Architectural Description Languages and their Role in Component Based Design

Andreas Grau[1] Basem Shihada[2] Mohamed Soliman[3]

August 2002

1 Introduction

Architecture description languages (ADLs) are widely used to describe system architectures. Bass et. al. (1988) describe the main features at these languages. Components and connectors are the main elements of ADLs, they include rules and guidelines for well formed architectures. The output of the architecture description process is the system architecture documents. This consists of a number of graphical representations of the system model along with associated descriptive text. It should describe how the system is structured into sub-system and how each sub-system is structured into components. Each component is described based on many properties such as behaviour [1].

The generated architecture description includes a static structure of components, their dynamic process, interface module, and relationship among components. ADLs are important in system component design, since it affects system performance, robustness, disreputability and maintainability. Obviously, there is potential conflict between some of the mentioned architectures. For example, performance is improved by using large-grained components and maintainability is achieved by using fine-grained components. If both of these are important system requirements, a compromise must take place. This can be achieved by using different architecture description for different sub-system parts and later interchange the information among them [1].

1.1 Project Contribution

This is a project report for a seminar at the University of Waterloo. It is a seminar in software engineering entitled “Component based Design” It is intended to give graduate students an overview of the state of the art of this topic. Our contribution to the course focuses on software architecture and its use in software engineering. In this report we explore the capabilities of Architecture Description Languages (ADL) and how they can be used in a component based design. ADLs were developed as a utility for software developers of very large software systems. Very large and large software systems have special design problems. An introduction to these problems and possible solution by component based design and architecture description languages is given. An overview to existing ADLs and an integration framework called ACME will be presented in addition to an insight analysis into one of the ADLs called Rapide language. A description of how ADLs can be related to other interesting issues of software engineering and especially what the UML has to do with software architecture.

[1] agrau@elora.math.uwaterloo.ca
[2] bshihada@bbcr.uwaterloo.ca
[3] msoliman@bbcr.uwaterloo.ca

1.2 Project Outline

This project presents the role of architectural description languages (ADL) in component-based design. It will particularly highlight the fundamental ideas of architectural description languages which lead to better component-based design.

The rest of the project report is organized as follows: section 2 provides a literature survey of component based design and ADLs. Section 3 analyzes component based design requirements in terms of different component parts and component properties. Section 4 analyzes ADLs form two aspects, first, why it is needed? And what can it do. Section 5 specifically provides a classification and comparison analysis among the current available ADLs, also provide the role of ADLs in software modeling. Section 6 illustrates Rapid as an ADL. Section 7 illustrates Acme as an architectural description interchange language. Section 8 illustrates UML as an ADL. Section 9 provides a detailed point of view for ADLs and Ptolemy. Section 10 provides a detailed point of view for ADLs and Actor Languages. Section 11 provides a detailed point of view for ADLs and coordination models and languages. Section 12 concludes with the goals achieved from this project and explores the opportunities for future extensions.

2 Background of Component based Design and ADLs

Component-based design is a trend in software engineering for the last few years. Software engineers want to develop large and very large software systems with a long maturity. Component-based design can be an answer for many problems which arise in the development and maintenance of these large systems. In a component based design one focuses on the connections between different objects in a way similar to electrical or mechanical engineering. By defining standards for the connection one can make sure that not the whole system has to be redesigned but only one component of it [2].

Software engineers often compare this with the developments made for home stereo systems. There are products which integrated all functions into one box. If one wanted to upgrade or fix one function of the stereo system one had to deal with the whole box. Other systems with component-based design have different boxes for the main functions. The amplifier, the radio-tuner, the CD player, the tape deck and many more components can be bought in separate bodies. One can replace upgrade or fix each one of these boxes if needed instead of the complete system.

