the valid elements, the description must contain facts and rules to answer the queries `validcontainer`, `validpart`,`validconfiguration` and `validconnection`. To create the composite application it must define rules that allow to compute the description of the application code from the facts that describe the application: `composite`, `part`, `configure`, `connect`. The description of the application code will be a meta-level representation of the base program in terms of logic facts, as described in chapter 3.

In addition to the description of the component model itself, we need descriptions of the components that provide how to instantiate the component, how to link it to a container, how to configure and how to connect it.

The set of rules and facts that represent a component model are organized in different groups that describe the different elements of the component model: the container, the protocols for adding parts to a container and the protocols for connecting the parts. Each of this descriptions is responsible for part of the obligations of the component model:

- The description of the container is responsible for providing a rule for

  ```
  validcontainer(?container, ?compositename),
  ```

  and to generate the code corresponding to the declaration

  ```
  composite(?container, ?compositename).
  ```

- The description of the creation of parts is responsible for providing a rule for

  ```
  validpart(?compositename,?componentname,?partname)
  ```

  and to generate the code corresponding to the declaration

  ```
  part(?compositename,?componentname,?partname).
  ```

- The description of the configuration of parts is responsible for providing a rule for

  ```
  validconfiguration(?compositename,?partname,
                     ?propname,?value)
  ```

  and to generate the code corresponding to the declaration

  ```
  configure(?compositename,?partname,
            ?propname,?value)
  ```

- The description of a connection protocol is responsible for providing a rule for

  ```
  validconnection(protocolname,?compositename,
                  ?part1,?plug1,
                  ?part2,?plug2)
  ```

  and to generate the code to realize the connection specified by

  ```
  connect(protocolname,?compositename,
          ?part1,?plug1,
          ?part2,?plug2).
  ```

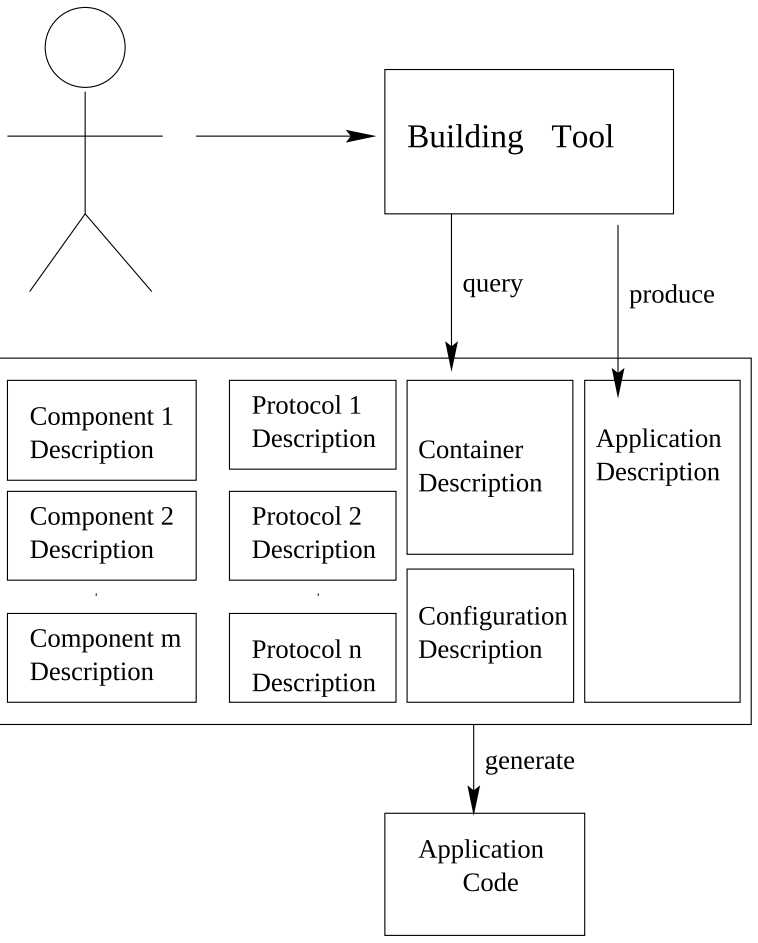


Figure 4.6: Architecture of the component model description

In accordance to the above discussion, the architecture of the generic composition tool from figure 4.1 can be detailed to show the organization of the rule base into the description of the different elements in the component model, as shown in Figure 4.6. The rest of this section discusses the description of each of the elements.

4.3.1 Describing components

The description of a component must specify how the component is to be used: how to integrate it as part of an application, how it must be configured and how it must be connected to other components. Usually, a component model will standardize these mechanisms for all the components that conform to the component model. In such a case, the component model provides support for the standard mechanisms and the components provide hooks for the operation of the mechanisms. Therefore, the description of how to use the component is indirect and relies on the description of the component model, namely in the description of the creation of parts and the connection protocols.

In Java Beans, for example, the mechanism for deploying a bean in an application is the same for all beans, and is indirectly described by knowing that the component is a bean and fromthe protocol for adding a part into an application. Similarly, the bean's properties are indirectly described by getter and setter methods and the events are indirectly described by listener registration/unregistration methods.

The description of a bean consists of a fact stating that the component is a bean, and a set of facts describing its public methods.

The way of deploying beans is standard for all beans, and described is section 4.3.3.

The descriptions of the connection protocols allow to deduce the bean's properties and events that are indirectly represented by some of the beans public methods(we discuss how in section 4.3.5).

