AN EXTENSIBLE WORKBENCH FOR THE CommUnity ADL

AN EXTENSIBLE WORKBENCH FOR THE CommUnity ARCHITECTURE DESCRIPTION LANGUAGE

By

JORGE SANTOS, B.Sc.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2007)	McMaster University
(Computer Science)	Hamilton, Ontario

TITLE:	An Extensible Workbench for the COMMUNITY Architecture Description Language
AUTHOR:	Jorge Santos, B.Sc. (Universidad Nacional Autonoma de Mexico, Mexico City, Mexico)
SUPERVISOR:	Doctor Thomas Maibaum
NUMBER OF PAGES:	x, 85

Contents

List of Figures

Abstract

The field of Architecture Description Languages (ADL) is in rapid and constant evolution. Change and experimentation with different language features is the norm. Additionally, it is unlikely that one ADL will ever satisfy the needs of every architect. On the other hand, ADL experimentation and usage require the use of easy-to-use tools that will help with the research into different characteristics of ADLs. This leads us to the need for highly extensible tools that will make it easy to work with and evolve ADLs. This thesis presents the design of a new tool developed to work with the CommUnity Architecture Description Language with the goal of being a highly extensible platform for future experimentation with the language.

Acknowledgments

There are too many people that have helped me here to list, mainly because I know I would probably forget to mention someone. Nevertheless, I will mention my wife, Edna, which has helped me tremendously in the small and big things, without whom these last months and the rest of my life I cannot imagine. I would also like to thank my parents and sisters, who have always been there for me in the good and the bad. I also have a debt of gratitude towards my friends, with whom I have had the luck to share a little bit of my life. Ic ertainly must thank my supervisor, Tom; without his support, confidence and help I would not have been able to do this work. My examiners were also very helpful and insightful, they increased tremendously the quality of this work. Thanks also go to my many teachers, from my undergrad years all the way to the ones that instructed me during my Masters studies, I learned something from all of them.

Chapter 1

Introduction

Every software system of significant size must be organized in such a way that its complexity can be effectively managed. Thus, such systems are best organized around an *architecture*. This organization has implications for the performance, security, performance and scalability of the system. It is therefore desirable to document this architecture and give it an explicit representation amenable to documentation and analysis.

In order to fill this need, Architecture Description Languages (ADLs) have been created. These languages allow the representation (more or less formally) of a software system's architecture. This allows the architect to better communicate and analyze her design. ADLs are an active area of research and these languages are in a state of rapid evolution.

CommUnity is an ADL geared towards formal specification and analysis of Software Architectures. It draws ideas from category theory, parallel program design languages, coordination languages, and architecture description languages to obtain a framework for the formal specification of open, reconfigurable, mobile systems in which the computation, coordination, and distribution dimensions are explicitly separated [25].

Even though ADLs are interesting and valuable formalizations in their own right, they are of little practical use without appropriate tools to create and analyze their architectural descriptions. It is therefore of paramount importance to have robust and usable tools to support work with ADLs. Furthermore, since ADLs are in constant evolution, it is very important that these tools be extensible, since researchers will often want to try new ideas and adapt the languages to the problems of their interest.

1.1 Objective

Although there is an existing (partly) graphical editor for the CommUnity ADL, it has not been designed with extensibility in mind and it has proven hard to modify and maintain. Therefore, the objective of this research is to develop a solid and extensible graphical editor and platform for manipulating and analyzing CommUnity architecture descriptions. This tool should provide an easily extensible base that will allow for the creation of the necessary additional tools that will be necessary for working and experimenting with extensions to the core CommUnity language. It will therefore be of paramount importance to pay particular attention to the ways in which this extensibility will be realized and to have confidence that the resulting system will be indeed useful for this purpose.

1.2 Introduction to Software Architectures

From [2]: "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." The architecture of a system focuses on the major components of said system and models the interactions between these components, without regard to their internal mechanisms. Unlike in the design of a system, the primary interest of a system architecture is the *externally visible* components of its elements, such as the services they provide, their performance characteristics, usage of shared resources and so on, i.e., public aspects of the system's elements, as opposed to their private aspects.

This descriptions can be static, such as the interfaces of the modules, their shared data and so on. They also comprise dynamic aspects, such as the synchronization between the several software modules.

It is worthwhile to note that every software system has an architecture, whether described explicitly or not. It is in explicitly describing the architecture of a system that ADLs come into play. Architecture Description Languages are intended to describe the architecture of a (software) system.

Architecture Description Languages, then, are languages, more or less formal, that allow us to describe the architecture of a computer system. ADLs are still maturing, there is little consensus on what exactly constitutes an ADL and what characteristics it should have. Some of the most common ADLs are: Aesop [9], C2 [24], Darwin [21], Rapide [20], UniCon [29], Acme [11] and Wright [1].

1.3 Introduction to ADLs

The *architecture* of a software system is a term in common use among software engineers; however, this architecture is often described in an informal, ad-hoc way. The use of Architecture (or Architectural) Description Languages allows the description of this architecture to be realized in a formal way. This aids communication by giving a common language in which to describe said architecture, allows formal analysis and simulation, and aids in the reutilization of software components, among other advantages.

1.3.1 Characteristics of ADLs

Although there is no agreement on exactly what properties an ADL should have to be considered an ADL, according to [23] they must explicitly model *components*, *connectors*, and *configurations*; furthermore, to be truly usable and useful, it must provide *tool support* for architecture-based development and evolution. These four elements of an ADL are further broken down into constitutive parts.

In [11] it is argued that ADLs share a common ontology (or conceptual basis), the elements of which are:

Components represent the computational and data aspects of a system. They are similar to the classes in object-oriented design, and to the boxes in box-and-line descriptions of software architectures. Some examples of components would be clients, servers, and databases. The components have at least one interface, and possibly more, that allows them to interact with the other elements of the environment.

Connectors allow components to interact, since they cannot do so directly. The connectors correspond to lines in box-and-line diagrams. They specify the means of interaction between components. They may represent simple communication channels, such as buffers or shared variables, or more complex ones, such as a connection to a database or a communication protocol. Connectors also have interfaces, they specify the way to interact with the various participants in the interaction represented by the connector.

Systems are connected graphs of components and connectors that describe architectural structure. They represent configurations of said elements. A system descriptions' overall topology is defined independently from the components and connectors that make up the system (in contrast with programming languages, in which modules are usually tied via import statements). Another important characteristic of systems is that they are possibly hierarchical, that is, components and connectors may have internal architectures. They are akin to the *configurations* of [23].

Constraints are akin to module invariants, they represent claims about an architectural design that should remain true even as it evolves over time. Some of the usual constraints include restrictions on allowable properties, topology, and design vocabulary. For example, an architecture might constrain its design so that users of a server belong to a certain group.

Styles represent families of related systems. An architectural style typically defines a vocabulary of design element types and rules for composing them. Examples include layered systems, and data-flow architectures based on graphs of pipes and filters. Some architectural styles additionally prescribe a framework as a set of structural forms that specific applications can specialize. Examples include the traditional multistage compiler framework, 3-tiered client-server systems, and user interface management systems.

Regarding the tool support mentioned by [23], it is worth noting that although the tools supporting an ADL are not formally part of the language, they are necessary for it to be useful. There is a push from the software engineering community to identify a canonical "ADL toolkit" [10]. Some desirable abilities provided for the tools of an ADL are architectural design, analysis, evolution, executable system generation, and so forth.

1.3.2 Differences Between ADLs and Other Languages

In order to more clearly see what ADLs are, we can contrast them to other notations that, though similar, are not properly ADLs. The languages we use for comparison are implementation languages, object-oriented modeling notations, and module interconnection languages (MILs). The main criteria that distinguishes ADLs from other languages is the need of ADLs to model *configurations* explicitly.

In implementation languages the architecture of a system is only implicit, via subprogram definitions and procedure calls. MILs typically describe *uses* relationships among modules in an *implemented* system and support only one type of connection.

Object oriented modeling languages, such as UML, can be extended to support modeling of software architectures [22][27] to be able to model architectural abstractions that either differ or do not exist in object oriented design. This has the advantage that there already are good tools for working with UML, and it is a widely known and used language. The different ADLs have certain aspects in common with UML, some of which can be expressed with UML's extension mechanisms, while others may be included in a UML specification but can only be interpreted by ADL-specific tools [27]. Moreover, it is convenient to use a language that closely matches the concerns facing the software architecture, by making the peculiar aspects of architecture modeling (e.g. components and connectors) explicit the job of the software architect is made easier.

1.4 Some Sample ADLs

There is a multitude of ADLs with very diverse focuses and characteristics. In this section we will describe some of the more popular ones to give a more concrete idea of what they look like and what they may be used for. For a thorough description and comparison of a large number of them see [23].

1.4.1 Wright

Wright [1] is an architectural description language based on the formal description of the abstract behavior of architectural components and connectors. It is distinguished by the use of explicit, independent connector types

```
Configuration SimpleSimulation
      Component TerrainModel(map : Function)
          Port ProvideMap = [Interaction Protocol]
          Computation = [provide terrain data]
      Component = VehicleModel
          Port Environment = [Interaction Protocol]
          Computation = [compute vehicle movement]
      Connector UpdateValues(nsims : 1..)
          Role Model_1..nsims = [Interaction Protocol]
          Glue = [Data travels from one Model to another]
      Instances
          Hamilton : TerrainModel([map of Hamilton])
          Bus : VehicleModel
          C : UpdateValues(2)
      Attachments
          Hamilton.ProvideMap, Bus.Environment as C.Model
  End SimpleSimulation
```

Figure 1.1: A sample system

as interaction patterns, the ability to describe the abstract behavior of components using a CSP-like notation, the characterization of styles using predicates over system instances, and a collection of static checks to determine the consistency and completeness of an architectural specification. Because the semantics of Wright specifications are formally defined, an architecture characterized in Wright provides a sound basis for reasoning about the properties of the system or style described.

To give an overview of Wright we now illustrate its main ideas via a simple example. The system will simulate a bus driving through Hamilton. It will have two components, one for simulating the bus and its movements, and another simulating the places through which it drives through. The two components communicate by transmitting updates to the values of the objects' attributes. Figure 1.1 shows the outline of how this would look like in Wright. Of note in this description is the explicit specification of components and connectors, as well as the delineation of instances and their attachments. In our example, the terrain model `Hamilton` is accessed by the vehicle `Bus` component, and will interact with its environment via the `Environment` port.

A connector represents an interaction among a collection of components. For example, a pipe represents sequential communication between two filters, while an RPC connector represents one component requesting a service of another. A Wright description of a connector consists of a set of *roles* and *glue*. A connector, in our example `C`, acts as a source of data and recipient of data, one for each model it coordinates. The connector glue defines how the roles will interact with each other.

The parts of a Wright description (port, role, computation and glue) are described using a variant of CSP [13]. For example, the Model role of UpdateValues might be defined by:

$$\begin{aligned}\textbf{Role}\ \mathsf{Model} \;=\;& \overline{\mathsf{update!x}} \rightarrow \mathsf{Model}\\ & \sqcap\ \overline{\mathsf{request}} \rightarrow \mathsf{newValue?y} \rightarrow \mathsf{Model}\\ & \sqcap\ \S\end{aligned}$$

This defines a participant in an interaction that repeatedly either provides an updated value ($\overline{\mathsf{update!x}}$) or request a new value ($\overline{\mathsf{request}}$). If it requires a new value, it will be provided one ($\mathsf{newValue?y}$). It may also choose to terminate successfully at any time (§).

One immediate benefit of describing architectural designs with Wright, obtained from making the meaning of an architectural description precise, is that it facilitates the communication of ideas from the architect to the other interested parties. For example, the Model role defines exactly what actions a component may or may not take if it is to participate in an UpdateValues interaction. Furthermore it provides a basis for analyzing architectures. The description of connectors in Wright can be used to determine whether the connector satisfies certain critical properties, such as internal consistency of the protocol and whether the roles are sufficiently constrained to ensure proper behavior by the participants. In considering the UpdateValues connector above, for example, we notice that, as it is described, if both `Bus` and `Hamilton` were to choose to request a value before providing an update, a conflict would occur. Both expect a value and there is no value available. In addition to analyzing connectors, components can be analyzed to determine, for example, whether they conform to their interface specifications.

Wright further structures the description of an architectural configuration by distinguishing between component or connector types and specific instances of them in the configuration. In the example, UpdateValues is a connector type: it is defined by a set of potential participants, the Models, and constrains how they may behave, via the Glue. C is an instance of this type: the two participants of which are Hamilton and Bus, which are associated with the protocol in the attachments.

Since the global system behavior is derived from the architectural structure and behavior descriptions of types, Wright provides a means of extending the type-level guarantees to system instances. At the configuration level, Wright provides checks to confirm that a given component port properly fulfills the obligations of any role to which is attached. If the appropriate constraints are met, then any analyses at the type level automatically apply to instances.

In addition to describing and analyzing system configurations, Wright permits the designer to describe and analyze entire families of systems, or architectural styles. By formalizing a style, the architect is able to leverage analysis across many systems and thus reduce the effort to produce individual systems.

1.4.2 Acme

Acme was originally conceived as a language to interchange architectural representations between various implementations of the different ADLs; however, it has evolved into an ADL on its own. The creators of Acme call it a *second generation* ADL [11], meaning that it has been built on the experience of other ADLs, and has been built with the intention of providing the basics of ADLs with a simple syntax.

Nowadays the Acme language and toolkit provide three different capabilities:

Architectural interchange as was its original goal, Acme provides a generic interchange format for architectural design, thus allowing architects using Acme-compatible tools a wider access to analysis and design tools.

Extensible foundation for new architecture design and analysis tools. Acme provides a solid, extensible foundation and infrastructure that allows tool builders to avoid needlessly rebuilding standard tooling infrastructure.

```
System simple_cs = {
    Component client = { Port sendRequest }
    Component server = { Port receiveRequest }
    Connector rpc = { Roles {caller, callee} }
    Attachments : [
        client.sendRequest to rpc.caller ;
        server.receiveRequest to rpc.callee }
}
```

Figure 1.2: Simple Client-Server System in Acme.