For a software development one wants to benefit from all the properties of Component based Design. So, the first thing to do is to define what we mean by a component regarding software. In the literature there is large variety of definitions and as we saw in our discussion “Component vs. Object” that it is not easy do come up with a short and sufficient definition. Two possible definitions are:

“A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.” [3]

“A component is a language-neutral, independently implemented package of software services, delivered in an encapsulated and replaceable container, accessed via one or more published interfaces. A component is not platform constrained or application bound.”[4]

These definitions are basically viewed from two different perspectives. The first one clarifies what a component can be for a developer of components where the second one focuses on the user of components. In large scale software projects, components not only give you the opportunity to split it into replaceable, reusable encapsulated parts but you can also think about their interconnection. The interconnection of components, the topology of the components is usually considered as architecture. Similar as for components, many different definitions of architecture can be found in the literature and the authors disagree in minor points. Shaw and Garlan give one definition for software architecture:

"Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns." [5]

Similar from IEEE:

"Architecture: the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution." [IEEE 1471]

In short form, this is often considered as "components + connectors + topology" [6,7]. These architectural decisions on a software system have to be done in an early stage of a software project. Errors and changes of the architecture are the most expensive mistakes if they are discovered in later phases of a project. So, a software architect needs a tool to support architectural decisions before actual implementations are done. Many people in the literature propose Architecture Description Languages (ADL) for this task [7,8]. With an ADL you can simulate the architecture for a software system without implementing the actual components. You can test and analyze it and then refine it to actual classes, procedures or functions. ADLs give you the opportunity to do a top-down design in the project. It describes the whole software system so that you can maintain it when the system gets older: It documents the system and a developer can easily understand its functionalities. So, ADLs deliver the tool to apply Component based Design to a software project. ADLs can support the whole production cycle of a software project so that one can build consistent implementations for a specific software architecture [9].

3 Foundations of Component based Design

3.1 Different parts of a component

Components essentially consist of three parts: Interface, implementation and deployment [4].

3.1.1 The interface

Interface tells the user of the component how to use it. It is the only point of access to the functionality.

3.1.2 The implementation

Implementation is the program that makes the component work. There is usually no constraint on the used language because different components in a system communicate only via the interfaces.

3.1.3 The deployment

Deployment is the environment in which the component is executed.

3.2 Properties of Components

Components do not only have the three different parts but they have important properties to be useful in a component-based design. These are encapsulation; replace ability, extensibility and description [4].

3.2.1 Encapsulation

Components functionality is only access able via the interfaces and the implementation is hidden. This allows the developer of the component to change the interior while the user is not affected. The user does not need to understand “how” the functionality is done, but “what” he can do with the component.

3.2.2 Description

There are several levels of description of a component based on the three component parts: Description if the interface, description of the implementation and description of the deployment. The most important of these descriptions is the interface description. It has to provide rich information what the component does and how it can be called. The vocabulary to access all component services has to be provided. This description can be done by formal code (i.e. in an ADL), diagrams (i.e. semiformal via UML) and informal text. The description of the implementation should show how the component was constructed and the deployment description includes the requirements for its execution.

3.2.3 Replace ability

Encapsulation and the description make components replace able. Since the component is replaceable the implementation can be altered without affecting the user. That means that the whole component can be replaced if the interface of the new component accommodates the same functionality as the old one.

3.2.4 Extensibility

In a component based system the developer of the component can extend the provided services without effect on the component user. Functionality can be added by adding interfaces to the component. The old interfaces must be still access able so that the legacy code can be executed without any change. The total functionality might be achieved by delegating the responsibility for the old services to another component which is called by the new one.