The OurButton bean has a read/write property Background. This property is indirectly described by the following facts stating the presence of a getter and a setter method:

```
feature1(OurButton,
         method<java.awt.Color,get<Background>,[]>).
feature1(OurButton,
         method<void,set<Background>,[java.awt.Color]>).
```

A bound property is explicitly described as such by a fact, as it is not possible to deduce that it is bound from the methods. Background is a bound property:

```
bound(OurButton, Background).
```

As OurButton has a bound property, it has methods for registering and unregistering property change listeners:

```
feature1(OurButton,
         method<void,add<PropertyChange<Listener>>,
                [PropertyChange<Listener>]>).
feature1(OurButton,
         method<void,remove<PropertyChange<Listener>>,
                [PropertyChange<Listener>]>).
```

The following methods describe event listener registration and unregistration methods, and they indirectly describe that the bean emits an action event:

```
feature1(OurButton,
         method<void, add<Action<Listener>>,
         [Action<Listener>]>).
feature1(OurButton,
         method<void, remove<Action<Listener>>,
         [Action<Listener>]>).
```

For the purpose of our experiments, we simply wrote down the facts describing the beans' methods but we could also obtain them using Java introspection. We could write a program that inspects a bean and generates the set of facts that describe it. We only considered the plugs that can be deduced using the programming conventions defined in Java Beans and not the ones described using the BeanInfo. The BeanInfo is a way of providing explicitly information about the plugs, instead of letting them to be deduced from the programming conventions. To extend our representation to handle BeanInfo, we just have to map the description of the plugs in the bean info into facts that asses the existence of the plugs, and these plugs would coexist with the plugs deduced using the rules.

4.3.2 Describing a container

The description of a container must define how to create a composite application, what components it can host, and how to add parts into it. Additionally, it may provide support for the connections between the parts, such as collecting and hosting initialization code for the connections. The description of the container uses the description of the components, the protocol to add parts and the protocols for connecting components.

In the case of beans, the description of the container specifies the creation of a class that implements the composite application. This class contains pieces of code handling the creation of the parts and the initialization of the connections. These pieces of code are described by the part and connection protocols. The container is responsible for collecting and hosting this pieces of code.

Let's consider an example of a container for beans. It creates an applet class for the composite application. The following facts are part of the description of the container:

```
class_M(?comp) :- composite(beancontainer,?comp).
public_M(?comp):- composite(beancontainer,?comp).
extends_M(?comp, MyAbstractApplet):-
    composite(beancontainer,?comp).
```

`MyAbstractApplet` is a support class we defined that provides basic functionality for all the generated applets.

The generated applet has a method that initializes all the parts. This method host the code for initializing each part. This code is generated by the description of the creation of parts (section 4.3.3) from the declaration of the parts in the composite application.

```
method_M(?comp, void, initContents, [], {
   public void initContents() throws
      java.lang.ClassNotFoundException, java.io.IOException {
      ClassLoader myLoader =
         this.getClass().getClassLoader();
      ?initcode
      addConnections();
      ?configcode
   }}):-
  composite(beancontainer,?comp),
  FINDALL(initialization(?comp,?part,?impl),?impl,?initcode),
  FINDALL(configuration(?comp,?part,?impl),?impl,?configcode).
```

The first statement in the generated method obtains the class loader, which is used in the instantiation of the beans. The variable `?initcode` will be bound to the initialization code of all the parts, which are collected by the `FINDALL` predicate in the condition of the rule. The variable `?configcode` will be bound to the code that configures the parts, and is collected in the same way as the initialization code. The statement `addConnections();` method invokes the method that initializes the connections between the beans. The method for initializing the connections is generated by the following rule:

```
method_M(?comp, void, addConnections, [], {
   public void addConnections() {
      ?code;}
   }):-
   composite(beancontainer,?comp),
   FINDALL(connection(?comp, ?impl),?impl,?code).
```

The above method hosts the code for initializing the connections. This code is generated by the connection protocols from the declaration of the connections (section 4.3.5)

4.3.3 Describing parts

A part is a particular use of a component in a composite application. The description of a part specifies the component and the container that hosts it.

In Java Beans, creating a part implies defining a variable in the container and code for initializing this variable.

```
var_M(?comp,?bean,?part,{}):-
    part(?comp,?bean,?part).

initialization(?comp,?part,{
    ?part=(?bean)Beans.instantiate(myLoader,
                                   "?bean");
    }):-
    part(?comp,?bean,?part).
```

The initialization code, that creates an instance of the bean, will be collected and hosted by the container, as described in the previous section.

4.3.4 Describing configuration

Configuration is achieved by setting some parameters in the components. Beans are configured by providing values for its writable properties. A valid configuration for a part created from a bean provides a value for a property for which a setter method is defined. This is expressed by the following rule:

```
validconfiguration(?comp,?part,?prop,?value):-
        part(?comp,?bean,?part),
        setter(?bean,?prop,?meth).
```

To realize the configuration of the part, some initialization code is generated. It invokes the setter method with the provided value:

```
configuration(?comp,?part, {
        ?part.?setterName(?value);}):-
        part(?comp,?bean,?part),
        configure(?comp,?part,?prop,?value),
        setter(?composite,?part,?prop,
               method<?setterType,?setterName,[?propType]>).
```

4.3.5 Describing connection protocols

The representation of a protocol describes which are the valid connections between the parts and how to build them. A valid connection links a plug in one part of the composite application to a compatible plug in another part of the application. To determine whether a connection is valid we need to determine the presence of valid plugs in the parts and that those plugs are compatible. The plugs are described by the components from which the parts are created. A plug can be described directly or indirectly. An indirect description means that the protocol deduces the plugs from some information in the description of the component.

We will discuss the representation of the two protocols corresponding to the connections provided by the BeanBox, described in section 2.4.3: synchronize and events. The synchronize protocol links a property to another so that a change in the value of the first one will be propagated to a change in the value of the second one. The events protocol links an event to a method so that the occurrence of the event produces the invocation of the method.

Synchronize

The synchronize protocol links two properties: a source and a target property. The changes of the value in the source property will be propagated to the second property. The presence of a property plug is indirectly described by some public methods in the bean.

A valid connection links a valid source property to a valid target property whose types are compatible (the type of the source is a subtype of the target):

```
validconnection(synchronize,?composite,
                ?source,?sprop,
                ?target,?tprop):-
    validSource(synchronize,?composite,?source,?sprop),
    validTarget(synchronize,?composite,?target,?tprop),
    compatible(?composite,?source,?sprop,?target,?tprop).
```

To determine that a part has a valid source or valid target plug, we consider the component from which it has been created.