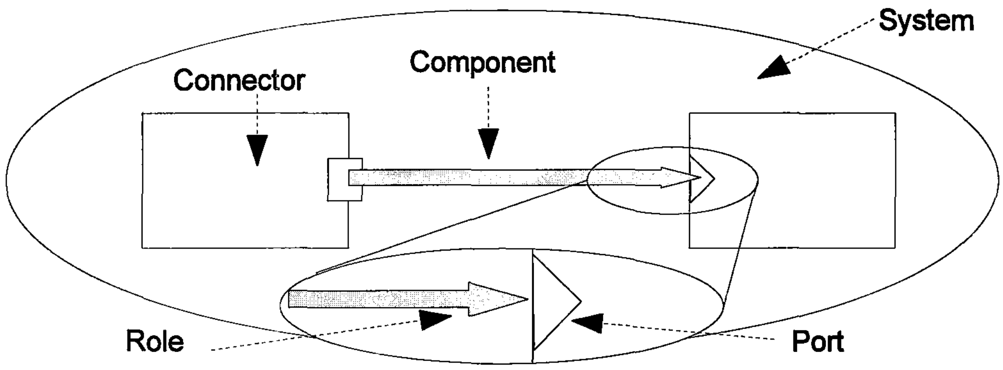


Figure 1.3: Elements of an Acme Description

Architecture description by itself. Although not appropriate for all applications Acme can serve to describe relatively simple software architectures.

To illustrate the characteristics of Acme we will go trough a small example: a simple architecture in which a *client* component is declared to have a single *send-request* port, and the server has a single *receive-request* port is shown in Figure 1.2. The connector has two roles designated *caller* and *callee.* The topology (configuration) of the system is defined by listing a set of *attachments* that bind component ports to connector roles, and a graphical representation can be seen in Figure 1.3.

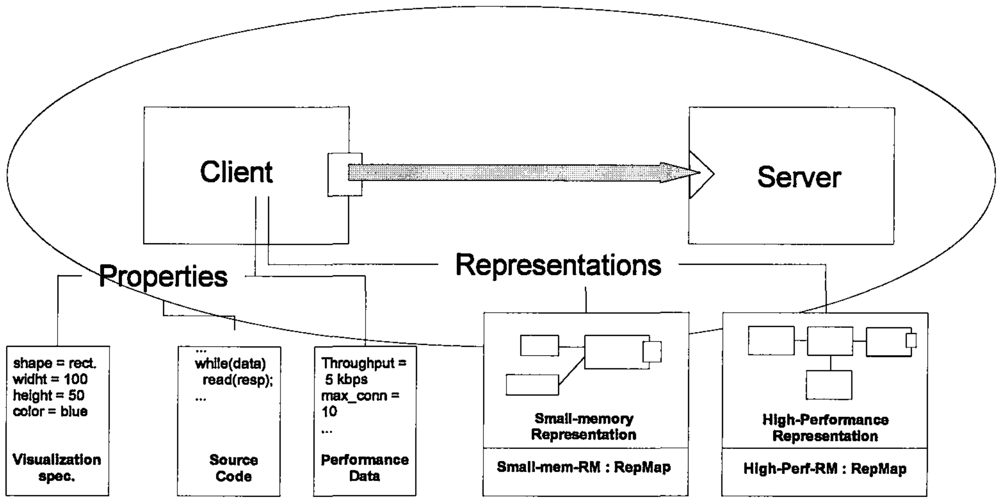


Figure 1.4: Representations and Properties of a Component

Acme allows any component or connector to be represented by one or more detailed, lower level descriptions, called *representations*, in order to support hierarchical descriptions of architectures. The ability to associate multiple representations with a design element allows Acme to encode multiple views of architectural entities. Representations of a component are illustrated in Figure 1.4.

1.4.3 Rapide

Rapide [20] is an ADL focusing on large-scale, distributed systems. It allows the definition and execution of models of system architectures. The result of executing a Rapide model is a set of events that occurred during the execution together with *causal* and *timing* relationships between events. These sets of events together with their causal histories form a *poset* (a partially ordered set) [38].

Rapide 1.0 is structured as a set of languages consisting of the Types, Patterns, Architecture, Constraint, and Executable Module languages; called the Rapide *language framework*.

Rapide implements an interface connection architecture model. It provides tools to express the functionality offered by an interface, the functionality required by other modules/interfaces and the connections between interfaces. Rapide also allows for expressing the requirements/constraints an interface behavior has to exhibit.

The main elements that define a Rapide interface are:

Actions represent a "one-way" message to be sent or received by the interface. They are asynchronous from the point of view of sender and receiver.

Functions represent a typed request/replay pair with synchronous interaction between the involved interfaces.

Behavior An interface behavior can be expressed in three ways, either attaching an implementation module to the interface, defining an architecture that implements the interface, or describing its behavior by means of reactive rules that specify the reaction of the interface to events offered to it.

Actions and functions may be grouped in services to aid in their reusability.

Interfaces are assembled into an architecture by using connections. Connections, as is usual in ADLs, are dynamic entities. Rapide allows connection behavior to be specified in terms of the relationships that the events going into a connection and the ones coming out have.

The semantics of a connection in Rapide is such that when the triggering event is present (expressed in the left hand side of the connections), the connection triggers and produces the event specified in the right hand side of the connection. Three types of connections are supported by Rapide: Basic connections (`A to B`), pipe connections (`A => B`), and agent connections (`A ||> B`).

Basic connections are identity connections, events `A` and `B` are the same.

Pipe connections behave as a single thread control when producing `B` events, regardless of the concurrency behavior of the triggering `A` events.

Agent connections behave as if each `B` event is generated by a different thread of control, and thus its produced `B` events are not related to each other.

The main distinguishing characteristic of Rapide is its model of computation based on Partially Ordered Sets of Events (posets). Each event represents the occurrence of an activity within a program at a particular level of detail. Events are generated by communication between two components of the system via actions and functions. Actions generate a single event, while functions generate two events; one corresponding to the function invocation and another to the function return. Events in Rapide are typed, that means that they are characterized by the number, order and type of their arguments.

Events can be ordered both by time and causality. Each of these criteria yield partial orders on the event set of a computation.

Time order is specified with respect to local clocks, since in Rapide there is no required global clock. Events are ordered with respect to the clocks that apply to it, so times can only be compared with times obtained by clocks in its scope. This fact of multiple clocks and the relationship of events to clocks imply that events that refer to different, non related clocks are *not* ordered with respect to each other.

Causal order represents the generator/generated events. Both interfaces, via their behavior and connections may generate dependent events. Two events `A` and `B` are dependent (`A` precedes `B`) if:

1. `A` and `B` are generated by the same process or
2. A process is triggered by `A` and then generated `B` or
3. A process generated `A` and then assigns to a variable `v`, another process reads `v` and then generates `B` or
4. `A` triggers a connection which generates `B` or
5. `A` precedes `C` which precedes `B` (transitive closure).

By introducing this concept of order and causality, Rapide enables the architect to explicitly visualize and analyze the execution of the system. In a more sophisticated use of this facility, a system behavior may be expressed as constraints on how events can relate to each other.

1.4.4 UniCon

UniCon is an architectural description language whose focus is on supporting the variety of architectural parts and styles found in the real world and on constructing systems from their architecture descriptions.

As in other ADLs, there are *components*, where data or computation are located, and *connectors*, which are used to connect components. The components export *players*, which serve as input or output points. These players connect to connector's *roles*, and thus communicate with other components. Both components and connectors have a *specification part* and an *implementation part*.

Components are specified by an *interface*, which describes three things: its computational commitments, constraints on its usage, and performance and behavior guarantees. It contains three types of information:

Component type is similar to the type of an object in an object oriented language. The type of a component captures the semantics of its behavior, the kind of functionality it implements, its performance characteristics, and its expectations of the style of interaction with other components.

Properties are attribute-value pairs that specify additional information about a component as a whole, such as assertions or constraints.

Player definitions are the way in which a component interact with other components, via connectors.

Connectors are specified by a *protocol*, which defines the kind of communication possible among a collection of components and provides guarantees about those interactions. It contains three types of information:

Connector type expresses the designer's intentions about the general class of interactions to be mediated by the connector.

Properties are attribute-value pairs that specify additional information about a component as a whole, such as assertions or constraints, just as in components.

Role definitions give the requirements and responsibilities for the players in a connection. They are the elements to which components' players associate in a system.

Component implementations can be *primitive* or *composite.*

A primitive implementation is a pointer to a a source document external to the UniCon language that contains the implementation. For example, it could be an object file or a C language source code file.

A composite implementation is a description of other components and connectors defined with UniCon. It contains three types of information:

Pieces are the specific component and connector *instances* used to create the configuration description.

Configuration information is a description of the way in which components are hooked together to form a *configuration.*

Abstraction information is a description of how the players in the component interface are implemented by players in the component instances of the composite implementation.

1.5 Concluding Remarks

Architecture Description Languages, if used consistently, can be an useful tool in the development of large systems. By having a formal description of the architecture of the system, is possible to communicate clearly the design of the system to interested parties, as well as analyze the system before building it.

Wright, by allowing the detailed description of components and connectors, allows detailed analysis of components and connectors, allowing, for example, to determine if components conform to their interface specifications.

Of the four ADLs reviewed in this chapter, Acme seems to be the more mature one. This is probably due to the fact that it is based on older ADLs and was originally meant to be an interchange language for several different tools, thus is encompasses the common elements of other, older, ADLs. Nevertheless it was not meant to be an ADL by itself, instead aiming at becoming a basis for the development of other ADLs, so it is not as powerful as could be required for complex projects. It may also be noted that it does not provide any support for formal specifications by itself.

Rapide focuses on modeling and simulation of the dynamic behavior described by an architecture. It also has code generation and has a strong notion of event-based communication.

UniCon allows the construction of systems from their architectural descriptions, this may have the advantage of simplifying the mapping of the architectural motel to the implementation of the system, but has the disadvantage of constraining said implementation.

ADLs are still not as well developed as, for example, programming languages or even Object Oriented Design Languages (such as UML). There are many of them and they differ in several areas, such as area of application. Furthermore, having a generally applicable and widely available ADL with good tools would go a long way in establishing the use of these tools in more projects. However, it is not clear if a single ADL can be flexible enough to model the architecture of all categories of systems.

1.6 Support Environments

Architecture Description Languages are interesting formalisms in their own right, and useful for reasoning about the architecture of computer systems. However, if we are to use them for describing real systems, we need tools to aid us in the design and analysis of said systems.

In this sections we will describe a few of the tools that have been developed to work with ADLs and talk about their advantages and weaknesses.

1.6.1 Wright

There is a tool to convert an ASCII or LaTeXrepresentation of a Wright specification to a CSP specification. This tool also inserts some of the checks pre-defined by Wright (like connector deadlock freedom) into the CSP specification making them directly invocable from FDR menus. The CSP specification can then be checked by the FDR checker, where counter-examples may be viewed for failed checks [14]. FDR is a refinement checker for establishing properties of models expressed in CSP [18].

This tool does not seems to be maintained anymore [40] and we were unable to locate any information regarding licensing.

1.6.2 Acme

The Acme ADL is supported by AcmeStudio, an architecture development environment, written as a plug-in to IBM's Eclipse IDE Framework [28].

AcmeStudio supports the development of architectural models and architectural styles as defined by the Acme ADL as well as viewing and defining of elements' properties, rules, substructure, and typing.

For the creation and modification of styles, AcmeStudio provides a style editor, which allows component, connector, and interface types to be defined, as well as required properties and substructure of these types. The style studio can also be used to assign rules to the style, these rules prescribe the composition of models belonging to the style.

These rules are checked by AcmeStudio while creating models of the corresponding style, and it informs the architect of the model of the correctness of the architecture.

AcmeStudio also provides editors allowing the specification of several graphical characteristics of the defined types, thus aiding in the visualization of an architecture. This characteristics include the shape, color, size, icon, label, and layout policies of components and connectors.

Additionally AcmeStudio allows the integration of different analysis tools to support analysis beyond the built-in rule evaluation. This allows the development of analysis tools suitable for each different style. Using the Eclipse plug-in mechanism, AcmeStudio provides access to architectural models and provides notifications of changes to these models to allow tools to update their analyses. To allow the visualization of the analyses' results, AcmeStudio allows plug-ins to add specific views, such as tables and actions to the user interface [28].

Development of AcmeStudio

As of this date, AcmeStudio appears to be in active development [30]. There is a forum for plug-in developers [31], however activity in this forum by AcmeStudio developers seem to be sporadic.

The license for AcmeStudio does not allow redistribution of the program [32].

Final Remarks

AcmeStudio is an ADL tool with a solid technical and research foundation and is in active development, however, the relative closedness of the development is a definite disadvantage.

1.6.3 Rapide

There is a Rapide compiler, "rdpc" that translates programs written in Rapide-1.0 into executable load modules, or into library information that may subsequently be used to build executables. It is normally used in conjunction with analysis tools such as the "pov" tool to build and analyze a Rapide simulation. The Rapide library management system allows libraries of Rapide components to be compiled separately for later re-use and for organization of source files. It includes standard commands for creating, destroying, cleaning, and examining libraries of components [37].

There also exists a Partial Order Viewer, "pov", a tool for graphically browsing the partial orderings of events (posets) produced by Rapide computations. It reads log files generated by the execution of a Rapide program, and displays a graphical representation of the resulting poset. The pov also has facilities for "filtering" the poset to include only events of interest [37].

This tools also seems to be unmaintained and the license forbids redistribution [37].

1.6.4 UniCon

UniCon has a set of tools for working with graphical and textual representations of the language: a compiler for the textual form, a graphical interface, a semiautomatic wrapper-generator, and a facility for invoking the RMA analysis tool [29].

This tools can generate artifacts that assist in constructing the finished system. The UniCon compiler itself is built from a UniCon specification [29]. This allowed the researches to acquire experience in what the requirements for a practical ADL would be.

The tools also support analysis of specifications using external tools (such as real-time properties) and incorporation of the results back into the architectural description [29].

Unfortunately the tools do not seem to be supported nor available anymore [39]. In any case, there is no indication that the implementors were concerned with extendibility beyond the interaction with external tools.

1.6.5 ArchStudio

ArchStudio is a development environment for software systems architectures based on the xADL 2.0 architecture description language. It intends to support modeling of the hierarchical structure of complex systems, the types of various components, connectors and interfaces, product-lines of systems that are related by a common base, and so on [33].

The ArchStudio environment allows the manipulation of architectural descriptions using one of several views, while maintaining consistency among them.

It provides Archipelago, a box-and-arrows editor to manipulate architectures, similar to the common diagram editing tools. It also provides ArchEdit, a syntax-directed editor that adapts to new xADL schemas automatically with no recoding. The Type Wrangler tool provides a custom view of an architectural model that makes it easier to achieve type consistency [33].

ArchStudio also provides a framework called Archlight that allowing the testing of architecture descriptions against different criteria. Errors can be displayed and inspected, and users can navigate to the site of a problem in any editor with a few mouse clicks. This tests are provided by Archlight plug-ins, and users can provide new tests and analysis engines. Archlight ships with the powerful Schematron XML constraint engine, which allows complex architectural tests to be specified in about a dozen lines of code [33].