4 ADLs in Component based Design

4.1 Why do we need ADLs

Component based design concept in a very large software system is not easy to handle without an appropriate tool to support the whole development process, especially the early stages and decisions [6]. The different participants in a project have different concerns or different views of its requirements [10]. The system user wants to have a reliable and available system while the project manager is concerned about the cost and schedule. Besides, the manager wants the architecture to allow the team to work on independent chunks of the system and interact in a controlled way. The developer worries about how to

achieve all of these goals. Component based design can be a solution to meet these requirements. But, the different people need a common source of information about the system so that they can talk and understand each other. The participants of a software project need a tool to support the different views and help to construct a consistent product. ADLs have been proposed to be part of this tool [6]. They support the design of software architecture. In an ADL different levels of abstraction are supported and they can be represented in appropriate views. They also allow the simulation and the analysis of an architectural design before an implementation. This makes it possible to choose an effective, appropriate architecture and proceed with a top-down development. The components can help to distribute the work over the different team members and the ADL can help to ensure the constraints on the different components and their topology.

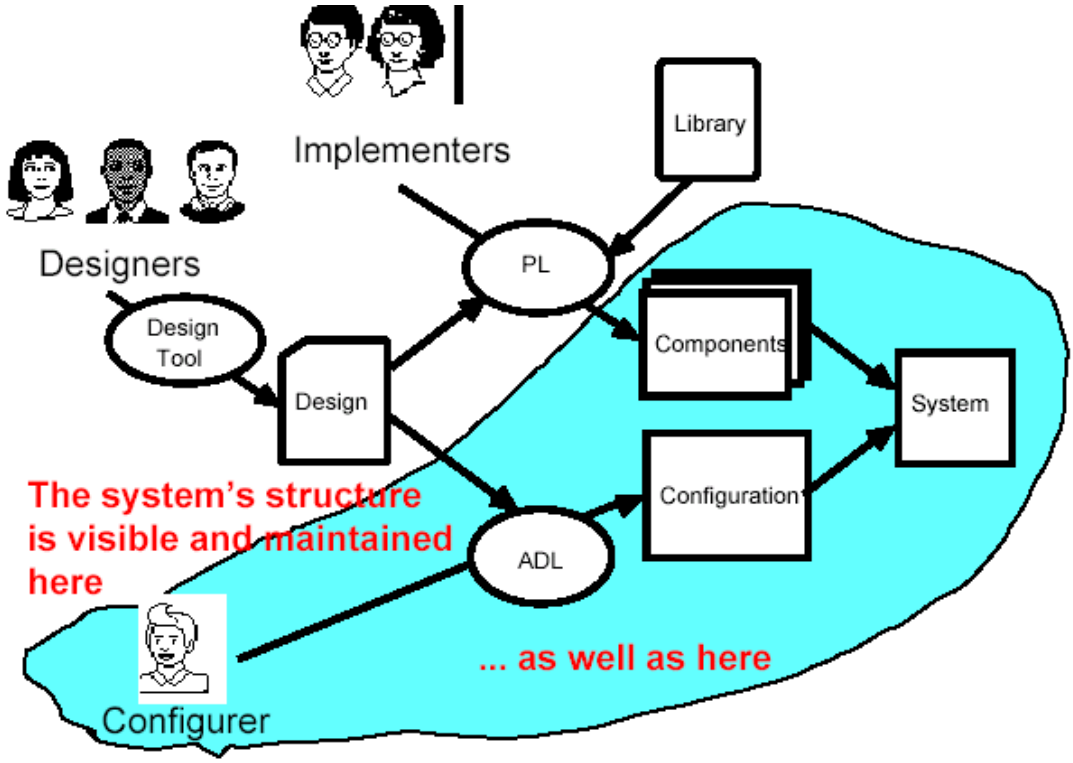

Figure 4.1: Architecture in the development process, see [11]

The project lifecycle of a software project can be complemented by an ADL to get benefit from an architectural design. Figure 4.1 shows the part an ADL can have in such a development process. The Configures or Architect develops the overall functionality of the system. The components with their configuration are described and simulated in the ADL so that a functional system originates. The designers use their tools to design components consistent with the description in the ADL. Finally the Implementers can use the design to implement the components in a programming language (PL). Such a development cycle is similar to the consent of many software engineers who demands in IEEE 1471.