To be a valid source of a synchronize connection, a property must be readable: it must have a getter method. It must be bound: it will raise an event PropertyChangeEvent when its value changes, and it must have methods to register and unregister listeners of the PropertyChangeEvent. The following rule states what is a valid source:

```
validSource(synchronize,?comp, property<?name>):-
   getter(?comp,property<?name>,
          method<?proptype,?getname,[]>),
   bound(?comp,?name),
   feature(?comp,
           method<void,add<PropertyChange<Listener>>,
                 [PropertyChange<Listener>]>),
   feature(?comp,
           method<void,remove<PropertyChange<Listener>>,
                 [PropertyChange<Listener>]>).
```

The conventions to detect a bound property are specified in the Java - Beans Specification [5]. These conventions allow to determine that a bean has a bound property form the presence of methods to register and unregister PropertyChaneListeners, but it is not possible to determine which is the bound property. For this reason, we explicitly state which are the bound properties in the description of the bean. Therefore, the condition `bound(?comp,?name)` will be directly solved by the fact in the description of the bean. So will the conditions about the presence of registration and unregistration methods. The presence of the getter method will be deduced using the following rule:

```
getter(?comp,property<?name>,
          method<?proptype,get<?name>,[]>):-
        feature(?comp,method<?proptype,get<?name>,[]>).
```

A valid target of a synchronize connection is a writable property, that is, a property for which a setter method is defined:

```
validTarget(synchronize,?comp,property<?name>):-
   setter(?comp, property<?name>,
          method<void, ?setname, [?proptype]>).

setter(?comp, property<?name>,
       method<void,set<?name>,[?proptype]>):-
   feature(?comp,method<void,set<?name>,[?proptype]>).
```

Note that in order to determine that a plug in a part is a valid source or target for a connection we considered the corresponding components. This properties will be transfered or inherited into the parts created from the component using the following rule

```
?property(?protocol,?comp,?part,?prop) :-
   inheritfromcomponent(?property),
   part(?comp,?type,?part),
   ?property(?protocol,?type,?prop).
```

We state that `validSource` and `validTarget` will be transfered to the parts by the facts

```
inheritfromcomponent(validSource).
inheritfromcomponent(validTarget).
```

Building the connection. To realize a connection through the synchronize protocol, we generate a class for an object that mediates the synchronization and some code for initializing the connection that creates this mediating object and registers it as a listener to the source object.

The class implements the PropertyChangeListener interface. The method that is called for notifying a PropertyChange event sets the value of the target property to the value of the source property when the source property changes. As the same event is emitted for the change of all the properties, this method has to determine whether the property that was modified is the one that is the source of the connection.

```
method_M(synchronize<?composite,?source,
         property<?sname>,?target,property<?tname>>,
         void,propertyChange,[PropertyChanged<Event>], {
   public void propertyChange (PropertyChange<Event> e){
     String propertyName = e.getPropertyName();
     if (propertyName == "?dsname"){
        target.?setterName((?propType)e.getNewValue());}
   }
}):-
    connect(synchronize,?composite,
            ?source,?property<?sname>,
            ?target,?property<?tname>),
    setter(?composite,?target,property<?tname>,
           method<?setterType,?setterName,[?propType]>),
    CAP(?dsname,?sname).
```

The `CAP` predicate associates an identifier starting with a down-case to the same identifier, but where the first letter is the corresponding uppercase. It is necessary to appropriately handling the naming conventions.

The code to initialize the connection creates an object for mediating the synchronization and register it as a listener of property changes in the source

```
initializeconnection(?composite,
   {?source.add<PropertyChange<Listener>>( new
       synchronize<?composite,?source,
                   ?sprop,?target,?tprop>(?target));
   }):-
   connect(synchronize,?composite,
           ?source,?sprop,
           ?target,?tprop).
```

`synchronize<?composite,?source,?sprop,?target,?tprop>` is the class of the mediating object, named after the connection for which it is generated. Its constructor has a parameter that is the object whose property is to be updated.

The code for initializing the connections will be collected by the container and inserted as initialization code in the composite application, as explained in section 4.3.2.

Events

The events protocol links and event to a method, in such a way that the method is invoked when the event is raised. A valid connection using the events protocol links a valid event plug in a part to a valid event target method in another part. This is expressed by the following rule:

```
validconnection(events, ?composite,
                ?source, event<?ev, ?action>,
                ?target, method<?name, ?args>):-
   validEventSource(events, ?composite,
                    ?source,event<?ev, ?action>),
   validEventTarget(events, ?composite,
                    ?target,method<?name, ?args>),
   compatible(event<?name>, ?args).
```

As in the synchronize protocol, the plugs are described by the components. An event source plug is described indirectly by the programming conventions: the bean must have methods for registering and unregistering event listeners, there is a class of event objects and the listener type must contain a method for notifying the occurrence of the event. The following rule uses this conventions to deduce what is a valid event source:

```
validEventSource(events, ?component, event<?name,?action>):-
   registrationmethod(?component,event<?name,?action>,
                      ?method<?R, ?regnam, [?ltype]>),
   unregistrationmethod(?component,event<?name,?action>,
                        method<?R,?unregnam, [?ltype]>),
   class(?name<Event>),
   interface(?ltype),
   subtype(?ltype, Listener),
   CAP(?downname,?name),
   feature(?ltype,
          method<void,?downname<?action>,[?name<Event>]>).
```

A method is a valid target of an event connection if it has no parameters or a single parameter that is an event object:

```
validEventTarget(events, ?component, method<?n, ?args>):-
```

```
        type(?component, ?ctype),
        feature(?ctype, method<?R, ?n, ?args>),
        validArgs(?args).
    validArgs([]).
    validArgs([?t]):-subtype(?t, Event).
```

To determine the validity of the connection, in addition to determining that the plugs are valid, it is necessary to determine that they are compatible. In this case, we need to determine that the arguments of a target method are compatible with the event. It is so if they are empty or a sole argument that is a subtype of the event.

```
    compatible(event<?name>,[]).
    compatible(event<?name>,[?t]):-subtype(?t, ?name<Event>).
```

Building the connection To build the connection we generate an adaptor class and code for initializing the connection, much in the same way as we did for the synchronize protocol. The adaptor class implements the listener interface of the event we are connecting to. The method that is invoked for the notifying the occurrence of the event, invokes the target message. The initialization code creates an adaptor object and registers it as listener of the event in the source.

4.4 Summary

This chapter showed how the knowledge used by a builder tool to create applications out of components can be separated from the builder tool and represented using a logic programming language.

We described how a builder tool interacts with a logic representation of a component model to help the user to create an application out of components. The logic representation of the component model consists in a set of facts and rules store in a rule base. The tool guides the user in creating the application by allowing to build only valid component assemblies, and generates the code for the application.