ArchStudio itself was developed using xADL. Whenever ArchStudio starts up on a machine, its architecture description is used to instantiate and connect components and connectors in the architecture [33].

The ArchStudio development environment is implemented as a set of Eclipse plug-ins. Eclipse is an IDE and development platform written in Java that provides for extendibility by providing a plug-in architecture for adding new functionality. ArchStudio itself is written in Java 2 Standard Edition (J2SE) version 5.0, also known as Java 1.5 [33] and should run on any platform that supports Eclipse, including recent versions of Windows, Mac OS X and Linux.

ArchStudio is an open-source project and is free to download and is actively maintained and has an active development team [33].

1.7 Conclusions

The field of Architecture Description Languages is evolving quickly and this is reflected in the fast evolution of the tools used to work with them. Often, as old ADLs die and their best ideas are incorporated into new ones the old tools are the victims of bit rot. Additionally, the effort to build a capable ADL environment is substantial and the user base of such programs is probably fairly small.

Therefore, the approach taken by AcmeStudio and ArchStudio seems the one with the most likelihood of a healthy future: Extend a best of breed IDE and allow these extensions to be extended themselves. In this way the number of code bases can be minimized and improvements to the common parts propagate without extra effort to dependent projects.

The ArchStudio team went a step further by choosing an open-source license for its product. This gives confidence in the continued development of the project and allows the possibility of third parties contributing bug fixes and features that the maintainers may be unwilling or unable to provide, and increases the number of potential developers of the project.

1.8 Contributions

This research has produced a solid and extensible graphical editor and platform for manipulating and analyzing CommUnity architectural descriptions. This will make possible the creation of extension tools for working with extensions to the language. This has been made possible by choosing a flexible and powerful model-driven development approach and by using modern tools designed for extensibility and of proven quality.

Chapter 2

Community and Friends

CommUnity is a platform for research into formal aspects of Software Architecture. It draws ideas from category theory, parallel program design languages, coordination languages, and architecture description languages to obtain a framework for the formal specification of open, reconfigurable, mobile systems in which the computation, coordination, and distribution dimensions are explicitly separated [25].

In this section we will describe the Community language as well as DynaComm and extension to model dynamically reconfigurable systems and systems with subcomponents. This chapter is heavily based on [26].

2.1 Description of the language

2.1.1 Designs

CommUnity's most basic unit of architectural description and conceptual unit of computation is the *design* (also known as component). The components can have *input*, *output* and *private variables* (referred to as *channels* in [19]). The input variables are set by some other component in the environment and the design has no control over their contents. The output variables are set by the design, to be read by the environment, and the private variables are for use of the design, without interaction from the environment. Together, the internal output and private variables are called *local* variables.

Additionally CommUnity designs can have *public* and *private actions*. Public actions can be coordinated with the public actions of other components, *private* ones can not. The actions have a name and a body. Each body has two *guards*, the *safety* guard, and the *private* guard, which are propositions over local variables (since input variables are not controlled by the design). These guards establish a logical interval within which the enabling condition of the action must lie, with the safety guard being the lower bound. The enabling guard must be implied by the safety guard, that is, it should not be possible for the enabling guard to be true while the safety guard is false. The maximal interval is achieved by setting the safety guard to true and the progress guard to false. It is possible to write only one guard if the two are equivalent. The bodies contain one or more parallel assignments to the local variables. These assignments are specified by first-order logic formulas concatenated by the parallel execution operator (| |). Formally, these formulas are build from the variables in the *write frame* (the subset of the local variables into which executions of the action can write) and their primed versions (references to the values of the variables after the execution of the action). In practice, the primed versions of the variables are identified by always putting them in the left side of the equality operator and the prime is omitted.

The designs in CommUnity interact with each other by means of connecting their private variables with other designs' output variables as well as the synchronization of their public actions. Private variables and actions are not involved in interactions. It is not allowed to synchronize two actions of the same component. The scope of each variable and action name is local to the component where it occurs, so interaction is possible only through explicit attachment of names. Coordination is separate from computation because the name bindings are specified independently of the actions' bodies.

Systems in CommUnity are ultimately built by using superposition morphisms [41]. A CommUnity design may be refined to a program by collapsing the guards to one that lies "between" the guards being refined, meaning that it cannot be weaker than the lower bound nor stronger than the upper one. Also, the assignments of a program may not be under-specified, in other words, they must be deterministic.

To illustrate these concepts we give an example of two designs in Figure 2.1, modified from [26]. These are named `laptop` and `printer`; the intention is to model the production and transmission of documents by the laptop to the printer.

The output variables are denoted by the `out` keyword, the input by `in` and the private ones by `prv`. The `laptop` has two variables, an output variable

```
design laptop
  out outfile : enum(ps, pdf)
  prv saved : bool
  do edit : true, false -> saved := false
     [] save_ps : ~saved -> outfile := ps || saved := true
     [] save_pdf : ~saved -> outfile := pdf || saved := true
     [] send : saved -> skip

design printer
  in infile : enum(ps,pdf)
  prv busy : bool; printfile : enum(ps,pdf)
  do get : ~busy -> printfile := infile || busy := true
     [] prv print : busy -> busy := false
```

Figure 2.1: The `laptop` and `printer` designs

`outfile` of type `enum(ps,pdf)` and a private one `saved` of type `bool`.

The `printer` design has an `infile` mirroring `laptop`'s `outfile` variable and two private variables `busy` to indicate it is printing and cannot receive any more files and `printfile` to copy the file to print into since `infile` can be modified by the environment.

The `do` keyword denotes the start of the actions-specification. Each action has the form:

```
name[writeframe] : safety_guard, progress_guard
                   -> assignment || assignment || ...
```

The part after the arrow (`->`) defines a set of parallel assignments (the parallel execution operator is `||`). The `safety_guard`, `progress_guard` and `writeframe` are as described above.

The `laptop` design has four actions: `edit`, which just sets `saved` to `false` whenever it is selected for execution; `save_ps` which sets the value of `outfile`; and `send`, which does not perform any assignments (denoted by `skip`), but will be needed for synchronization purposes. The actions are separated by `[]` which denotes external choice.

The `printer` design has two actions: `get` which gets a file if it is not busy and `print` which just updates the `busy` status to false.

Although it is not considered in some formal descriptions of CommUnity , we can consider an initial state for the variables of a design by specifying their values using a first-order logic expression. Later extensions to the language define this explicitly.

2.1.2 Connectors

As stated in 2.1.1, interaction in CommUnity is accomplished by explicit name-binding of variables and actions. In the example of Figure 2.1, we could bind `outfile` with `infile` and `send` with `get`. In this way the synchronized action `send/get` can only execute when the file has been `saved` and the printer is not `busy`.

When communication between two designs has a more complex nature, such as in a communication protocol, we define this behavior in a *connector*, thereby separating the communication concerns of the system from the computational ones. In general terms, a connector comprises *glue* and one or more *roles*. The roles are a kind of "formal parameter" of the connector, restricting the components to which the connector can be applied, while the glue is the "body" of the connector, describing how the actions and variables of the roles are to be coordinated. In CommUnity , the glue and roles are specified using CommUnity designs and the glue is connected to each role by name-binding.

Going back to the example in Figure 2.1 we will define a connector that will allow the communication between the `laptop` and the `printer` to be asynchronous. With that goal, it will have a role for the sender, one for the receiver, and the glue will model the behavior of a buffer.

The design for the sender is shown in Figure 2.2. It models the behavior of a producer of files. Notice the use of the non-deterministic assignment `oneof`: the role requires the sending component to have an action to produce a file, but does not constrain the production policy.

The design for the receiver is shown in Figure 2.3. The design for the glue, which we will call `async` is shown in Figure 2.4.

Finally, the connector is formed from these designs and a *configuration* that instantiates them and binds their actions and variables, see Figure 2.5 for a textual representation and Figure 2.6 for a graphical one.

The connector can then be used by refining each role with a design (or component). The formal definition of *refinement* of designs can be found in [8], but it basically consists in a mapping from the role's names to the

```
design sender
  out outfile : enum (ps, pdf)
  prv ready : bool
  do produce : true, false -> ready := true
               || outfile oneof enum(ps,pdf)
     [] send : ready, false -> skip
```

Figure 2.2: The sender design

```
design receiver
  in infile : enum (ps, pdf)
  do receive : true, false -> skip
```

Figure 2.3: The receiver design

```
design async
  in infile : enum (ps, pdf)
  out outfile : enum (ps, pdf)
  prv ready : bool; buffer : list(enum(ps, pdf))
  do put : length(buffer) < 3 -> buffer := buffer:<infile>
     [] prv next : buffer /= nil & ~ready -> ready := true
                   || outfile := head(buffer)
                   || buffer := tail(buffer)
     [] get : ready -> ready := false
```

Figure 2.4: The glue design

```
connector async {
  glue
    design async
  roles
    design sender
    design receiver
  configuration
    s:sender r:receiver a:asynchronous
  attachments {
    a.infile - s.outfile a.put - s.send a.lb - s.ls
    a.outfile - r.infile a.get - r.receive
  }
}
```

Figure 2.5: The configuration (textual)

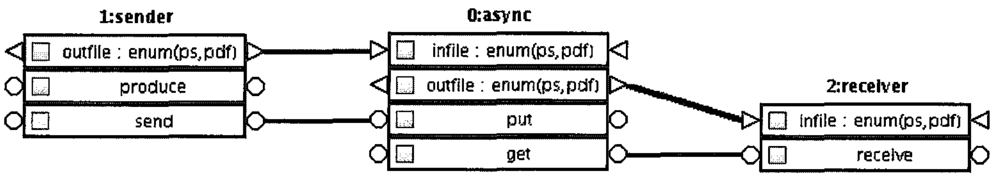


Figure 2.6: The configuration (graphical)

component's names and a translation of terms, such that the behavior is preserved. In this example, the role `sender` is refined by `laptop` and the role `receiver` is refined by `printer`. For example, the refinement of `sender` by `laptop` is given by mapping `produce` to `save_ps` and `save_pdf`, `send` to `send` and `outfile` to `outfile`. As illustrated, an action of a role can be refined my multiple actions of the component.

The modeling of a software architecture is accomplished in CommUnity by the creation of architectural `configurations`. These are created by selecting components and connectors, creating as many instances of each as needed, and interconnecting component instances by direct name bindings or through the application of connector instances.

A configuration has a very precise mathematical semantics, given by a diagram in a category whose objects are the designs and whose morphisms capture a notion of program superposition [5]. Any such categorical diagram can be transformed, by an universal construction called a *colimit*, into a single design that represents the whole (distributed) system.

Intuitively, the variables of the colimit are the union of the variables of all the design instances in the configuration, with shared variables merged into a single one. The resulting merged variable is an output variable only if one of the variables is. The actions of the colimit are obtained by taking at most one action from each design instance and synchronizing them; synchronizations imposed by the configuration must be obeyed, but if an action is "free" then it can co-occur with any other actions from other design instances. Private actions can only co-occur with other private actions due to fairness requirements. Synchronizing actions amounts to taking the union of their bodies. The colimit, which is unique up to renaming of its variables and actions, avoids clashes between equal names of different design instances.

The size of the colimit is bounded by the product of the size of the instances, because it amounts to taking the Cartesian product of the actions, removing combinations that are prevented by the explicit synchronizations imposed by the configuration.

2.1.3 The CommUnity Workbench

The CommUnity Workbench is a (partially) graphical editor to support the creation and editing of CommUnity architectures and of designs, connectors and architectural configurations. Almost all of the work can be performed graphically, except for the definition of the designs, an appropriate choice given the nature of designs' definitions.

The CommUnity Workbench is written in Java and it uses the javacc [15] compiler compiler to deal with the lexical and parsing parts of its work.

It is capable of checking the syntax of the designs[1], calculating the colimit of a configuration, and running a simulation of the colimit.

An interesting question to make about this tool (and the equivalent one fro any other ADL tool, for that matter) is if its architecture is described using the CommUnity ADL. The answer in this case is that it is not.

The program is freely available on the web [7] and is in active development. The source is distributed along with the program; however, it is not clear what license is attached to its use.

2.2 Extensions to CommUnity

In order to give a better idea of the kinds of extensions that our environment should be able to support, we will next present two of the extensions that have been made to CommUnity .

2.2.1 Location-aware CommUnity

In an extension to CommUnity described in [26], modeled systems are made location-aware by adopting an explicit, but designer defined, representation of space. This is accomplished by adding location variables to the designs and attaching them to the regular variables and actions. Additionally, two binary relations over locations need to represent our concept of space. This allows the modeling of different definitions of "space", such as a 2D, discrete representation of space or a three dimensional one using reals to represent each coordinate. The relations are called *touches*, to which two positions in space belong if they are "in touch" with each other, and *reaches*, that holds when one position is reachable from the other.

Communication in CommUnity is achieved via bi-directional sharing of variables and synchronization of actions; hence *touches* must be reflexive and symmetric. For example, we can define two positions to be in touch if the are at most one length unit apart.

[1]the connectors and configurations need no such checking, since they are created by graphical (correctness preserving) actions

```
design laptop
  inloc ll
  out outfile@ll : enum(ps, pdf)
  prv saved@ll : bool
  do edit@ll : true, false -> saved := false
     [] save_ps@ll : ~saved -> outfile := ps || saved := true
     [] save_pdf@ll : ~saved -> outfile := pdf || saved := true
     [] send@ll : saved -> skip

design printer
  inloc lp
  in infile : enum(ps,pdf)
  prv busy@lp : bool; printfile@lp : enum(ps,pdf)
  do get@lp : ~busy -> printfile := infile || busy := true
     [] prv print@lp : busy -> busy := false
```

Figure 2.7: The `laptop` and `printer` designs with location variables

The *reaches* relation must be reflexive and movement of data or code to a new position is possible only when that position is reachable from the current one.

The specification of distribution (through the definition of the coordinate system and *touches* and *reaches* relations) is separate from the specification of computation to allow reuse of components in different topological contexts.

To illustrate these concepts we will rework the designs in Figure 2.1 to include location information.

First, we have to define the *type of location*, which will be the type of our location variables. To keep things simple, we will define a discrete linear space using the type `int` for the *type of location.* The *touches* relation will be defined as `(x-y = 1) | (y-x = 1) | (x = y)` and the *reaches* relation will be defined as `(x < y) | (x = y)`. The new design specifications will look like those in Figure 2.7.