4.2 What can ADLs (not) do?

One problem with the ADLs is that the whole area of software architecture is not mature yet [6]. There are many ADLs from different developers that are usually designed for special purposes only. So there are only very few that offer a project procedure as described in the previous paragraph: Most of the ADLs only concern on the most abstract

level and the architecture itself. They don't offer any help for refinement of the design or implementation. The tool support for most of the ADLs is very poor.

Anyhow there are some people who claim to offer the appropriate tool support for the project cycle [9,12]. In these examples, UML standard tools do the refinement of the architecture. The developer community of which Unified Modeling Language (UML) has reached bases this on the wide acceptance. The architecture itself is modeled by one of the ADLs (e.g. C2SADL).

The following sections focuses on the architectural definition in ADLs since this is the main task of the different languages.

5 ADL Classification and Comparison

Different ADLs have explored different features of the overall component architectural design problem. Many studies have surveyed different features of architectural design and ways to use them; all agreed that they are helping to extend our understanding of the roles that architectural description can play in software development and component design. At this early stage in the development of a discipline of software architecture, research exploration of multiple approaches to architectural description is both appropriate and necessary [6]. On the other hand, however, each ADL typically operates in a stand-alone fashion, making it difficult to combine facilities of one ADL with those of another. Instead, there are many common aspects of architectural design support that are repeatedly re-implemented again for each ADL. Examples include graphical tools for visualizing and manipulating architectural structures, facilities for storing architectural designs, and certain domain-independent forms of analysis (such as checking for cycles, or the existence of dangling connections). Such unnecessary redundancy is clearly a waste of resources for individual researchers as well as the community as a whole. Finally, for many practitioners, deeper semantic differences between different ADLs are a second-order issue. Primary it is needed to find a way to describe software architectural structures to allow a clear system structures at an appropriate level of abstraction. Currently, however, adopting an existing ADL requires a substantial effort to install the tools and learns to use them effectively, along with a significant lock-in to the selected ADL [6,13].

5.1 ADL as Software Modeling

Software architecture modeling has been covered using ADLs for both specific and general purpose design tasks. For example, Aesop, Adage, Meta-H, C2, Rapide, SADL, UniCon, Acme and Wright [6,13] architectural design tools that provide certain distinctive capabilities. Aesop supports the use of architectural styles; Adage supports the description of architectural frameworks for avionics navigation and guidance; Meta-H provides specie guidance for designers of real-time avionics control software; C2 supports the description of user interface systems using a message-based style; Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations; SADL provides a formal basis for architectural refinement; UniCon has a high-level compiler for architectural designs that support a mixture of heterogeneous component and connector types; Wright supports the specification and analysis of interactions between architectural components. Acme is a generic software architecture description language that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools [6,13,14].

5.2 Differentiating ADLs

ADLs as described earlier provide an underlying framework for characterizing architectures. ADL's suitability varies for modeling particular kinds of systems (e.g., highly concurrent systems) or particular aspects of a given system (e.g., its static properties). For example, Statecharts, partially-ordered event sets, communicating sequential processes (CSP), model-based formalisms (e.g., chemical abstract machine, or CHAM, Z), algebraic formalisms (e.g., Obj), and axiomatic formalisms (e.g., Anna) [6,13].

Z has been demonstrated as an appropriate for modeling only certain aspects of architectures, such as architectural style rules [6]. Partially-ordered event sets, CSP, Obj, and Anna have already been successfully used by existing modeling languages (Rapide, Wright, and LILEANNA, respectively). Modeling capabilities of the remaining two notations, State charts and CHAM, are somewhat similar to those of ADLs. Although they do not express systems in terms of components, connectors, and configurations, their features may be cast in that mold and they have indeed been referred to as examples of ADLs [14].