The interface between the tool and the logic representation consists of a set of queries the tool executes to determine what is a valid assembly, a set of facts describing the composite application created by the user that are added into the rule base, and a query that generates the application code.

We described a proof-of-concept implementation of a builder tool. This builder tool will be used to validate the approach by using it to assemble applications in different component models.

We discussed how to represent component models, components and applications as facts an rules to support the operation of the tool. The representation of a component model is composed by several interacting parts describing the different elements in the component model: the container, the parts and the protocols. We illustrated these representations by representing Java Beans and an application built by assembling beans.

Using the representation of Java Beans and the prototype tool it is possible to build applications by composing beans. This shows that the proposed representation effectively supports the operation of a builder tool. In the next chapter we will discuss some experiments to validate that the tool is generic and can be used with extended versions of the component model and with a different component model.

Chapter 5

Independence between the component model and the builder tool

This chapter discusses some experiments that show that the tool and the component model are independent and we can change the component model and continue to use the same tool.

The first experiment shows that the component model can evolve while continuing to use the same tool. We do this by extending the Java Beans component model with new protocols.

The second experiment shows that the tool can be used with a different component model. To do this we represent a new component model.

One of the protocols we add to Java Beans is a protocol for connecting a required service (operation in another component) to a provided service. The other is a two-way synchronization of properties: the change in any of them updates the other one. The two-way synchronization protocol is built upon the basic synchronize protocol described in the previous section.

The other model of components we represent is based on a pedagogical example about simulation of water conduits. This is not a component model in the sense of Java Beans of the other commercial models, but it is a kind of framework which offers a number of reusable software pieces to build applications with, and in this sense is very similar to a component model. We identify in this framework the main elements that (according to our definition) make up a component model and represent them as facts and rules, conforming to the interface described in the previous chapter.

In both cases, the extension of Java Beans and the new model, the prototype tool described in the previous chapter can be used without any change, which show that it is generic and independent from the component model.

5.1 Extending the protocols for beans

5.1.1 Outcall

Using the regular connection mechanisms in Java Beans it is not possible for a bean to specify that it requires a service from other bean. It imposes a push communication model, where information is pushed as it comes up, rather than a pull model where information is pulled as needed [6]. It is not possible to make a call and get a return value when needed. The reason for this is providing decoupling, allowing for late connection of components. The outcall protocol presented in this section adds the feature of a pull communication model, while keeping decoupling and late connection.

The outcall protocol describes the connection of a required service to a provided service. A required service means that the bean invokes an operation. A provided service is an implementation of the operation. It is not necessary that the required and provided service have the same name.

A connection of a required to a provided service is described as

```
connect(outcall,?compositename,
        ?client,outcall<?oname,?oargs>,
        ?provider,method<?pname,?pargs>).
```

Following the style of Java Beans, a required serviced is described using programming conventions. A provided service is just any public method.

The required service is described by an interface named servicetypeService that contains the method to be invoked. This interface extends the interface Service. Service is an empty interface we define for the purpose of identifying service descriptions, much in the same way listener interface is identified because it extends the empty interface Listener.

A bean that requires that service will provide a method `set`*`servicetype`*`Service`, this method allows to deduce that the bean requires the service and also to register the provider of the service when building the connection.

The convention defined for outcalls allows to code a bean that require a certain service in plain Java: it just refers to the required service through a variable whose type is the service interface. The setter method for the service must bound this variable to the provider of the service.

The following rule determines whether a bean has an outcall plug:

```
validClient(?component,outcall<?oname,?oargs>):-
   servicesetmethod(?component,outcall<?oname,?oargs>,
                    ?meth).

servicesetmethod(?component,outcall<?oname,?oargs>,
                 method<void,
                        set<<?oname,?oargs>,<Service>>,
                       [?servicetype<Service>]>):-
   feature(?component,
           method<void,set<<?oname,?oargs>,<Service>>,
                 [?servicetype<Service>]>),
   interface(?servicetype<Service>),
   implements(?servicetype<Service>),
   feature(?servicetype<Service>,
           method<?R,?oname,?oargs>).
```

A valid connection links a required service, described by the above conventions, to a provided method which has a compatible signature:

```
validconnection(outcall,?composite,
                ?client,?o,
                ?provider,?m):-
    validClient(?composite,?client,?o),
    compatibleoutcall(?composite,?client,?o,?provider,?m),
    validProvider(?composite,?provider,?m).

validProvider(?component,method<?name,?args>):-
    feature(?component,method<?T,?name,?args>).
```

The signature of a method is compatible with an outcall if it is correct according with Java's type system to invoke the provided method instead of the required one. This is so if the return type of the provided method is a subtype of the required one, and each of the arguments of the required method is a subtype of the corresponding argument of the provided service.

To build the connection the protocol generates an adaptor class that implements the service interface. The implementation of the method corresponding to the required service invokes the provided service method on the service provider. The protocol also generates code for initializing the connection: it creates an adaptor object and registers it to the client of the service:

```
connection(?comp,
   {?client.?setname(new
    Adaptor<?comp,?client,?oname,?provider,?pname>(?provider));}):-
   connect(outcall,?comp,
           ?client,outcall<?oname,?oargs>,
           ?provider,method<?pname,?pargs>),
   servicesetmethod(?comp,?client,
                    outcall<?oname,?oargs>,
                    method<?t,?setname,?setargs>).
```

The code for initializing the connection will be collected by the container in the same way as it happens for the events and synchronize protocols, described in section 4.3.2.

5.1.2 Combined Protocols: two-way synchronize

The two-way synchronize protocol links two properties making that a change of the value in any of them is reflected in the other. The two-way synchronize protocol is a combined protocol: a connection of two properties using the two-way synchronize protocol is equivalent to two connections using the synchronize protocol with the roles of the source and the target exchanged. A connection using two-way synchronize is valid if both connections using synchronize are valid. The declaration of a connection using two-way synchronize generates the declaration of the two corresponding synchronize connections.

The combination of protocols can be generalized, so that we describe how the combined protocol combines other protocols and this allows to deduce what are the valid connections and how to build them on top of other, more basic connections.