Mobility is explicitly specified by assignments to location variables: all data and code associated to the location variable on the left-hand of the assignment are moved to the position given by the right-hand side expression. In Figure 2.8 we can see an example of a design that modifies location variables, this design is used as the glue of a connector that models the behaviour

```
design follow
  inloc lchased
  outloc lchaser
  do prv move@lchaser : lchaser /= lchased -> lchaser := lchased
```

Figure 2.8: The `follow` design

of a portable printer that moves wherever an associated laptop goes.

Execution of designs is then carried on in a similar fashion to regular CommUnity , with two additional requirement:

1. That the location of each body of the action chosen for execution must be in touch with the locations of the other bodies and of the variables occurring in the body's guards and assignments.
2. Every assignment to a location variable changes its value to a position that is reachable from its current value.

As noted in section 3.1, this extension was incorporated into the CommUnity Workbench and it helped highlight some of the shortcomings of its design.

2.2.2 DynaComm

DynaComm is an extension to CommUnity that adds dynamic reconfiguration and support for design of sub-architectures. It was developed to solve the two most important problems with CommUnity : its lack of support for specifying systems capable of modifying their own architectures at run time (dynamic reconfiguration) and the lack of facilities for specifying components made up of other components (in other words, defining hierarchical system architecture specifications). It is described in [19].

DynaComm modifies the syntax of CommUnity designs and renames them as *components.* It adds to CommUnity's notion of design an *initial condition*, which constrains the values of its local channels when a component instance is created. The CommUnity Workbench already supported these although it did it outside of the design descriptions.

DynaComm also adds the possibility of specifying parameters for the components to facilitate the design and development of systems. These parameters can be values belonging to a type, which can be used in initial

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

Figure 2.9: A DynaComm Component

conditions and in the guards and body of actions. They can also be types, in which case they can be used to set the type of variables at instantiation time.

An example component definition in DynaComm, modified from [19], can be seen in Figure 2.9. It is very similar to the specification of CommUnity designs except for the two new features described above:

- An `init` section, which specifies the initial state of the component using a first-order logic expression.
- Initialization parameters, specified between parentheses after the component name, which facilitate the design and development of systems.

In order to make the description of dynamic systems more convenient, DynaComm adds parameterized actions, which help to specify a family of actions that use this index to identify the member of the family to which reference is being made. The index may also appear in the enabling and progress guards and in the assignment expressions differentiating members of the family. Note that the cardinality of the indexing set must be finite. An action of this kind is called a *schema action* or *indexed action*, and they are specified after the action name:

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

The formal definition (from [19]) is:

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

This extension is needed for the specification of multiple interfaces of the same body of a component, which is essential for dynamic reconfiguration in DynaComm, since reconfiguration actions will use them to differentiate which action corresponds to (is synchronized with) which component of the configuration.

In order to create and delete instances of components, DynaComm defines a component manager M. This component manager can be defined for different component types within a system or at the subsystem level to manage all the instances defined in that subsystem.

DynaComm also defines connector designs, which are components specialized in implementing interactions between components. This allows us to separate the computational aspects of components from their communication logic and allows the same communication pattern to be applied in different contexts. In contrast with CommUnity, DynaComm provides a special syntax for connectors, which simplifies the definition of complex systems in CommUnity and ADLs in general.

A connector in DynaComm has a finite set of roles that can be instantiated with specific components of the system under construction, a glue specification, which describes how the activities of the role instances are to be coordinated, and the connections between the glue and the roles, given by synchronization of channels and actions between them. Roles define the minimum requirements the components instantiating them must meet, and the behaviors to be plugged into the connectors. Each role can refine another already defined component or subsystem, in which the refinement morphism is given by the mapping of the channels and actions between the components or subsystems (refer to section 2.1.2 for more on refinement).

The connector can also have attributes and constraints to enable it to specify a general architectural pattern. An attribute can be private, input or output, as in components. The constraint is a formula in first-order temporal logic, which constrains the architectural evolution of the connector's

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

Figure 2.10: A DynaComm connector

configuration. An example definition of a connector from [19] can be seen in Figure 2.10. The whole system model can be found in [19].

The name of the connector is specified after the `design connector` keywords. The `glue` part specifies the name of the component that will be playing that part. Likewise, we can have zero or more `role` declarations naming the components playing these parts. The `connections` section gives the synchronization of channels and actions between the roles and the glue, for which we can define a "middle" component (a cable) and corresponding regulative superpositions [41] from it to the glue and the role (see section 2.1.1). These connections can specify the actual parameters for designs and actions. Each role can refine another already defined component or subsystem, in which the refinement morphism is given by the mapping of the channels and actions between the components or subsystems.

The other main aspect in which DynaComm improves over CommUnity is the possibility of creating new systems from components. The term it uses for this new system is, a bit confusingly, *subsystem*; it can be thought of as configurations of simpler components, which due to reconfiguration can be dynamically modified. They can be be used to combine the instances of components, connectors and other subsystems. When a subsystem is defined in DynaComm a schema of the subsystem is also defined, which means the instances of this subsystem can be used to construct other subsystems.

An example of a subsystem from [19] can be seen in Figure 2.11. The name of the subsystem is given after the `design subsystem` keywords. Then, the associations section name the components of the subsystem in the `component` section and the `morphisms` section gives the synchronization between the components' channels and actions. The `participants` keyword is followed by a mapping of new names to the designs that will be contributing to the interface of the subsystem. Then, the channels and actions of the subsystem are named and connected to the interface contributors' own. Finally, the `init` section gives the initial state of the subsystem.

2.2.3 CommUnity to SMV

Although not an extension of the Workbench itself, work has been started in automatic translation of CommUnityarchitectural descriptions to SMV modules. The SMV system is a tool for checking finite state systems against specifications in the temporal logic CLT [17]. This was motivated the CommUnity Workbench's current lack of verification of temporal properties of designs, such as verifying that actions execute in the proper order. This is being done by implementing a new lexer and parser for the syntax produced by the CommUnity Workbench and producing SMV modules. It would be desirable that this parsing work could be skipped by relaying on a higher level API for working with CommUnity designs.

2.2.4 Conclusions

In conclusion we can find the following main deficiencies in the current CommUnity Workbench:

- It has a monolithic architecture, unsuited for extendibility.
- Lack of code documentation, making it hard to maintain.
- Built from the ground up, making the code base larger than it needs to be.
- For the same reason it lacks some amenities of more modern graphical editors. For example, its graphical editing capabilities are severely limited.

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

Figure 2.11: A DynaComm subsystem

- It does not integrate well with the environment it is running in. For example, it does not use native file selectors.

Chapter 3

CommUnity Workbench 2

In this section, we will describe the requirements, design criteria and design decisions related to a new CommUnity Workbench, a tool meant to work with CommUnity and related languages with the goal of being easily extensible to facilitate further modification and experimentation with the language. It is only natural to ask if the the new CommUnity Workbench's architecture was described using CommUnity itself. As previously stated, different ADLs are suitable for different projects and, in this case, the style of design and development was not a good fit to CommUnity's characteristics and therefore it was not used to model the Workbench's architecture.

3.1 Rationale

Although the CommUnity Workbench developed by Cristóvão Oliveira et al. is a very useful tool for experimenting and working with CommUnity designs, it has design limitations that prevent it from being an ideal platform for working with extensions of the language. From the source of the CommUnity Workbench, available with the distribution, we can distinguish at least two features that were ostensibly added after the initial implementation of the Workbench. From the analysis of these extensions we can see that the extensions of the Workbench were not as straightforward as they should have been.

The first one was the addition of location variables to the design's variables and actions. This necessitated modifications to the grammar that parses the designs, as well as a number of files that implement different aspects of the CommUnity Workbench functionality. Most worryingly, these changes necessitated modification to the core implementation of the language; in other words, after these modifications, all subsequent designs would need to take these changes into account, even if this new extension does not concern them.

Another change that required modification of the code base was the addition of the ability to generate X-Klaim programs[1]. It was possible to add this modification in a more local way, only a few classes were affected. Still, it would be ideal if the core implementation of the language did not need to be modified at all in order to add new functionality to the editor. Although it may prove impossible to attain this initially, the intention is to fix the original design as needed to enable this goal, instead of violating this principle to implement new functionality and in essence forking the code base.

As discussed in section 1.7, Architecture Description Languages are in constant and rapid evolution. This makes flexible tools that are easy to extend a must if the language is to be useful for research purposes with a reasonable investment in terms of researcher effort. With this in mind, we set out to develop a robust and extensible platform for working with CommUnity and its extensions.

3.2 Design

3.2.1 Design goals

As discussed previously in section 3.1, given the desire to make the workbench a solid base for experimentation with ADL features, extensibility was the main design concern of the new tool. It should be possible to add new syntaxes and semantics without modifying the original code base. It should also be possible to use external tools to process the systems modeled with the tool.

A concern of any long-term software project is maintainability; with this in mind, the design of the new Workbench should allow for as easy as possible maintainability. This implies an easy to understand design and, most importantly, ease of modification.

The design should also have reasonable and scalable performance, so it can work with reasonably sized problems.

[1]X-Klaim is a programming language for object-oriented mobile code [3]

It is also worth mentioning that it is important for the tool to be able to run in the most common desktop environments in existence today. Especially in academic settings. Nowadays, that translates into a recent version of Windows, Max OS X and several different Linux distributions.

3.2.2 Implementation Considerations

Before considering the design of the tool itself it will prove useful to briefly discuss the implementation strategy chosen, as this will direct and restrict the specific design for the new tool.

ArchStudio as a base

Initially, ArchStudio was intended to serve as a base to develop the new CommUnity Workbench on. However, several problems prevented this from being the best option.

ArchStudio 4, the current version of ArchStudio, is based on the code base for ArchStudio 3, which was a stand-alone Java application, which sustained more than 70 releases over 5 years [33].

Figure 3.1 (from [6]) shows ArchStudio's tools and their relationships. The ArchStudio team developed XML Schemas using XML Spy, a proprietary XML Editor. Then they used a tool they developed, Apigen, to generate Data Binding Libraries; these last two tools use Apache Xerces, which is an XML parser and DOM implementation. xArchADT builds on top of the Data Binding Libraries in order to provide an event-oriented interface to ADL documents. Finally, the different tools and editors of ArchStudio, such as ArchEdit (a visual syntax-directed, tree-based editor) in the figure use the xArchADT to perform their duties.

The ArchStudio 4 release moved ArchStudio from being a stand-alone Java application into being a set of Eclipse plug-ins. This allowed the ArchStudio developers to have a more solid and modern base for their editors and tools.

When ArchStudio was selected to develop the new CommUnity Workbench, an initial XML Schema document describing an XML serialization of a design was elaborated. Then, we proceeded to use Apigen [6] to generate the necessary data bindings to build on. Unfortunately, this is where the deficiencies in ArchStudio started to show up.

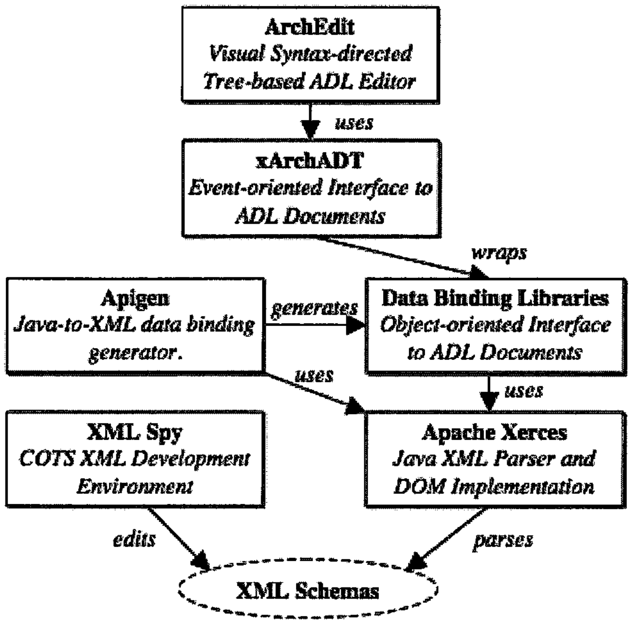

ArchEdit
Visual Syntax-directed Tree-based ADL Editor
uses
xArchADT
Event-oriented Interface to ADL Documents
wraps
Apigen
Java-to-XML data binding generator.
generates
Data Binding Libraries
Object-oriented Interface to ADL Documents
uses
uses
XML Spy
COTS XML Development Environment
Apache Xerces
Java XML Parser and DOM Implementation
edits
parses
XML Schemas

Figure 3.1: ArchStudio Infrastructure tools and their relationships

The first issue encountered during this step was that Apigen was failing without much of an error message while trying to generate bindings for our extension schemas. While this problem was later solved, this process revealed that Apigen is not quite comprehensive in its understanding of XML Schema. While this was not a design goal for Apigen, it does limit its applicability. Additionally, while at the time Apigen was written there were no mature options for generating data bindings based on XML Schemas [6], since that time production quality tools have been developed to that effect [16].

Another problem that was uncovered during this initial effort was the lack of documentation for the ArchStudio 4 editor and related development tools (both in the form of tutorials, guides, references and the like, and code documentation, such as javadoc).

In addition to these problems, while experimenting with ArchStudio 4, several bugs and performance problems were uncovered that, while not being show-stoppers, weakened our confidence in its overall maturity, especially given the existence of, in our opinion, better options that will be discussed next.

Finally, ArchStudio 4 itself does not seem to really be designed for extensibility, as evidenced by the fact that one of the members of the team suggested forking the code base for the implementation of the CommUnity editor.

In conclusion, the perceived benefits of basing our work on ArchStudio would be offset by the difficulties of adapting it to our ends.

The Eclipse Graphical Modeling Framework

While researching alternatives to base our work on, we came across the Eclipse Graphical Modeling Framework (GMF), a generative component and runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF), which is also an Eclipse project. The big advantage that GMF brings to the table is a proven base to develop production quality graphical editors. We will first provide a description of EMF and GEF, on which GMF is based and then proceed to describe the unique characteristics of GMF.

The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) provides code generation facilities for the construction of tools based on structured data models, as well as runtime support for such tools. It provides a meta-model for model building and the tools necessary to make working systems out of the models created with it [34].

EMF uses the word *model* in a similar way in which UML, for example, does. It is a description of a system of structured data. EMF uses XMI (the XML Metadata Interchange XML application) [12] as its canonical form of model definition[2]. However, there are several ways to get from a model description to an XMI model definition:

- Write an XMI document directly, using an XML editor or, macho-style, character by character.
- Export the XMI document from modeling tools such as Rational Rose.
- Annotate Java interfaces with model properties.

[2]As a side note, the EMF meta-model itself is itself an EMF model.

- Use an XML Schema serialization of the model.