ADLs have been surveyed for applicability scope of categorization. ACME works as an architectural interchange predominantly at the structural level. Aesop specialized of architectures in specific styles. C2 focuses on architectural of highly distributed evolvable and dynamic systems. Darwin focuses on architectures of highly distributed system whose dynamism is guided by strict formal underpinnings. Rapide is used for modeling and simulating dynamic behavior described by architecture. MetaH, focuses on architectures in the guidance, navigation, and control (GN&C) domain. SADL is used for formal refinement of architectures across levels of detail. UniCon is a glue code generation for interconnecting existing components using common interaction protocols. Weaves is a data-flow-architectures, characterized by high-volume of data and real-time requirement on its processing. And Wright is used as a modeling and analysis (specifically, deadlock analysis) for the dynamic behavior of concurrent systems [6].

6. The Architecture Component in Rapide

Rapide work evolved in early 90's in Stanford University. By that time, hardware architecture description languages such as VHDL had more maturity. The event-based concept, the hardware architecture concept were incorporated with other features such as Module Languages (ML), type systems, and task sequencing languages, to inspire Rapide evolution. Now Rapide is a widely accepted software architecture description language. That is attributed to its strong model of component interaction and simulation namely; Poset or the Partially Ordered Set of Events. Rapide objectives for an ADL [16] can be summarized as:

1. Component Abstraction
2. Communication Abstraction
3. Communication Integrity
4. Dynamicism
5. Causality and Time
6. Hierarchical Refinement
7. Relativity

For a component there are two sets of behavior. The internal behavior of the component is the set of behavior that is specific to the component itself. By the definition of encapsulation, the behavior of the component should be kept internal. There is another type of behavior that is in interest to CBD, which is the external behavior of the component. Therefore, in Rapide, the component is defined by its interface and the behavior of our interest is the external behavior of the component. The external behavior of the component is the one that is necessary to allow for inter-component interactions. The event model of Rapide is a powerful tool to allow the gluing among components. In order for Rapide to achieve the gluing among components and the other required features, handling of events take special consideration.

6.1 Component Abstraction and Interface Semantics in Rapide

Component Based Design (CBD) requires components with smarter interfaces between each other to achieve several design capabilities. In Rapide, the component is abstracted by its interface. Definitely, for components to interact, they are interested in each other's interface. Therefore, description of richer interfaces is needed. It should go beyond traditional information hiding [16]. They should describe the external component behavior that is visible to other components. Interface semantics in Rapide is tackled by using two types of interaction: synchronous or asynchronous. In synchronous communication, function calls can be defined to be in the set (*provides* or *requires)*. The *"provides"* set defines the set of functions the component provides to others. The "*requires*" set defines the set of functions the component needs in order to function. Asynchronous communication can be achieved by using (*in* or *out)* event types. In and Out events represent what are the actions the component will do and what are the events the component is observing. The behavior of an interface is described by a set of transition rules. Similar to the finite state machine transition rule, it contains a trigger and a body. Figure [6.1] provides an interface description in Rapide that has provides, in, and out actions as well as the behavior section. Provides-Requires and in-out interactions are glued by connectors described in the next section.

```
Type AutoControls is interface
Provides
function speedometer return MPH;
function  Gas return Gallons;
in action Steering_Wheel (A:Angle),
in action Accelerator(P : Position),
in action Brake(p : Pressure);
out action Warning(S : Status);
behavior
Speedometer > 55   | | >
Accelerator(0) | |  Brake(High)   | | Warning(On);;
……..
End AutoControls;
```

Figure 6.1: An example component in Rapide [16].

6.2 Communication Abstraction

A successful ADL needs not only describe the interfaces and their behavior, but also describe how components are glued together. In Rapide context, this sort of description is referred as communication abstraction. In Rapide, components are connected to form architecture. The description of the connection is described in architecture in the connection section using the architecture language.