This manipulation of protocols shows the expressive power provided by logic meta programming for the description of component models. The descriptions of the connections are facts, so we can write rules that define them indirectly. The valid connections are described by rules, and so we can write rules that allow to deduce that a connection is valid from the validity of other connections.

The description of the combined protocol specifies the list of more basic protocols that it combines, and the correspondence between the plugs in the combined protocol and the plugs in the basic protocols. The description of the two-way synchronize as a combined protocol is as follows:

```
combinedprotocol(
      twowaysynchronize<?comp,?part1,?prop1,?part2,?prop2>,
      [synchronize<?comp,?part1,?prop1,?part2,?prop2>,
       synchronize<?comp,?part2,?prop2,?part1,?prop1>]).
```

A connection using a combined protocol implies a connection for each of the subprotocols that constitute the combined protocol:

```
connect(?subprot,?comp,?p1,?pl1,?p2,?pl2) :-
  connect(?cp,?comp,?cp1,?cpl1,?cp2,?cpl2),
  combinedprotocol(<?cp,?comp,?cp1,?cpl1,?cp2,?cpl2>,?subprots),
  element(<?subprot,?comp,?p1,?pl1,?p2,?pl2>,?subprots).
```

A connection using a combined protocol is valid if each of the connections implied by the combined protocol is valid:

```
validconnection(?prot,?composite,
                ?part1,?plug1,
                ?part2,?plug2):-
  combinedprotocol(?prot<?composite,
                         ?part1,?plug1,
                         ?part2,?plug2>,
                   ?protlist),
  andlist<validconnection>(?protlist).
```

`andlist<?predicate>(?list)` is true if `?predicate` is true for all the elements in `?list`.

5.1.3 Summary

In this section we extended Java Beans with new connection protocols. The prototype builder tool can be used to build applications in this extended version of the model thus showing that it allows the evolution of the component model.

5.2 Conduits component model

Conduits is a mini framework for building water flow simulations developed by prof. Theo D'Hondt (Vrije Universiteit Brussel, PROG laboratory) that is used for pedagogical purposes in several courses. It consists of a set of classes representing different kinds of water conduits, such as water sources, pipes and joins. A simulation is build by assembling a set of instances of this classes. Although this framework is not a “real” component model like Java Beans,it provides a set of reusable software pieces that can be put together to build applications. In this sense it is similar to a component model. As we will explain in the rest of the chapter, it is possible to identify in this framework the features we identified as fundamental in a component model. We will represent it following the structure for representing component models introduced in chapter 4 and use the prototype tool to build applications in this framework.

The classes in this framework form a hierarchy shown in figure 5.1. A composite application puts together a set of instances of the concrete classes in the hierarchy. Figure 5.2 shows a graphical view of an example composite application using conduits. These classes represent different kinds of

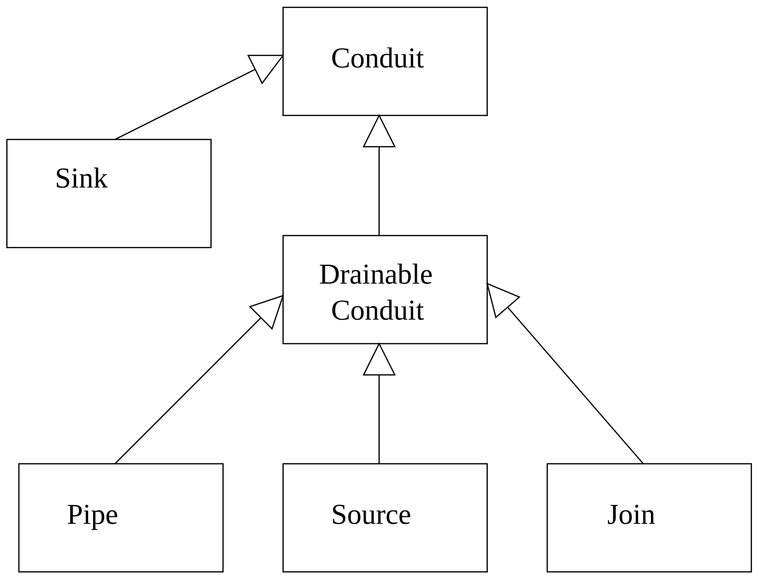


Figure 5.1: Conduits hierarchy

conduits:

Source: a source conduit produces a water flow.

Pipe: a pipe conduit has an incoming flow and produces an outgoing flow.

Join: a join conduit has two incoming flows and produces and outgoing flow.

Sink: a sink conduit has an incoming flow and “consumes” the water.

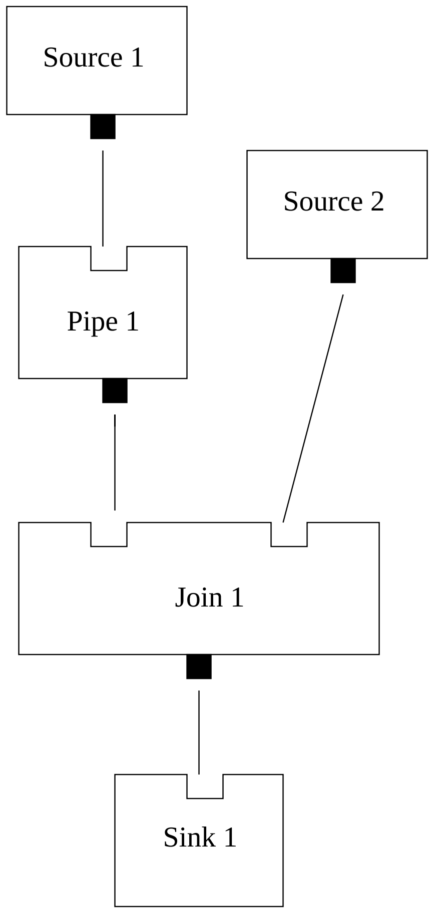


Figure 5.2: A composite application

A conduit with an outgoing flow can be connected to a conduit that expects and incoming flow. The behaviour of having an outgoing flow is factorized in the class DrainableConduit. This class implement the method drain, that is the method invoked to simulate the flow from one conduit to another.

An incoming flow in a conduit is identified by a parameter of type DrainableConduit in the constructor. The invocation of the constructor providing an instance of DrainableConduit as argument establishes the connection between the outgoing flow in the argument and an incoming flow in the created conduit instance.