Although not all these methods have natively compatible meta-models, EMF provides work-arounds to get around each format's limitations. Generally, an approach suiting the project at hand will be chosen. Regardless of the method chosen to describe the model, once it is specified and in XMI format, the EMF generator will produce a set of Java implementation classes of the model. This provides a fast way to create the code for manipulating the model, avoiding the error-prone process of creating one by hand and simplifying maintenance. Once the code for the classes is generated, it is possible to add variables and methods and subsequent modifications of the model and regeneration of code will respect these changes. Changes to the model do not affect manually-inserted code, except for modifications to features of the model the code depends on. This is accomplished by changing the meta-data embedded in the comments for the code.

The generated code supports model change notification, several kinds of persistence, a framework for model validation, and an efficient reflective application programming interface for manipulating EMF objects generically. Additionally, as we will see below, there are additional useful tools that build on EMF.

In addition to the code generation and runtime support, EMF also provides the EMF.Edit framework. It builds on the core framework to provide support for generating adapter classes that enable viewing and command-based (undoable) editing of a model and can even generate a basic working model editor.

EMF has its roots in the OMG (Object Management Group) MOF (Meta Object Facility). It started as an implementation of the MOF specification, but evolved based on practical use experience. EMF is basically an efficient implementation of a core subset of the MOF API. The MOF-like core meta model in EMF is called Ecore. In Figure 3.2 we show a greatly simplified graphical representation of the Ecore kernel. For more detail refer to [4].

The proposal for the next version of MOF defines a similar subset of the MOF model, called EMOF (Essential MOF) and separates it out. EMF can transparently read and write serializations of EMOF [34].

The Graphical Editing Framework

The Graphical Editing Framework (GEF) allows developers to create a rich graphical editor from an existing data model. GEF uses as its layout and

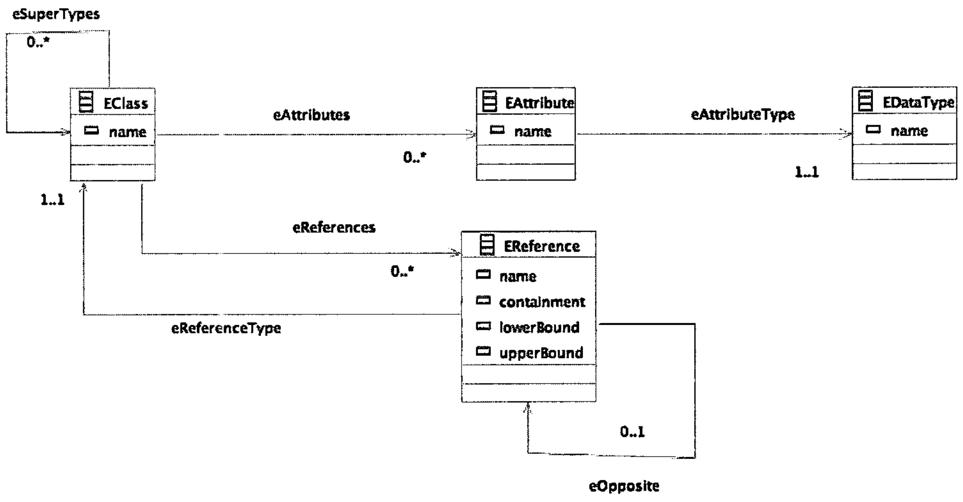


Figure 3.2: The Ecore kernel

rendering toolkit the draw2d plug-in. It builds on it to provide standard graphical editing tools such as selection, creation, connection and marquee; a palette in which to hold these tools; handles for resizing objects and bending connections; graphical and tree viewers; undo/redo support using commands and a command stack; and, most importantly, a controller framework for mapping the business model to a view. This controller framework provides plug-in policies for mapping interactions with the view to changes in the model. Various implementations for showing feedback and adding selection handles, and various request types and tools or actions that send these requests to the controllers.

GEF is application neutral and provides the groundwork to build almost any graphical editing application, such as GUI builders, class diagram editors, state machine editors and even WYSIWYG text editors [35].

Summarizing, using GEF provides a set of standard tools for creating graphical editors and solves many of the common problems encountered while building these tools. This avoids some of the problems encountered while trying to use ArchStudio as a base for our work.

Even without GMF, applications can be built using EMF and GEF since, between the two, they provide all the components of a well-designed graphical editor (a model, a view, and a controller). Although it is possible to create an editor using just GEF and create the model using Plain Old Java Objects (POJOs), using EMF provides the advantages of code generation, built in persistence and tools for manipulating your model. Additionally, by using EMF we avoid many of the pitfalls and bugs we would encounter when developing a model written from scratch.

Putting it all together

GMF is a technology that has its origins in IBM Rational modeling products. It solves some technical challenges (like different command stacks) of using EMF models with GEF. Additionally, it provides a model-oriented development approach to developing a graphical editor, with models describing the editor and the correspondence between its elements and the model the editor is going to work on. It may be viewed as analogous to the way EMF allows us to generate a tree based editor from a model definition; only here there is quite a bit more work involved, since graphical editors allow much more variation and flexibility [36].

When developing a new application using GMF, the first step is usually to create a model using EMF, as discussed previously. Then a graphical definition model is created to represent the elements that will be visible on the canvas. Next, a tooling definition model is created to describe the tools that will be available to the editor's users. Finally, a Mapping definition model, specifying the relations between all the other models, is created. From this, a generator model is created that allows us to modify generation parameters and then generate the graphical editor. Just as with the EMF case, it is possible to modify the generated code to our liking and the tools will respect the modifications when regenerating the code.

In addition to all the architectural benefits that these three frameworks bring to the table, there is extensive documentation for them. Furthermore, there are plenty of tutorials and other resources on the web to help a new developer get started on them. This is not only important for initial development of the project. As developers come and go, and new people get incorporated into the development of the project, it is of paramount importance to have a stable, maintained and well documented code base to help newcomers get up to speed.

In conclusion, for a new project that has no dependencies on an already developed model implementation, it is a very good choice to adopt the GMF framework and develop from there. Although this forces us to use EMF as a meta-model framework, it is sufficiently rich and flexible that this represents more of an advantage than a disadvantage.

3.2.3 The Design Itself

By using EMF and related technologies, the design of the tool becomes the design of the models describing the domain models, the graphical editor and the mapping between these two. Consequently, in what follows these models are described.

To develop the new CommUnity Workbench, we first defined a domain model for the CommUnity designs and related concepts using EMF. From there, we defined all the required GMF models and produced a graphical editor for the model. Although at this stage the editor is quite "rough around the edges" it was considered a better investment of time in the long run to build a solid base for future development than to have a shaky base with a pretty front-end.

The development of a GMF-based editor necessarily starts with the design of the model, since this is the base from which the editor's code will start.

At first, we started the model specification using the Java interfaces plus annotations mentioned above. Unfortunately, Java has been evolving rapidly as of late, and although EMF has been evolving along with it, the documentation has not been consistently updated[3]. This influenced us to move the main modeling method to XML Schema; using the XML Schema generating capabilities of EMF saved a lot of time during this migration. The designs were modeled down to the point of logic expressions, that is, logic expressions were not given structure; this was decided given that the modeling of more detail inside these expressions would have produced a bigger, more verbose model without any immediate benefit. Additionally, as exemplified in the example extensions reviewed in section 3.1, the expression language has so far remained the same, so the individual modeling of their components would not have added or detracted from the ease of their implementations. In any case, it is possible to extend the model to do fine-grain expression modeling without disturbing existing users of the model.

Next, we will give an illustration of the model divided into three parts, to simplify the explanation.

First off, we will describe the part of the model describing designs. A simplified graphical representation can be seen in Figure 3.3. The XML representation of the Ecore model for the designs and the rest of the domain models discussed below can be found in appendix A.

The `Design` model has three references to `VarLists`, list of variables, one for each kind of variable (private, in and out). These variable lists aggregate `Variables`, which have a type. The design also incorporates an `ActionList`. The action list contains the `Actions`, these have a boolean attribute indicating if they are private or not, an enabling guard, and a progress guard. Actions also have a series of `Assignments` which have a left side, a right side and a type, which is of the type `AssignmentType`, an enumeration of the two strings := and `oneof`. We also model an `ActionCable` and a `VariableCable` for action and variable synchronization, respectively.

All the elements that need to have a `name` attribute inherit from the `CNamedElement`, which defines only this attribute.

Finally, a `Design` can have an `Invariant`, establishing the properties that the state of the `Design` must meet.

Next, we present a simplified model for connectors, as seen in Figure 3.4.

[3]A new edition of the EMF book should be out before the end of the year; this will be a big help for current and future developers of the editor.

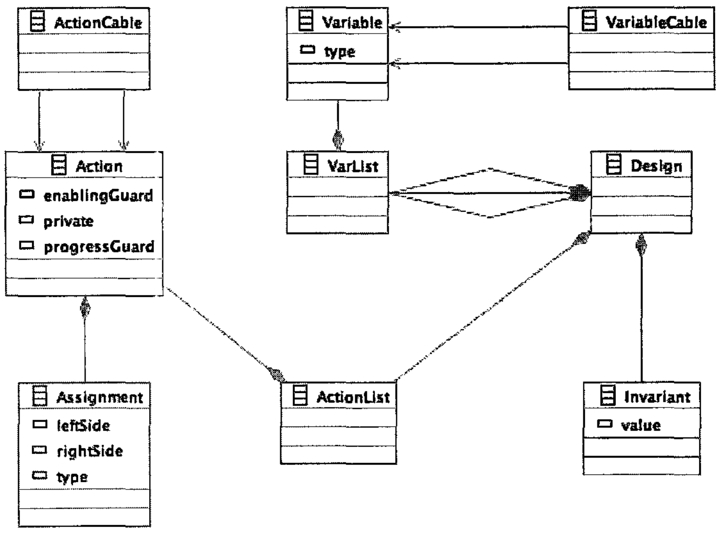


Figure 3.3: The Design Model

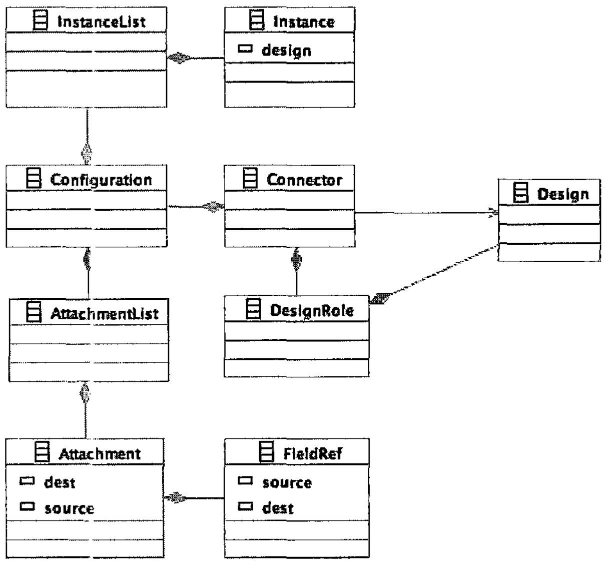


Figure 3.4: The Connector Model

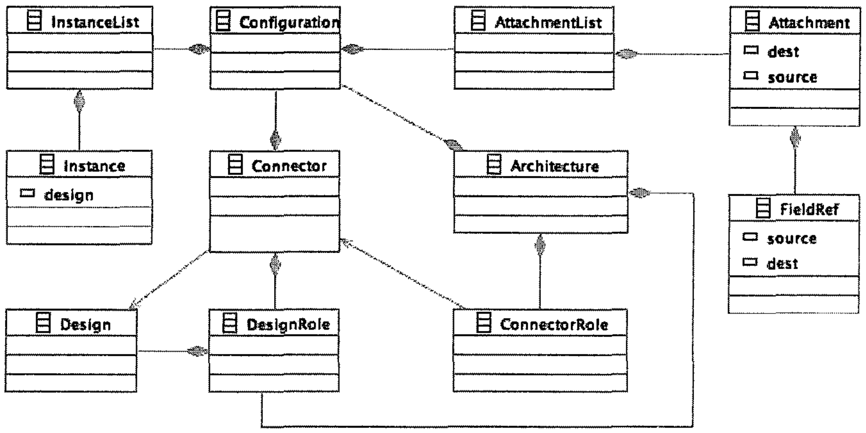


Figure 3.5: The Architecture Model

`Connectors` can also be named, so they inherit from `CNamedElement`, they have a `Design` that acts as the `glue`, and they also have several `DesignRoles` which should have a name and refer back to a `Design`.

They also have a `Configuration`, which has an `InstanceList` for enumerating the `Instances`, which have an attribute name via `CNamedElement` to give an instance name to the role instances (not shown in the diagram). The `Configuration` also has an `AttachmentList`. This defines an `Attachment` for every two designs that are to be connected and each of them can have one or more `FieldRefs` to define the way the variables and actions will connect to each other.

Finally, we describe the part of the model that deals with architectures. A graphical representation can be found in Figure 3.5.

The `Architecture` can have zero or more `ConnectorRoles`, which inherit from `CNamedElement` the `name` attribute. They each have a reference to a `Connector`, which themselves refer back to a `Design`.

Additionally, an Architecture can have `DesignRoles`, which also inherits from `CNamedElement` (not shown in the diagram) and, again, they refer back to the `Designs`. `Configuration` has `InstanceLists`, that name instances of `Designs`. They also have `AttachmentLists` which have attachments between roles of the connectors and designs and specify the refinements between them via `FieldRefs`.

This completes the model for an architecture. With this model created, EMF is capable of producing a tree-based editor for the model. There is also a generator model that allows the modification of several characteristics of the generated editor. Additionally, the generated editor code can be modified until we get exactly what we want. We can see in Figure 3.6 an example of this editor showing part of the printing architecture from section 2.1.1.

After we have a model for our application, we can model the graphical editor with EMF, defining the graphical representations of the elements, the tools used for creating them and so forth and EMF will generate a generator model for a graphical editor. This editor, in a way analogous to the way in which we can modify the EMF editor generator model mentioned previously, can be modified to alter the behavior of the generated editor. A tree representation for part of this model can be seen in Figure 3.7.