In non-component based paradigms and languages, communication is achieved inside the modules. In contrary, component-based software systems consider components as glue able components. Therefore, its communication is achieved externally. Component-based design requires more emphasis on describing the semantics of the connections among components. Thus allowing several aspired features such as: replacing components (Dynamic architectures) and facilitating COTS components.

In Rapide, a connection can bind a "*requires*" function of one interface to a "*provides*" function of another interface. It can also bind an "*in*" event to an "*out*" event of the same component or of a different component. Connection rules define this binding using event patterns with constraints. Creation Rules may be defined to set a rule on how to dynamically create a new component.

<table>
<tr>
<td><u>A Basic Connection</u>
?P : Person; ?B : Button;
Connections
?P.Push(?B) to Button_light_On(?B);</td>
<td rowspan="2"><u>Modem and Computer</u>
with computer, modem;
architecture Office is
PC : Computer;
Mod: Modem;
….
Connect
-- bi-directional flow of events
PC.S1 To Mod.S;
End Office;</td>
</tr>
<tr>
<td><u>A Dynamic Architecture</u>
with Airplane, Control_center;
architecture Air_Control_sector is
?A : Airplane; ?M : Msg;
SFO: Control_center
Connections
?A.Radio(?M) Where ?A.InRange(SFO)
| | > SFO.Receive(?M);
End Air_Control_Sector;</td>
</tr>
</table>

Figure 6.2: Example of connections in Rapide architecture language

The benefits of describing communication of component interfaces at the architecture level are to allow execution and analysis of architecture.

6.3 Communication Integrity

One objective of using an ADL is to speed up the choices on the design level. This feature is useful for CBD objectives as well. Pushing the test on ensuring that communication among the components is achieved is definitely a purpose. Rapide allows testing communication integrity at the defined architecture level. Constraints put conditions that

lead to the integrity on communication among components. In Rapide, two interfaces may indirectly communicate through an intermediate interface [16]

6.4 Dynamism and Architecture Evolution

Dynamism in an ADL is another property required for Component Based Design. That is because in our recent computer systems we have witnessed a paradigm shift in the types of systems. Dynamic Systems, Mobile systems are such types of applications that will require the architecture to expand over time to allow further functionalities to provide corrections to mal-functionalities at run time. In Rapide, dynamic architectures can use a set of declared interfaces for components along with a set of creation rules for these components. During execution, these components may be created and destroyed on the fly. In Pervasive computing systems, a user may need to roam in different domains with each one a set of services are available or unavailable depending on his location. In this time certain components may be created or destroyed on fly but they are all ensured to interoperate with the architecture when available.

Achieving complete dynamism could affect hierarchical refinement. When a component is removed dynamically from architecture, this may affect other components that may be built over it. Thus it may harm the interoperability of the built on components. Therefore, we expect some limitations on this issue. Rapide restricts this by using a previously defined creation rules in addition to a minimal set of required interfaces.

6.5 Hierarchical Refinement

One of the goals mentioned earlier for CBD is the need to build large systems. In order to achieve this goal and taking this concept from other building systems, a good idea is to define a new component that is composed from other components. This feature is named as the hierarchical composition. This feature not only allows to build big systems but to use the traditional technique of divide-and-conquer, this property is somehow similar to encapsulation. Using this concept we can build bigger and more complex systems. This concept has been recently observed in the Internet by Web-Services. Looking into a service as a facility that is provided, it can represent a component. Many nice features in Promised web-services paradigms are taken from the hierarchical composition. VASPs are Value Added Service Providers that can take the concept of composing new services from an existing set of services providing the new service on the fly to a mobile customer, for example. Rapide tools (raparch) provide a layered view of a component as shown in figure 6.3.