The constructors of the conduits also have other parameters for configuration: the name and the capacity. The Source's constructor has an extra parameter, the "profile" which describes the flow of water the source instance will produce during the execution of the simulation, by specifying the amount of water to be produced in each period of time.

Figure 5.3 shows the Java code of the composite application from figure 5.2.

```
class Simulation{
   static public void main(String argv[]){
     final int profileSRC_1[][]={{10,1},{20,3},{40,1}};
     Source SRC_1 = new Source("Source 1",5,profileSRC_1);
     final int profileSRC_2[][]= {{10,2},{20,4}};
     Source SRC_2 = new Source("Source 2",4,profileSRC_2);
     Pipe PIP_1 = new Pipe("Pipe 1",4,SRC_1);
     Join JOI_1 = new Join("Join 1",4,SRC_2,PIP_1);
     Sink SNK_1 = new Sink("Sink 1",4,JOI_1);
   }
}
```

Figure 5.3: An example of the code of a simulation

In this framework we can identify the main elements in a component model. The components are the concrete classes representing the different kinds of conduits. The plugs are the incoming and outgoing water flows. As in Java Beans, the plugs are described by programming conventions: a subclass of DrainableConduit has an outgoing flow plug; a parameter of type DrainableConduit in the constructor of a conduit means an incoming flow plug.

A connection links an outgoing to an incoming flow. The connection is realized by providing the conduit with the outgoing flow as an argument to the constructor that creates the conduit with the incoming flow.

The components are configured with a name and a capacity. A source component is also configured with the description of the water flow it produces. The configuration is done by providing arguments to the constructor.

In the rest of the chapter we will explain the logic representation of the conduits component model.

Figure 5.4 shows the description in the logic notation of the example application.

5.2.1 Conduit components

As we explained in the previous chapter, the description of a component must specify directly or indirectly how to deploy, configure and connect the component. In the case of beans, this description was indirect and deduced using knowledge from the component model. In conduits, the description of each component specifies how to deploy the component, how to configure it and how to build the connections. As we explained earlier in this chapter, the creation of an instance of a conduit also configures it an establishes the connection. So, the creation of the instance of the component depends on the configuration and the connections.

The following rule specifies the deployment of the Pipe component, and how it depends on the configuration and connection. At the same time it creates the part, configures it and builds the connections for the incoming flow:

```
initialization(?comp,?p,
 {Pipe ?p = new Pipe(?n,?c,?s1);}):-
   part(?comp,Pipe,?p),
   configure(?comp,?p,name,?n),
   configure(?comp,?p,capacity,?c),
   connect(flow,?comp,
           ?s1,outconduit,
           ?p,inconduit<DrainableConduit<1>>).
```

Note that as the configuration and connections are required to instantiate the component, it will not be instantiated if they are not provided.

Although the component describes how to build the connections, what is a valid connection is standard for all the components, and it is described by the `flow` protocol. The plugs are indirectly described by some conventions: a component has an outconduit plug if it is a subtype of the DrainableConduit

```
composite(conduitcontainer, Simulation).
part(Simulation, Source, SRC_1).
configure(Simulation, SRC_1, name, {"Source 1"}).
configure(Simulation, SRC_1, capacity, {5}).
configure(Simulation, SRC_1,
          profile, {{{10,1},{20,3},{40,1}}}).

part(Simulation, Pipe, PIP_1).
configure(Simulation, PIP_1, name, {"Pipe 1"}).
configure(Simulation, PIP_1, capacity, 4).

part(Simulation, Source, SRC_2).
configure(Simulation, SRC_2, name, {"Source 2"}).
configure(Simulation, SRC_2, capacity, {4}).
configure(Simulation, SRC_2,
          profile, {{{10,2},{20,4}}}).

part(Simulation, Join, JOI_1).
configure(Simulation, JOI_1, name, {"Join 1"}).
configure(Simulation, JOI_1, capacity, {4}).

part(Simulation, Sink, SNK_1).
configure(Simulation, SNK_1, name, {"Sink 1"}).
configure(Simulation, SNK_1, capacity, {4}).

connect(inconduit,Simulation,SRC_1,
        outconduit,PIP_1,
        inconduit<DrainableConduit<1>>).
connect(inconduit,Simulation,
        SRC_2,outconduit,JOI_1,
        inconduit<DrainableConduit<1>>).
connect(inconduit,Simulation,
        PIP_1,outconduit,JOI_1,
        inconduit<DrainableConduit<2>>).
connect(inconduit,Simulation,
        JOI_1,outconduit,
        SNK_1,inconduit<DrainableConduit<1>>).
```

Figure 5.4: A composite conduits application

class and it has an inconduit plug if its constructor has a parameter of type DrainableConduit. So the information about the class and the constructor allow the protocol to deduce the plugs. This information is describe by the following facts in the representation of the Pipe:

```
class(Pipe).
extends_M(Pipe,DrainableConduit).
feature1(Pipe,constructor<[String,int,DrainableConduit]>).
```

5.2.2 The flow protocol

The description of the flow protocol specifies what are the valid connections. As each component determines how to build the connections, the protocol does not determine how to build the connections.

A valid flow connection links an outconduit plug to an inconduit plug. In the beans' protocols the validity of connection only depended on the compatibility of the plugs. In conduits, the validity of a flow connection depends not only on plug compatibility but also on the existing connections: it is not possible to connect an outconduit or an inconduit plug more than once. Another restriction to connections through the flow protocol is that a conduit cannot be connected to itself.

```
validconnection(inconduit,?comp,
                ?s,outconduit,?t,inconduit<?n>):-
   plug(?comp,?t,inconduit<?n>),
   plug(?comp,?s, outconduit),
   NOT(connect(inconduit,?comp,
               ?s,outconduit,?,inconduit<?>)),
   NOT(connect(inconduit,?comp,
               ?,outconduit,?t,inconduit<?n>)),
   NOT(equal(?s,?t)).
```

The variable `?` is a variable in whose value we are not interested. Each occurrence of `?` stands for a new variable.