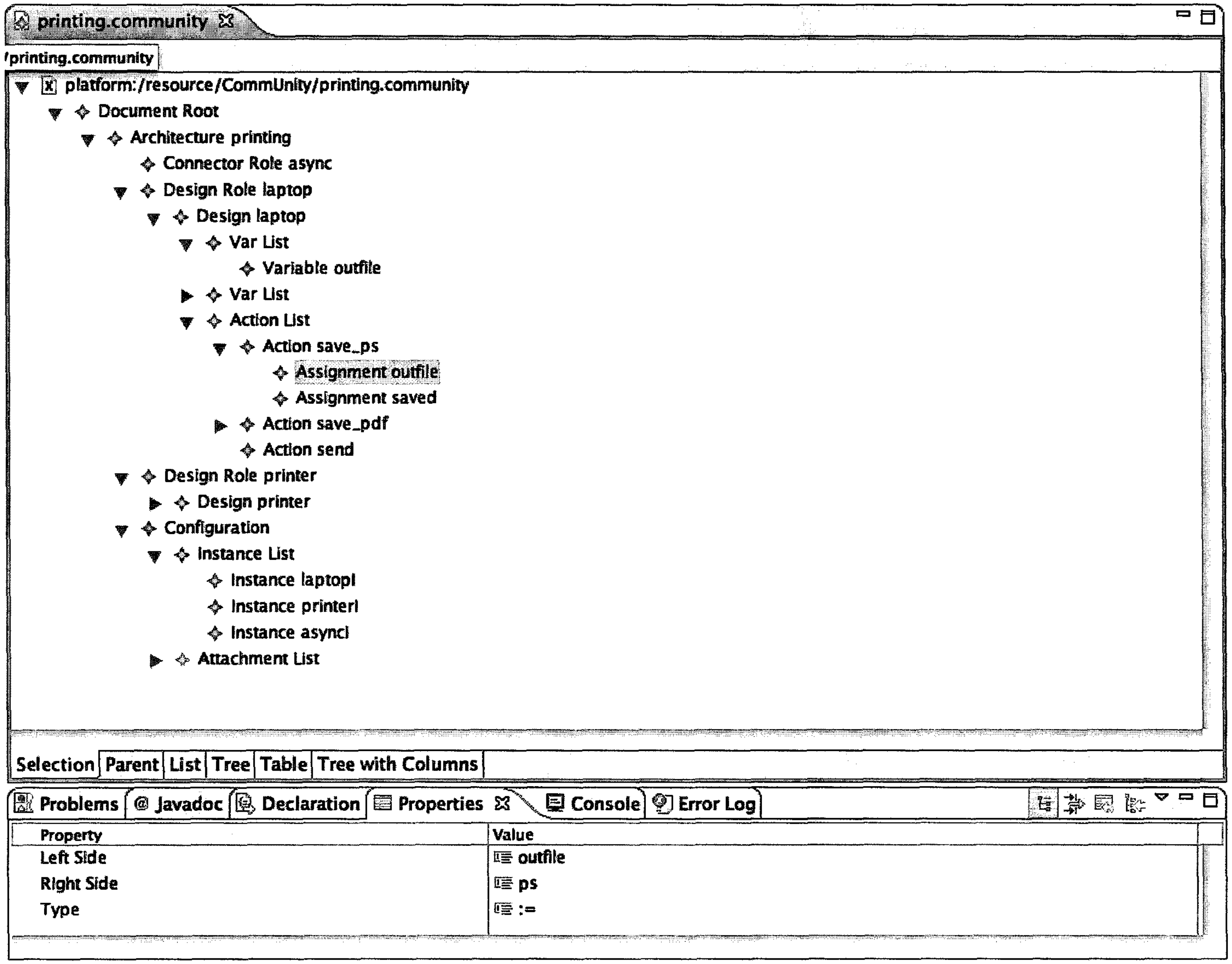
The graphical model consists (with some simplifications) of a `Gallery` of `Figure Descriptors`, which contain `Figures`, which can contain `Labels`, a collection of `Nodes`, `Connections`, `Compartments` and `Diagram Labels`. The basic idea is that the `Figures` describe the graphical representation of the `Nodes`, which are going to be linked with the domain elements in the mapping model.

There is also a simpler model for the tool palette that goes with the diagram editor. It consists of `ToolGroups` which contain the different `Creation Tools`. Complete documentation for this model can be found at the GMF web site [36].

Next, it is necessary to create the model mapping the domain model to the graphical editor model. The tree representation for part of this model can be seen in Figure 3.8.

The purpose of this model is to link which elements of the domain model will correspond with which features of the graphical editor. It specifies what the effects of the actions performed there will have in the model domain. For example, at the bottom of Figure 3.8 we can see how the `ActionCable Connection` from the graphical editor model will represent the `ActionCable` element of the domain model, with the `source` and `dest` references of this element being set to the source and destination of the `Connection`. This will translate in the graphical editor to setting the `source` and `dest` references of the `ActionCable` element in the model to be set to the source and destination of a graphical arrow drawn in the screen.

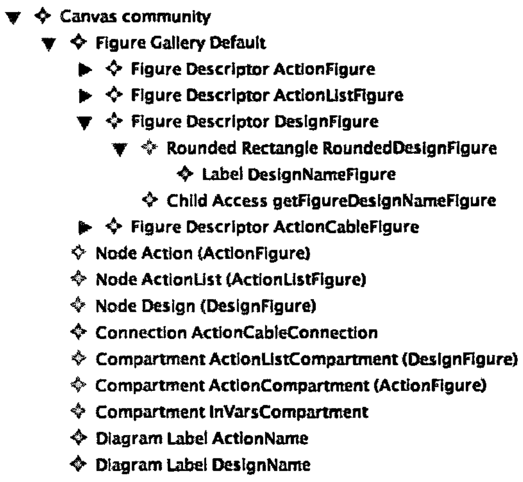
From this model, a model is automatically generated from which the final product will be generated. This model describes in detail the graphical editor that will be generated and thus allows customizing how it is created. In our case there was no need to modify it, although further refinement of the editor



```
platform:/resource/CommUnity/printing.community
  Document Root
    Architecture printing
      Connector Role async
      Design Role laptop
        Design laptop
          Var List
            Variable outfile
          Var List
          Action List
            Action save_ps
              Assignment outfile
              Assignment saved
            Action save_pdf
            Action send
      Design Role printer
        Design printer
      Configuration
        Instance List
          Instance laptopI
          Instance printerI
          Instance asyncI
        Attachment List
```

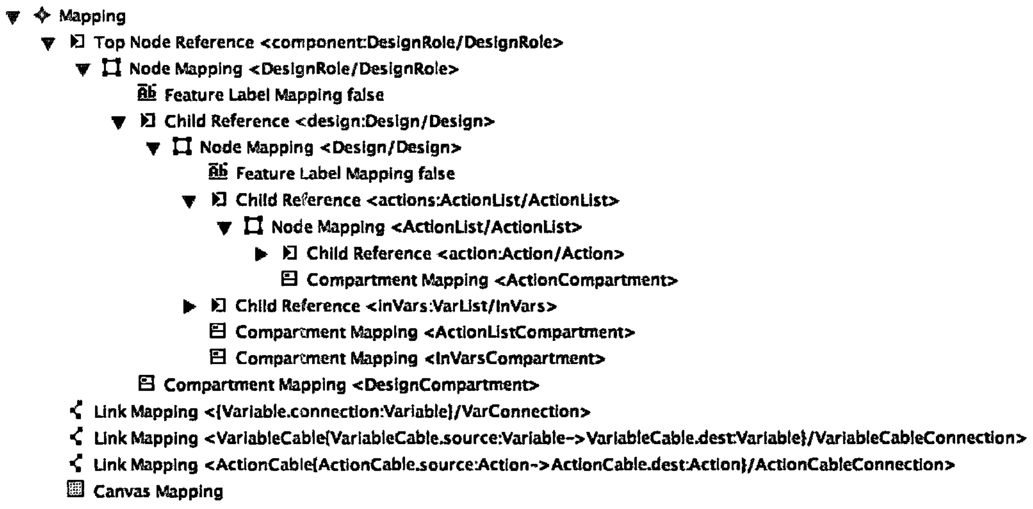
Property	Value
Left Side	outfile
Right Side	ps
Type	:=

Figure 3.6: An Architecture Editor



```
▼ Canvas community
  ▼ Figure Gallery Default
    ▶ Figure Descriptor ActionFigure
    ▶ Figure Descriptor ActionListFigure
    ▼ Figure Descriptor DesignFigure
      ▼ Rounded Rectangle RoundedDesignFigure
          Label DesignNameFigure
        Child Access getFigureDesignNameFigure
    ▶ Figure Descriptor ActionCableFigure
    Node Action (ActionFigure)
    Node ActionList (ActionListFigure)
    Node Design (DesignFigure)
    Connection ActionCableConnection
    Compartment ActionListCompartment (DesignFigure)
    Compartment ActionCompartment (ActionFigure)
    Compartment InVarsCompartment
    Diagram Label ActionName
    Diagram Label DesignName
```

Figure 3.7: The COMMUNITY Graphical Editor Model



```
▼ Mapping
  ▼ Top Node Reference <component:DesignRole/DesignRole>
    ▼ Node Mapping <DesignRole/DesignRole>
        Feature Label Mapping false
      ▼ Child Reference <design:Design/Design>
        ▼ Node Mapping <Design/Design>
            Feature Label Mapping false
          ▼ Child Reference <actions:ActionList/ActionList>
            ▼ Node Mapping <ActionList/ActionList>
              ▶ Child Reference <action:Action/Action>
                Compartment Mapping <ActionCompartment>
          ▶ Child Reference <inVars:VarList/InVars>
            Compartment Mapping <ActionListCompartment>
            Compartment Mapping <InVarsCompartment>
        Compartment Mapping <DesignCompartment>
  Link Mapping <{Variable.connection:Variable}/VarConnection>
  Link Mapping <VariableCable{VariableCable.source:Variable->VariableCable.dest:Variable}/VariableCableConnection>
  Link Mapping <ActionCable{ActionCable.source:Action->ActionCable.dest:Action}/ActionCableConnection>
  Canvas Mapping
```

Figure 3.8: Mapping model

may make it necessary.

A screenshot of a sample editing session with the graphical editor can be seen in Figure 3.9.

3.2.4 Comparison with CommUnity Workbench

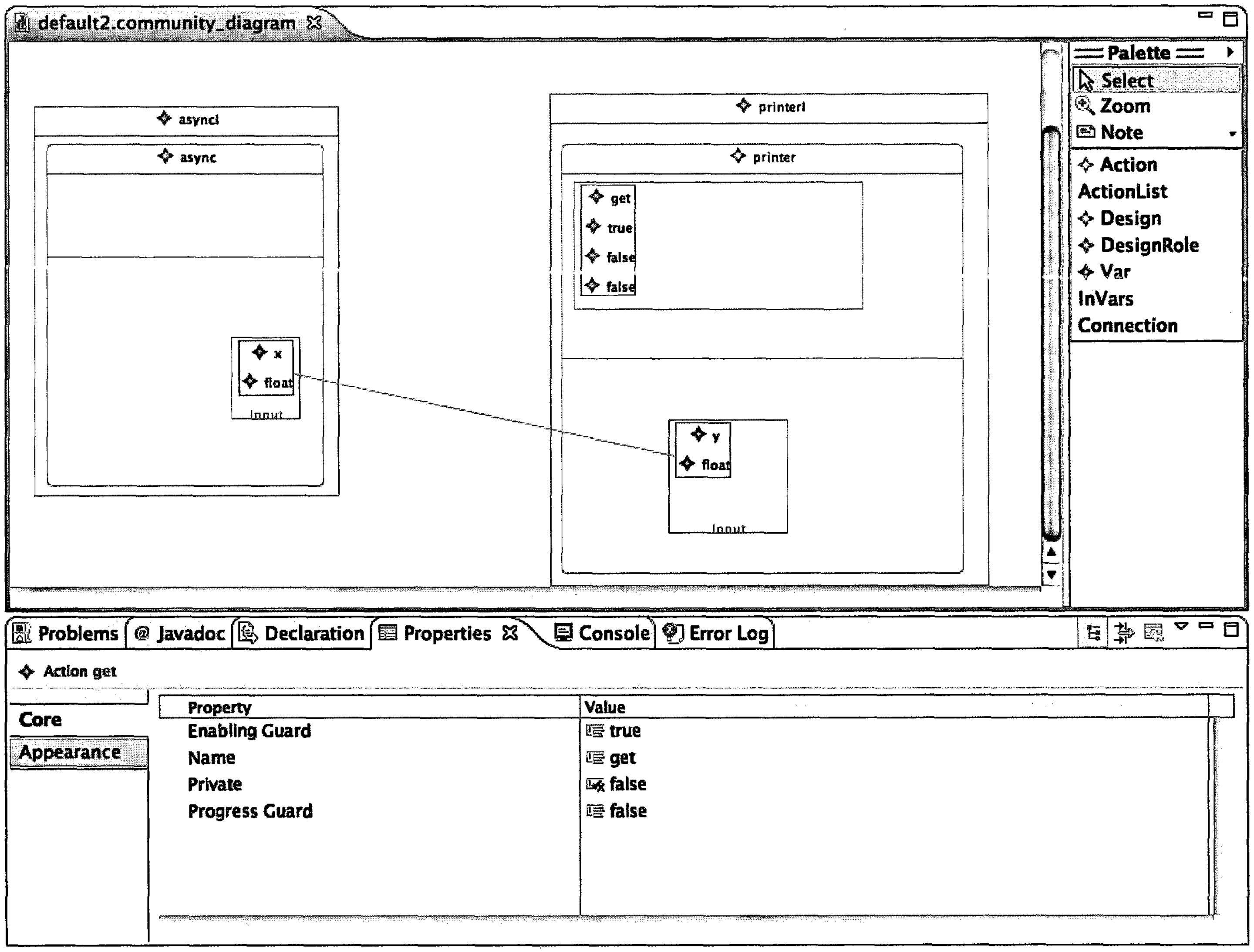
Although the COMMUNITY Workbench was a good start for working and experimenting with COMMUNITY designs, it proved to be too limiting when it came to extending the language. We believe that with the foundations established with the new COMMUNITY Workbench, it will be possible to implement extensions to the COMMUNITY language in a more modular way and without having to modify the core implementation.

This has been accomplished through the use of solid and proven frameworks from well established development communities. This has the advantage of providing a solid foundation for our work, one that we know will be maintained and kept relevant as the technologies evolve and mature. It also allows us to make use of the present and future ecosystem of tools developed around Eclipse, which will allow us to enhance the user experience in a way which would not be possible by developing a stand-alone application, especially considering that the resources available for its development will probably fluctuate with time.

More specifically, extensibility will be allowed by creating extensions to the base domain model by taking advantage of EMF extensibility features. In particular, the extensions may be created by extending the XML Schema description of COMMUNITY; thanks to EMF's design this will allow for the reuse of the common code between our original implementation and the extension to COMMUNITY. Furthermore, the work created for the graphical editor for COMMUNITY architectural descriptions can be extended in the same way (by extending the models created for the COMMUNITY editor).