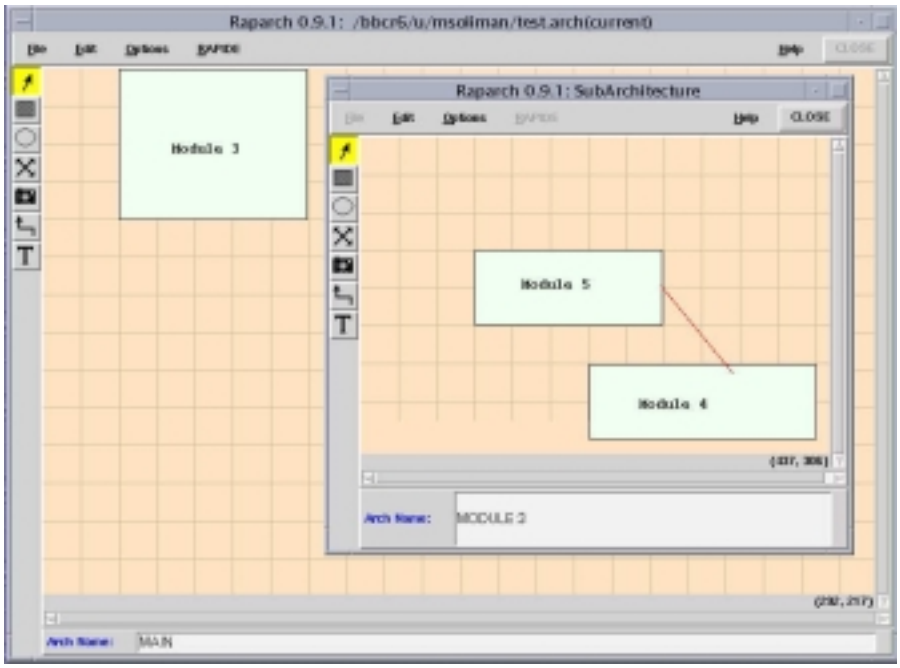

Figure 6.3: An architecture and a sub-architecture in Raparch.

In CBD perspective a component need to describe its external behavior so that other components can interoperate with it. But, when connecting several components, the new one needs to comply with component requirements. Thus achieving the requirement:

Component + Component => Component

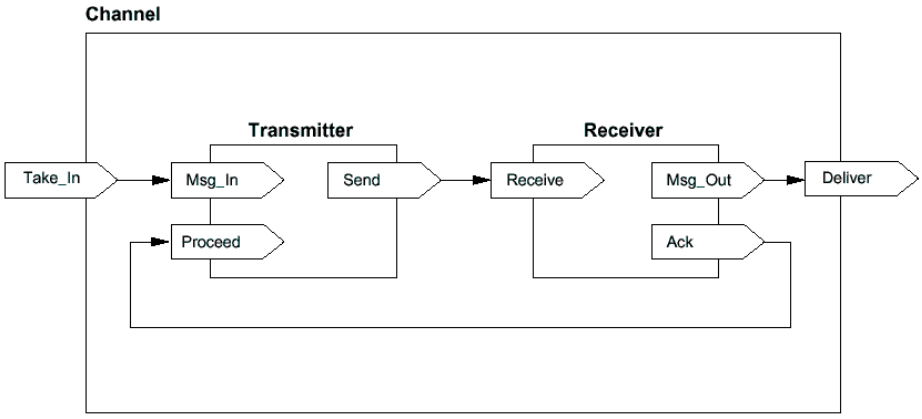

Figure 6.4: Architecture of Second Channels Module in Rapide [18]

For CBD interest, we need to have the composite component (parent architecture) have the capabilities to behave as a new component. Another important feature is to have this property built on the fly. An example of the models of computation that can achieve this task is CSP and Pi-Calculus (used in Darwin ADL, [15]). They can solve the following problem:

Given two components with each one interface is described, on connecting this set if components, the new interface should be automatically created so as it represents the external behavior of the new set of architecture (component).

In order to achieve the composability of architectures and dynamism, this property need to be automated. i.e., when having two components connected together, the resultant interface need to be automatically generated. Rapide have the capability of synthesizing a new component as described in figure 6.4. Also, Rapide can generate a new component from behavior.