As mentioned before, the plugs are deduced from the description of the component:

```
plug(?scomponent,outconduit):-
    subtype(?scomponent,DrainableConduit).
plug(?tcomponent,inconduit<?n>):-
    feature(?tcomponent,constructor<?args>),
        filter<equal<DrainableConduit>>(?args,?inconduits),
        GenNames(?inconduits,?innames),
        element(?n,?innames).
```

The previous rule deduces the presence of an inconduit plug from the argument of type DrainableConduit in the constructor method. It also assigns a name to the plug. This names are generated by the predicate GenNames,and consist in the name of the type and a number that serves to distinguish the plugs in the case there is more that one.

5.2.3 Conduit Container

The description of a container provides how to create a composite application, which are the valid parts and how to add them to the container. In the end the conduit container has to generate a class like the one that is showed in figure 5.4.

Parts of a conduit container are created from subclasses of Conduit:

```
    validpart(?comp,?conduit,?name):-
        composite(conduitcontainer,?comp),
        subtype(?conduit,Conduit).
```

The conduit container creates a class for the composite application. This class has a main method that hosts the initialization code for the parts of the application. Including the code for initializing the part in this main method establishes the link between the part and the container.

There is a dependency between the initialization code of the different parts. Recall that to establish a connection between an incoming and an outgoing flow, the conduit instance which as the outgoing flow plug has to be passed as an argument to the instantiation of the conduit with the incoming flow. For this reason, the conduit instance with the outgoing flow has to be created before the other one. So, the pieces of initialization code must be sorted according to these dependencies. It is the container who provides for this global restriction. The following rule creates the main method in the composite application class, by collecting and sorting the code for initializing all the parts:

```
    method_M(?comp, void, main,[String],{
      static public void main(String argv[]){
      ?firstsortedcode
      }}):-
```

```
        FINDALL(initialization(?comp,?part,?impl),?impl,?code),
        FIRST(sort<mustprecede>(?code,?sortedcode)).

    mustprecede(?i1,?i2):-
        initialization(?c,?p1,?i1),
        initialization(?c,?p2,?i2),
        connect(inconduit,?c,
                ?p1,outconduit,
                ?p2,inconduit<?>).
```

Note that there may be many possible orderings of the initialization code. As we only need one, we use the `FIRST` meta predicate that gives only the first solution.

5.2.4 Describing parts

The description of the valid parts and the creation of the parts is done directly by the container (valid part) and by the components (creation of the part).

5.3 Summary

In this chapter we discussed two experiments that show that our representation makes the tool and the component model independent.

In the first experiment, we extend Java Beans with two new protocols. One of the protocols allows to specify required services in a bean, and to bind them to provided services. This protocol is defined from scratch following the style of Java Beans of describing plugs by programming conventions. The other protocol is a defined on top of the synchronize protocol. The connection of two properties using this protocol is equivalent to two connections using synchronize where the roles of source and target are interchanged. The definition of this second protocol shows how we can manipulate protocols themselves to build more complex protocols, and how logic meta programming provides a powerfull means for this manipulation. The extended version of Java Beans can be manipulated using the prototype tool described in the previous section without any modifications, thus showing that a component model can evolve independently of the builder tool.

The second experiment defines a completely new component model. This component model is based on a pedagogical framework for building waterflow simulations. This is not a component model like Java Beans, but provides a framework for building domain specific applications by putting together a set of components, and in this sense it is similar to a component model. We identified the mechanisms characteristic of a component model in this framework and represented them following the structure for describing component models presented in section 4.3. This shows that this structure can be used to describe models other than Java Beans. The prototype tool can be used to assemble applications in this new component model, showing that the tool is independent of the underlying component model.

Chapter 6

Future Work

This chapter describes lines for further research coming up from this thesis. Some of the lines consist in overcoming the limitations of this work and others on further applying the approach presented in this thesis.

Conceptual Framework

The main features of component models we identified in this work provide a first step towards a characterization of component models. As component models are very diverse, a deeper study is needed to achieve a general foundation for component models.

One of the elements that differs the most, and which needs to be further investigated what is a part and how it is created from a component needs to be studied in more detail. All the analyzed models have a mechanism of deploying or linking a component into a composite application. This mechanisms vary from declaring a variable and putting some initialization code in a container class, like in Java Beans, to create a set of auxiliary classes and registering to a naming service as en EJB.

Representation of component models

This thesis focused mainly in the protocols for connecting components. The treatment of containers, creation of parts and configuration is limited. A continuation of this work should look more closely to this other mechanisms. About the container and the creation of parts, we conjecture that a container can be seen as a special kind of component and the linking of a part to a container as a kind of connection protocol.

In the connection protocols represented in this thesis the information used to determine the validity of a connection is limited to information like signatures and programming conventions. An extension of this work can take into account information about the behaviour of the components, like the order in which messages are to be exchanged.

In this work we have identified a generic structure in the representation of component models. This structure parallels the elements we identified as fundamental in a component model. The two representation of component models we presented follow this structure. However, each representations were built independently and no facts or rules are shared between them. The only share the underlying organization. A continuation of this work should extract this common structure or architecture and formalize it in such a way that it can be reused or specialized when building a new component model representations.

Representation of other component models

Another possible continuation of this work is to represent other industrial component models.

The experiment about the representation of the conduits component model (chapter 5) points out an interesting line to investigate. In this experiment, we started from an existing framework which already had some kind of components and composition mechanisms, although it did not belong to a "standard" component model. We identified and made explicit the composition mechanisms already existing in the framework. A research line suggested by this experiment is to capture and represent the component model of existing application families. This provides the advantage of migrating the application families into components without having to redo it according to an existing component model. The conduits examples is a simple and well structured case, research in this line should consider more complex applications and investigate if it is feasible to extract the component model and under which conditions.

Validity rules

The current representation supports constraining the assemblies that can be built so that only well-formed assemblies are made. It does not provide a means to ensure that all the connections and configurations needed by a component are made. In this case, what happens is that the code is not generated. An improvement for this would be reporting the requirements that are not fulfilled.

In this work to determine the if an assembly is well-formed we consider just the validity of each part and the point-to-point connections. An extension would consider validity rules that take into account the overall structure of the composite application. For example in the conduits model we may want to define a restriction stating that in the composite application the conduits are not connected forming a cycle.