In the old workbench, in contrast, each extension had to be approached in an ad-hoc way, and it usually involved modification of the existing code and/or reimplementation of existing features.

On the other hand, our editor lacks some characteristics of the original Workbench; in particular, there is no syntax checking of the expressions used in an action's assignments. It is also not possible yet to get the colimit of an architecture and, of course, it is not possible to "execute" the design. It also does not implement the export of COMMUNITY designs to X-Klaim, although it is unclear if this feature is in use anymore. It should be noted



Property	Value
Enabling Guard	true
Name	get
Private	false
Progress Guard	false

Figure 3.9: The COMMUNITYGraphical Editor

that the design of the tool allows for a clean incorporation of these facilities.

Also of note is the fact that the design's representation in-file is different for the two tools. With the old Workbench using a mixture of the traditional CommUnity design syntax for the designs themselves and an ad-hoc syntax for the rest of the architectural constructs. Our tool uses a more uniform XML syntax for all constructs, with the disadvantage that it would be harder to modify by hand because of its verbosity. It is nevertheless hoped that this editing will not be generally necessary since graphical editing tools are provided. Furthermore, it would be possible to develop text-driven editors that may render the new format using the old syntax and allow manipulation in this form.

3.2.5 Extending the CommUnity Model

Basing our basic CommUnityModel in well-defined standards such as XML Schema and Ecore provides us with mechanisms for the easy extensibility of these models to add new constructs to the language.

Additionally, using EMF and related technologies allows us to also cleanly extend the implementation of said model to support the new characteristics of the extension. As an example of the extensibility possible we will describe a model created to represent DynaComm (section 2.2.2), reusing what was already built for CommUnity. In Figure 3.10 we can see a graphical representation of the model created for representing the language described in section 2.2.2.

We added the necessary constructs to represent new DynaComm elements such as `Subsystem` and `Connector`. These elements inherit from the `Design` element of CommUnity's model, so that model information common to them and the core derived from that information is inherited in the DynaComm model without repetition.

Similarly to the way it was done with CommUnity, it is straightforward to generate a tree editor for working with DynaComm models and, with the creation of the corresponding models, it is possible to also create a graphical (boxes and arrows) editor for the language.

Additionally, thanks to the implementation generated by EMF and its dynamic characteristics, it is possible to programmatically load (deserialize), create and manipulate, and save (serialize) instances of the CommUnity Model and its extensions to perform transformations and analysis on them.

To further test the extendibility of our work an extension to edit location

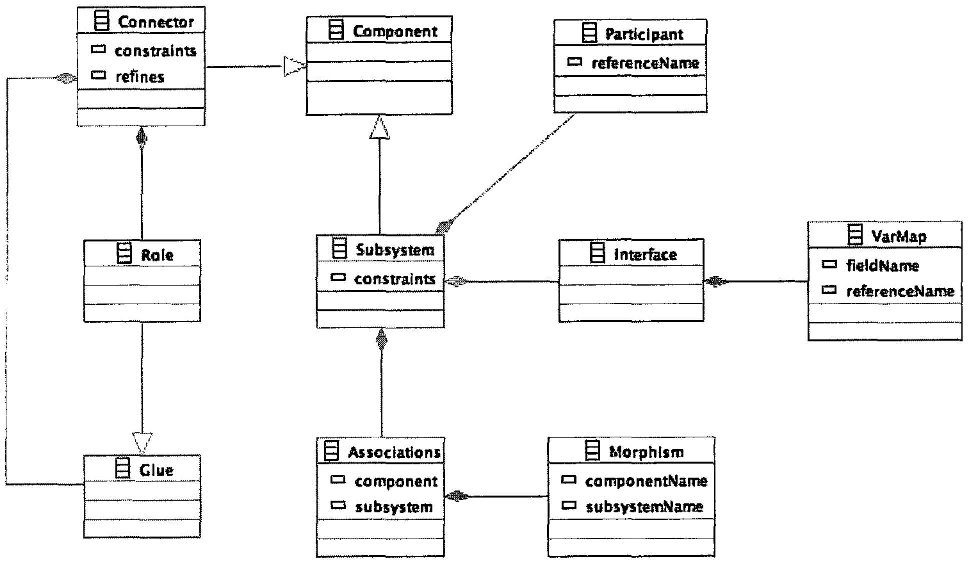


Figure 3.10: DynaComm Model

aware CommUnity designs was developed. The model for this extension is depicted in Figure 3.11.

We created two new types: `LocAction` and `LocVar` that inherit from `Action` and `Variable`, respectively and add a `location` reference that is of type `Variable`.

Additionally, we added a new `LocArchitecture`, which inherits from `Architecture` and adds a `location` reference, this reference is of type `Location`, which has `reaches`, `touches` and `type` attributes.

This new model allows us to model all of Location Aware CommUnity properties.

Finally a model uniting the DynaComm and location aware CommUnity models was created. A graphical representation of this model can be seen in Figure 3.12. It is simply a new type `LocSubsystem` that inherits both from `Subsystem` and `LocArchitecture`; in this way, it has the references and attributes of both systems. This last system was not created from an XML Schema, since XML Schema does not allow for multiple inheritance; instead, an Ecore model was created directly. EMF takes care of things on the Java side by generating the necessary (repeated) code, since Java does not support multiple inheritance either.

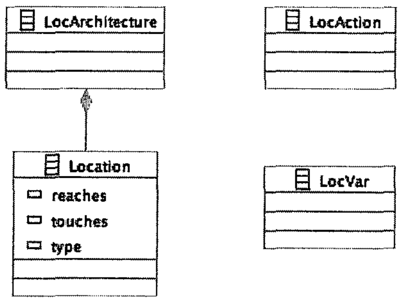


Figure 3.11: Location aware COMMUNITY Model

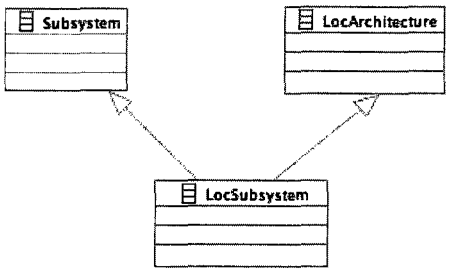


Figure 3.12: DynaComm plus Location aware COMMUNITY Model

This last example illustrates the extensibility of our approach. As before, the new model can be modified independently of the others (as long as the modifications do not affect the references from this model to the others) and all the code for working with the models this new model builds upon is reused for the new one.

Chapter 4

Conclusions and Further Work

This chapter contains a brief review of the work contained in this thesis. Additionally, it discusses some possible future directions that additional work may take.

4.1 Review of the work

We have presented the motivation behind Architecture Description Languages (ADLs) and the tools necessary to work with them. Furthermore, we examined some of these tools and the possibility of extending them for our own purposes.

We also gave a brief explanation of the CommUnity ADL and the ways in which it has been extended. We also discussed how this extensions were implemented using the existing CommUnity Workbench and the shortcomings it has for this purpose.

Finally, we presented the design of the new system and the foundations that give it its extensibility. This was possible thanks to the use of technologies such as EMF and GMF. The use of these frameworks will allow for continued development of the Workbench and a solid base for its extensibility. Although initially the need to learn about the metamodel language and its application may seem like a cumbersome requirement, the time saved by the code generation features and the maintainability, flexibility and extensibility of the resulting system makes it a worthwhile effort. Additionally the use of a model-driven development approach allows for the generation of code, allowing both a faster development and easier maintainability. The experience of trying to extend ArchStudio made us realize that it is wise to base our own efforts on a widely used platform with a proven record-track of extensibility and flexibility.

This lead us to the implementation of a new editor for CommUnity designs. This editor is rooted in a well defined model for the language and uses Eclipse and EMF to have a more solid and extensible base. The beginnings of a graphical, boxes-and-arrows editor was produced to show the feasibility of the task and also an extension of the CommUnity model to the DynaComm one was produced, from which the corresponding tree-based and boxes-and-arrows editors may be developed. Similarly, a model for Location Aware CommUnity was developed, and a final one combining this two models is used to illustrate the extensibility capabilities of our approach.

Although not all the functionality of the old CommUnity Workbench has been implemented, thanks to the design and development strategy adopted, it is our belief that the work performed will allow for an easier implementation of these capabilities and more as extensions. Furthermore, our work will form a solid base in which future development of tools for CommUnity's extensions may be based.

4.2 Future Work

As future work, we can suggest the development of the remaining missing functionality of the old CommUnity Workbench (where this functionality is still deemed important and useful). The development of these extensions, in addition to providing valuable tools, may reveal useful refactorings of our models, making the new workbench all the more useful and reliable. This refactoring will be facilitated by the model-driven approach adopted for the development, since this would not involve the rewrite and careful editing of large and widely dispersed snippets of code, as is often the case with refactoring of large code bases.

As a larger and harder project, it may be useful to reimplement other ADLs such as xArch and ACME using the approach adopted for this work, namely, using a model-driven development strategy. In particular, it might prove fruitful for the developers of tools such as ACME Studio and ArchStudio to adopt GMF as a basis for their tool development, as this may simplify maintenance work and enhance usability.

Appendix A

Ecore models for CommUnity and friends

Note that some formatting was necessary to make the lines short enough to fit the page. As a result some strings (attributes' values) expand several lines when they should be all in the same line

A.1 CommUnity Model

```
<?xml version="1.0" encoding="utf-8"?>
<ecore:EPackage xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
name="community"
nsURI="http://www.cas.mcmaster.ca/ADL/2007/CommUnity"
nsPrefix="community">
  <eClassifiers xsi:type="ecore:EClass" name="Action"
  eSuperTypes="#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Action" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="assignment" lowerBound="1" upperBound="-1"
    eType="#//Assignment" containment="true"
    resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="assignment" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="enablingGuard"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
```

```
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="enablingGuard" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="private"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//Boolean"
    unsettable="true">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="private" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="progressGuard"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="progressGuard" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ActionCable"
  eSuperTypes="#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="ActionCable" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="source"
    lowerBound="1" eType="#//Action">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="source" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dest"
    lowerBound="1" eType="#//Action">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="dest" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="ActionList">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="ActionList" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="action"
    lowerBound="1" upperBound="-1" eType="#//Action"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="action" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
```

```
<eClassifiers xsi:type="ecore:EClass" name="Architecture"
eSuperTypes="#//CNamedElement">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="Architecture" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="connector" upperBound="-1" eType="#//ConnectorRole"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="connector" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="component" upperBound="-1" eType="#//DesignRole"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="component" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="configuration" lowerBound="1" eType="#//Configuration"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="configuration" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="variableCables" upperBound="-1" eType="#//VariableCable"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="variableCables" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="actionCables" upperBound="-1" eType="#//ActionCable"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="actionCables" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Assignment">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="Assignment" />
    <details key="kind" value="empty" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute"
  name="leftSide"
  eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="attribute" />
```

```
        <details key="name" value="leftSide" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="rightSide"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="rightSide" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
    eType="#//AssignmentType" defaultValueLiteral=":="
    unsettable="true">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="type" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="AssignmentType">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="AssignmentType" />
    </eAnnotations>
    <eLiterals name="_" literal=":=" />
    <eLiterals name="oneof" value="1" />
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EDataType"
  name="AssignmentTypeObject"
  instanceClassName="org.eclipse.emf.common.util.Enumerator">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="AssignmentType:Object" />
      <details key="baseType" value="AssignmentType" />
    </eAnnotations>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Attachment">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Attachment" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="fields"
    lowerBound="1" upperBound="-1" eType="#//FieldRef"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="fields" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dest"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="dest" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="source"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="attribute" />
      <details key="name" value="source" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="AttachmentList">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="AttachmentList" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="attachment" lowerBound="1" upperBound="-1"
  eType="#//Attachment" containment="true"
  resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="attachment" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="CNamedElement"
abstract="true">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="CNamedElement" />
    <details key="kind" value="empty" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EAttribute" name="name"
  eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="attribute" />
      <details key="name" value="name" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Configuration">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="Configuration" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="instances" lowerBound="1" eType="#//InstanceList"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="instances" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="attachments" lowerBound="1" eType="#//AttachmentList"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="attachments" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
```

```
<eClassifiers xsi:type="ecore:EClass" name="Connector"
eSuperTypes="#//CNamedElement">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="Connector" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="role"
  lowerBound="1" upperBound="-1" eType="#//DesignRole"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="role" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="configuration" lowerBound="1" eType="#//Configuration"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="configuration" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference" name="glue"
  lowerBound="1" eType="#//Design">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="glue" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ConnectorRole"
eSuperTypes="#//CNamedElement">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="ConnectorRole" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="connector" lowerBound="1" eType="#//Connector">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="connector" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Design"
eSuperTypes="#//CNamedElement">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="Design" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="inVars"
  eType="#//VarList" containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="inVars" />
    </eAnnotations>
  </eStructuralFeatures>
```

```
    <eStructuralFeatures xsi:type="ecore:EReference" name="outVars"
    eType="#//VarList" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="outVars" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference" name="prvVars"
    eType="#//VarList" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="prvVars" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="invariant" eType="#//Invariant" containment="true"
    resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="invariant" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference" name="actions"
    eType="#//ActionList" containment="true"
    resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="actions" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DesignRole"
  eSuperTypes="#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="DesignRole" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="design"
    lowerBound="1" eType="#//Design" containment="true"
    resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="design" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DocumentRoot">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="" />
      <details key="kind" value="mixed" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mixed"
    unique="false" upperBound="-1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFeatureMapEntry">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="elementWildcard" />
        <details key="name" value=":mixed" />
  </eAnnotations>
</eStructuralFeatures>
<eStructuralFeatures xsi:type="ecore:EReference"
name="xMLNSPrefixMap" upperBound="-1"
eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
transient="true" containment="true" resolveProxies="false">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="kind" value="attribute" />
    <details key="name" value="xmlns:prefix" />
  </eAnnotations>
</eStructuralFeatures>
<eStructuralFeatures xsi:type="ecore:EReference"
name="xSISchemaLocation" upperBound="-1"
eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
transient="true" containment="true" resolveProxies="false">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="kind" value="attribute" />
    <details key="name" value="xsi:schemaLocation" />
  </eAnnotations>
</eStructuralFeatures>
<eStructuralFeatures xsi:type="ecore:EReference"
name="actionCable" upperBound="-2" eType="#//ActionCable"
volatile="true" transient="true" derived="true"
containment="true" resolveProxies="false">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="kind" value="element" />
    <details key="name" value="actionCable" />
    <details key="namespace" value="##targetNamespace" />
  </eAnnotations>
</eStructuralFeatures>
<eStructuralFeatures xsi:type="ecore:EReference"
name="architecture" upperBound="-2" eType="#//Architecture"
volatile="true" transient="true" derived="true"
containment="true" resolveProxies="false">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="kind" value="element" />
    <details key="name" value="architecture" />
    <details key="namespace" value="##targetNamespace" />
  </eAnnotations>
</eStructuralFeatures>
<eStructuralFeatures xsi:type="ecore:EReference"
name="connector" upperBound="-2" eType="#//Connector"
volatile="true" transient="true" derived="true"
containment="true" resolveProxies="false">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="kind" value="element" />
    <details key="name" value="connector" />
    <details key="namespace" value="##targetNamespace" />
  </eAnnotations>
</eStructuralFeatures>
<eStructuralFeatures xsi:type="ecore:EReference" name="design"
upperBound="-2" eType="#//Design" volatile="true"
transient="true" derived="true" containment="true"
resolveProxies="false">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="kind" value="element" />
    <details key="name" value="design" />
```

```
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="variableCable" upperBound="-2" eType="#//VariableCable"
    volatile="true" transient="true" derived="true"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="variableCable" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="FieldRef">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="FieldRef" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="source"
    lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="source" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="dest"
    lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="dest" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Instance"
  eSuperTypes="#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Instance" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="design"
    lowerBound="1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="design" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="InstanceList">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="InstanceList" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="instance" lowerBound="1" upperBound="-1"
    eType="#//Instance" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="instance" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Invariant">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Invariant" />
      <details key="kind" value="empty" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="value"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="value" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Variable"
  eSuperTypes="#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Variable" />
      <details key="kind" value="empty" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="connection" upperBound="-1" eType="#//Variable">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="connection" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="type" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="VariableCable"
  eSuperTypes="#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="VariableCable" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="source"
    lowerBound="1" eType="#//Variable">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="source" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference" name="dest"
    lowerBound="1" eType="#//Variable">
```

```
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="dest" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="VarList">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="VarList" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="variable" lowerBound="1" upperBound="-1"
    eType="#//Variable" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="variable" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
</ecore:EPackage>
```

A.2 The DynaComm Model

```
<?xml version="1.0" encoding="utf-8"?>
<ecore:EPackage xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="dynacomm"
nsURI="http://www.example.org/dynacomm" nsPrefix="dynacomm">
  <eClassifiers xsi:type="ecore:EClass" name="Associations"
  eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Associations" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="component" unique="false" upperBound="-1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="component" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="subsystem" unique="false" lowerBound="1" upperBound="-1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="subsystem" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="morphism" lowerBound="1" eType="#//Morphism"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="morphism" />
      <details key="namespace" value="##targetNamespace" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Component"
eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//Design">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="Component" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="parameters" lowerBound="1"
  eType="ecore:EClass ../../ca.mcmaster.cas.community/model/community.ecore#//VarList"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="parameters" />
      <details key="namespace" value="##targetNamespace" />
    </eAnnotations>
  </eStructuralFeatures>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="Connector"
eSuperTypes="#//Component">
  <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
    <details key="name" value="Connector" />
    <details key="kind" value="elementOnly" />
  </eAnnotations>
  <eStructuralFeatures xsi:type="ecore:EReference" name="glue"
  lowerBound="1" eType="#//Glue" containment="true"
  resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="glue" />
      <details key="namespace" value="##targetNamespace" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference" name="role"
  lowerBound="1" upperBound="-1" eType="#//Role"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="role" />
      <details key="namespace" value="##targetNamespace" />
    </eAnnotations>
  </eStructuralFeatures>
  <eStructuralFeatures xsi:type="ecore:EReference"
  name="connections" lowerBound="1"
  eType="ecore:EClass ../../ca.mcmaster.cas.community/model/community.ecore#//Configuration"
  containment="true" resolveProxies="false">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="kind" value="element" />
      <details key="name" value="connections" />
      <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="constraints"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="constraints" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="refines"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="refines" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="DocumentRoot">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="" />
      <details key="kind" value="mixed" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mixed"
    unique="false" upperBound="-1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFeatureMapEntry">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="elementWildcard" />
        <details key="name" value=":mixed" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="xMLNSPrefixMap" upperBound="-1"
    eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
    transient="true" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="xmlns:prefix" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="xSISchemaLocation" upperBound="-1"
    eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
    transient="true" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="xsi:schemaLocation" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="subsystem" upperBound="-2" eType="#//Subsystem"
    volatile="true" transient="true" derived="true"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="subsystem" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Glue"
  eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Glue" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="parameters" lowerBound="1"
    eType="ecore:EClass ../../ca.mcmaster.cas.community/model/community.ecore#//VarList"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="parameters" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="connections" lowerBound="1"
    eType="ecore:EClass ../../ca.mcmaster.cas.community/model/community.ecore#//Configuration"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="connections" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Interface">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Interface" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference" name="varMap"
    lowerBound="1" eType="#//VarMap" containment="true"
    resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="varMap" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Morphism">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Morphism" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="connections" lowerBound="1"
    eType="ecore:EClass ../../ca.mcmaster.cas.community/model/community.ecore#//Configuration"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="connections" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="componentName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//NCName">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="componentName" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="subsystemName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//NCName">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="subsystemName" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Participant"
  eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Participant" />
      <details key="kind" value="empty" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="referenceName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="referenceName" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Role"
  eSuperTypes="#//Glue">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Role" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Subsystem"
  eSuperTypes="#//Component">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Subsystem" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="associations" lowerBound="1" eType="#//Associations"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="associations" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="participants" lowerBound="1" upperBound="-1"
    eType="#//Participant" containment="true"
    resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="participants" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="constraints"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="constraints" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="interface" lowerBound="1" eType="#//Interface"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="interface" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="VarMap"
  eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//CNamedElement">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="VarMap" />
      <details key="kind" value="empty" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="fieldName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//NCName">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="fieldName" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute"
    name="referenceName"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//NCName">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="referenceName" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
</ecore:EPackage>
```

A.3 Location Aware CommUnity Model

```
<?xml version="1.0" encoding="utf-8"?>
<ecore:EPackage xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
name="locammunity" nsURI="http://www.example.org/locammunity"
nsPrefix="locammunity">
  <eClassifiers xsi:type="ecore:EClass" name="DocumentRoot">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="" />
      <details key="kind" value="mixed" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mixed"
    unique="false" upperBound="-1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFeatureMapEntry">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="elementWildcard" />
        <details key="name" value=":mixed" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="xMLNSPrefixMap" upperBound="-1"
    eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
    transient="true" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="xmlns:prefix" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="xSISchemaLocation" upperBound="-1"
    eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
    transient="true" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="xsi:schemaLocation" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="locArchitecture" upperBound="-2"
    eType="#//LocArchitecture" volatile="true" transient="true"
    derived="true" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="locArchitecture" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="LocAction"
  eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//Action">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="LocAction" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="location" lowerBound="1"
    eType="ecore:EClass ../../ca.mcmaster.cas.community/model/community.ecore#//Variable"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="location" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="LocArchitecture"
  eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//Architecture">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="LocArchitecture" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="location" lowerBound="1" eType="#//Location"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="location" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Location">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="Location" />
      <details key="kind" value="empty" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="reaches"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="reaches" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="touches"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="touches" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type"
    eType="ecore:EDataType http://www.eclipse.org/emf/2003/XMLType#//String">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="type" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="LocVar"
  eSuperTypes="../../ca.mcmaster.cas.community/model/community.ecore#//Variable">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
```

```
      <details key="name" value="LocVar" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="location" lowerBound="1"
    eType="ecore:EClass ../../ca.mcmaster.cas.community/model/community.ecore#//Variable"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="location" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
</ecore:EPackage>
```

A.4 DynaComm plus Location Aware CommUnity Model

```
<?xml version="1.0" encoding="utf-8"?>
<ecore:EPackage xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
name="locammunity" nsURI="http://www.example.org/locammunity"
nsPrefix="locammunity">
  <eClassifiers xsi:type="ecore:EClass" name="DocumentRoot">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="" />
      <details key="kind" value="mixed" />
    </eAnnotations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="mixed"
    unique="false" upperBound="-1"
    eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EFeatureMapEntry">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="elementWildcard" />
        <details key="name" value=":mixed" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="xMLNSPrefixMap" upperBound="-1"
    eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
    transient="true" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="xmlns:prefix" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="xSISchemaLocation" upperBound="-1"
    eType="ecore:EClass http://www.eclipse.org/emf/2002/Ecore#//EStringToStringMapEntry"
    transient="true" containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="attribute" />
        <details key="name" value="xsi:schemaLocation" />
      </eAnnotations>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EReference"
    name="locSubsystem" upperBound="-2" eType="#//LocSubsystem"
    volatile="true" transient="true" derived="true"
    containment="true" resolveProxies="false">
      <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
        <details key="kind" value="element" />
        <details key="name" value="locArchitecture" />
        <details key="namespace" value="##targetNamespace" />
      </eAnnotations>
    </eStructuralFeatures>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="LocSubsystem"
  eSuperTypes="../../ca.mcmaster.cas.locammunity/model/locammunity.ecore#//LocArchitecture
              ../../ca.mcmaster.cas.dynacomm/model/dynacomm.ecore#//Subsystem">
    <eAnnotations source="http:///org/eclipse/emf/ecore/util/ExtendedMetaData">
      <details key="name" value="LocArchitecture" />
      <details key="kind" value="elementOnly" />
    </eAnnotations>
  </eClassifiers>
</ecore:EPackage>
```

Bibliography

[1] R. Allen. *A Formal Approach to Softwaare Architecture.* PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley Professional, second edition, 2003.

[3] Lorenzo Bettini, Michele Loreti, and Rosario Pugliese. An Infrastructure Language for Open Nets. In *Proc. of SAC, Special Track on Coordination Models, Languages and Applications*, pages 373–377. ACM Press, 2002. URL `http://music.dsi.unifi.it/papers/SAC02-open-nets.ps.gz`.

[4] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework.* Pearson Education, 2003. ISBN 0131425420.

[5] K. M. Chandy and J. Misra. *Parallel Program Design—A Foundation.* Addison-Wesley, 1988.

[6] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 266–276, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-472-X. doi: http://doi.acm.org/10.1145/581339.581374.

[7] CommUnity Workbench development team. CommUnity workbench, August 2007 (accessed). URL `http://ctp.di.fct.unl.pt/ co/setupcw/install.htm`.

[8] J. L. Fiadeiro and M. Wermelinger A. Lopes. A mathematical semantics for architectural connectors. *Generic Programming, no. 2793 in LNCS*, pages 190–234, 2003.

[9] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, 1994.

[10] D. Garlan, J. Ockerbloom, and D. Wile. EDC architecture and generation cluster (http://www.cs.cmu.edu/~spok/adl/index.html), 1998.

[11] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[12] Object Managment Group. Mof 2.0 / xmi mapping specification, v2.1, August 2007 (accessed). URL `http://www.omg.org/technology/documents/formal/xmi.htm`.

[13] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[14] James Ivers. Wright analysis tutorial, September 1998. URL `http://www.cs.cmu.edu/afs/cs/project/able/www/wright/wright_tools.html`.

[15] javacc development team. javacc: Javacc home, August 2007 (accessed). URL `https://javacc.dev.java.net/`.

[16] jaxb development team. jaxb: JAXB reference implementation, August 2007 (accessed). URL `https://jaxb.dev.java.net/`.

[17] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992. URL `citeseer.ist.psu.edu/mcmillan92smv.html`.

[18] Formal Systems (Europe) Limited. FDR2 download page, August 2007 (accessed). URL `http://www.fsel.com/software.html`.

[19] Xiang Ling. Dynacomm: The extension of community to support dynamic reconfiguration. Master's thesis, McMaster University, 2007.

[20] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[21] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, 1995.

[22] N. Medvidovic and D. S. Rosenblum. Assessing the suitability of a standard design method for modeling software architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 161–182, 1999.

[23] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000. URL `http://citeseer.csail.mit.edu/medvidovic97classification.html`.

[24] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on The Foundations of Software Engineering (FSE4)*, pages 24–32, 1996.

[25] Cristóvão Oliveira. *The* COMMUNITY *Workbench User Manual*, 2005.

[26] Cristóvão Oliveira and Michel Wermelinger. The COMMUNITY workbench, 2005.

[27] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 209–218, 1998.

[28] Bradley Schmerl and David Garlan. Acmestudio: Supporting style-centered architecture development. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, 2004.

[29] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. In *IEE Transactions on Software Engineering*, pages 314–335, April 1995.

[30] AcmeStudio Development Team. Acmestudio download page, August 2007 (accessed). URL `http://acme.able.cs.cmu.edu/acmeweb/download.php`.

[31] AcmeStudio Development Team. Acmestudio forum page, August 2007 (accessed). URL `http://acme.able.cs.cmu.edu/forum/viewforum.php?f=2&sid=32d15a4320e2ec573fc1e734ec4323ed`.

[32] AcmeStudio Development Team. Acmelib license, May 2007. URL `http://acme.able.cs.cmu.edu/forum/viewtopic.php?p=152&sid=a9afce99cf38da0cb600778af992e4b1`.

[33] ArchStudio Development Team. Archstudio 4 - general - what is archstudio?, August 2007 (accessed). URL `http://www.isr.uci.edu/projects/archstudio/whatis.html`.

[34] Eclipse Modeling Framework Development Team. The eclipse modeling framework (emf) overview, August 2007 (accessed). URL `http://dev.eclipse.org/viewcvs/indextools.cgi/org.eclipse.emf/doc/org.eclipse.emf.doc/references/overview/EMF.html`.

[35] Graphical Editing Framework Development Team. Eclipse tools - gef project, August 2007 (accessed). URL `http://www.eclipse.org/gef/overview.html`.

[36] Graphical Modeling Framework Development Team. Graphical modeling framework, August 2007 (accessed). URL `http://www.eclipse.org/gmf/`.

[37] Rapide Design Team. Rapide: Toolset release, August 2007 (accessed). URL `http://pavg.stanford.edu/rapide/tools-release.html`.

[38] Rapide Design Team. Draft guide to the rapide 1.0 language reference manuals, 1997. URL `http://citeseer.ist.psu.edu/280921.html`.

[39] UniCon Development Team. Unicon toolset page, August 2007 (accessed). URL `http://www.cs.cmu.edu/ UniCon/toolset.html`.

[40] Wright Development Team. Wright toolset downloads, August 2007 (accessed). URL `http://acme.able.cs.cmu.edu/able-cgi/wright-register-new`.

[41] Michel Wermelinger and Jose Luiz Fiadeiro. Algebraic software architecture reconfiguration. In *ESEC / SIGSOFT FSE*, pages 393–409, 1999.