Component granularity

In the component models we represented (and also in the other commercial models we investigated), a component corresponds to a class or to a set of classes with a class acting as a facade. In other words, a component corresponds to an existing language construct. An interesting line to study is about components of a different granularity, such as components that are more fine grained than a class (for example mixins) or cross-cutting. A work that considers components of a granularity different to a class is the work on Role-Based Programming[15] , where components are roles implemented as mixins. It would be interesting to investigate the representation of this model using our approach. Logic meta programming provides a good support for expressing entities that do not directly correspond to a language construct such as a class (it has already been applied to aspect-oriented programming [18]).

Another extension concerning the granularity of components is to study hierarchical composition, that is, the assembly of components to build more complex components.

Builder tools

The prototype tool presented in this thesis is just a proof-of-concept implementation, with very limited functionality. Another line of continuation would be developing tools with better facilities, as for example a visual user interface.

Summary and conclusions

In this section we presented some possible lines of continuation of this work. The treatment of containers and parts in this work being quite primitive, a continuation of this work should look more closely at these features. The representation of protocols could be improved to consider more information to determine the validity of connections. Further applications of the approach of this thesis that could be investigated are the representation of other existing component models, the study of component models with different granularity for components, and the extraction of component models from existing application families.

The convergence of all this lines would in the end lead to a powerful application assembling enviroment. This enviroment would integrate standard, customized and domain-specific component models.

Chapter 7

Conclusions

Currently, the knowledge about component models is embedded into the builder tools used to assemble applications using the model. The purpose of this thesis was to investigate if this knowledge can be extracted from the builder tool and represented separately in such a way that it is possible to build generic builder tools capable of working with different models or variations of the models. The approach taken was to use logic meta programming, using facts and rules to represent and manipulate the knowledge about the component models.

As a first step towards the representation of the component models, we studied literature in the area and several existing industrial models to identify what elements need to be represented. From this study we extracted a set of features that we consider fundamental and set up a conceptual framework. We analyzed existing industrial models according to this conceptual framework. This analysis showed that the chosen features characterize the different models.

To provide a representation that is useful for a builder tool, we identified what is the information the tool needs from the representation of the component model to support its operation. As the representation is done using a logic language, we formalized this requirements an an interface consisting of a set of queries and facts.

To show that the representation is feasible, we represented two different component models: Java Beans and the conduits model. The representations are structured according to the elements in the conceptual framework and conform to the interface required by the tool. We choose Java Beans as a case study because it is an industrial, “real case” model, which is accessible and simple enough to be covered as an experiment. We extended Java Beans with new protocols, what shows that the representation proposed supports the evolution of the component models. The other model we represented is not a “real” model, but a framework which allows to assemble applications in a specific domain. In this framework we could identify the main elements from the conceptual framework and represent them according to the interface required by the tool. The two models are different, as one is an industrial, general purpose model and the other is an experimental, domain specific model. The composition mechanisms in them also differ: in Java Beans the component instances are created and afterwards configured and connected by method invocations, while in Conduits the component instances are configured and connected when they are created, by providing arguments to the constructor. Both models can be represented with the same interface and the same underlying structure based on the fundamental elements. This provides evidence supporting the genericity of the kind of representation proposed.

To show that the representation proposed supports the operation of a generic builder tool, we implemented a prototype tool that interacts with the logic representation of a component model to assist the user in assembling applications in the model. The prototype tool can be used to build applications in Java Beans, in its extended version and in the conduits model without any change. This provides evidence supporting the independence between the tool and the component model: in the first experiment the component model evolved and we continue using the same tool, in the second experiment we provided a completely new component model and we can still use the same tool.

Summarizing, in this thesis we wanted to show that the knowledge about the component model can be separated from the builder tool, allowing to develop generic builder tools. As a first step, we identified what are the main elements in a component model that need to be represented and what information must be provided by the representation to support the operation of a builder tool. As a second step we showed that it is possible to represent component models satisfying the requirements from step 1, by the representing two different component models and a variation of one of them. As a last step we showed that the representation proposed allows to develop generic builder tools by implementing a prototype tool capable of operating with the different models represented in step 2.

As a final conclusion we can say that the knowledge about the component model can be separated from the tool, allowing to develop generic builder tools and that a logic programming language provides a good formalism for representing this knowledge.

Bibliography

[1] X. Pintado. Gluons and the Cooperation between Software Components. In [3].

[2] O.Nierstrasz, D.Tsichritzis, V. de Mey and M.Stadelmann. Object + Scripts = Applications. In the Proceedings of Espirit 1991 Conference, Kluwer, Dordrecht, 1991, pp.534-552

[3] O. Nierstrasz and D. Tsichritzis, eds. Object Oriented Software Composition. Prentice Hall 1995.

[4] V. de Mey. Visual Composition of Software applications. In [3]

[5] The JavaBeans API Specification. http://java.sun.com/beans

[6] C. Szyperski. Component Software. ACM Press. 1997

[7] O. Nierstrasz and L.Dami. Component-Oriented Software Technology. In [3]

[8] Sun Microsystems. Specification of the Enterprise Java Beans 1.0 architecture.

[9] J-M. Geib, C. Gransart, P.Merle.. CORBA:des concepts a la pratique, Les composants CORBA. http://corbaweb.lifl.fr/CORBA_des_concepts_a_la_pratique/

[10] R.Allen and D. Garlan. A Formal Basis for Architectural Connection. http://www.cs.cmu.edu/afs/cs/project/able/www/able/. A revised version of the paper with the same name that appeared in ACM Transactions on Software Engineering and Methodology, July 1997.

[11] R.J. Allen, Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144. May 1997.

[12] CORBA 2.0 Specification. http://www.omg.org/corba.

[13] J.G Schneider, O. Nierstrasz. Scripting: Notes from the Tutorial Higher-level Programming for Component-based Systems helded at ECOOP'99.

[14] K. De Volder. Type-Oriented Logic Meta Programming. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[15] M.VanHilst and D.Notkin. Using Role Components to Implement Collaboration-Based Designs. In the Proceedings of OOPSLA'96.

[16] Jini Architecture Specification. http://www.sun.com/jini/specs/.

[17] Jini Architectural Overview. http://www.sun.com/whitepapers/.

[18] K. De Volder. Aspect Oriented Logic Meta Programming. In the Proceedings of Reflection'99.