The process of composing and transforming an abstract architecture into instance architecture has been recently under research. This effort leads to several enhancements such as the evolution of ACME for connecting architectures that will be described later in this paper.

6.6 Causality and Time

There is no doubt about the importance of loose coupling for component based design objectives. That has also seen in Linda, for example, to rely on the tuple space for achieving loose-coupling. An alternative is to use events as means to achieve loose coupling among components. That stresses the importance of the event model for the success of Rapide as a CBD friendly ADL. In order to check if two components are gluable (compatible), The ADL supporting CBD should provide a functionality to do consistency checks among connected components. Thus validating the connection or detecting component mismatches. Rapide surpasses this feature by providing the means to analyze the temporal aspects of interaction among connected interfaces.

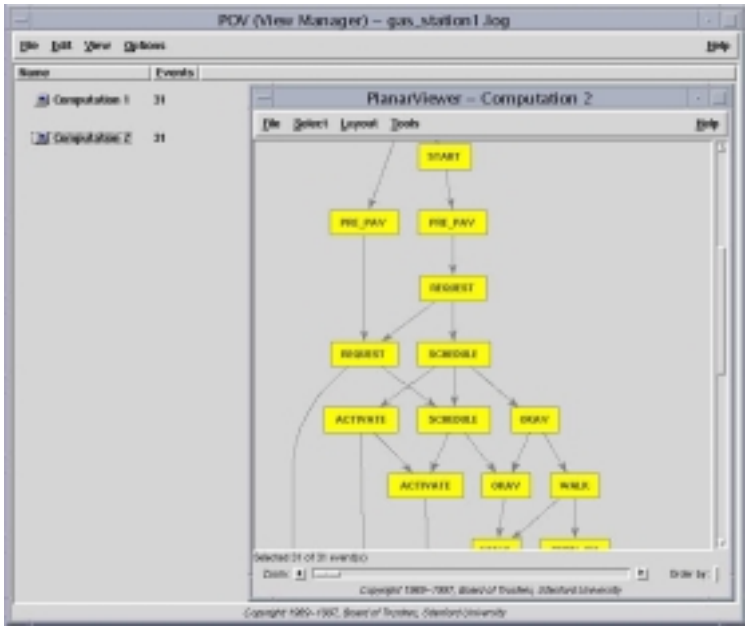

Figure 6.5: Rapide POV tool

Thanks to the POSET model, Rapide is capable of analyzing the dependencies and independencies among behaviors of interfaces, connections and their timing. Event traces can be produced to check the temporal aspects of interactions among components. For

these reasons, Rapide was selected among other ADLs to describe and analyze the behavior of service discovery mechanisms such as Jini and UPnP [19].

Packaged with Rapide Tools, a tool named POV, Rapide Partially Ordered events Viewer that allows the management of event traces generated on execution of architectures, figure 6.5. As an enhancement to VOP, Aladdin provides a set of enhancements to VOP by automatically performing dependency analysis [21]

6.7 Relativity and Dependency Analysis

Relativity of Architectures is a unique feature in Rapide. It allows comparing the behavior of one architecture (connected components) to another. Thus providing several advantages such as: comparison of several versions, proposals evaluations, and checking conformance to reference architectures. In Rapide, this is achieved by MAPPING. In this case, event pattern maps utilize the expressive power of event patterns to define the required mapping.

Two comparable architectures could be combined into a single architecture (hierarchy, section 6.5). Then an event pattern map could be created to map events from the first one to the poset of events of the second one. More details of this feature can be found in [16,17]. By utilizing a mapping at two different levels, several advantages can be gained, such as the ability to discover architecture drifts [16,17]. In other words, to discover how much the current implementation differed from the originally required architecture.

The property of relativity of architectures can be thankfully utilized for our component based design objective. It can serve in providing relative component evaluations and version inspection of COTS architectures. When a defective software component is discovered, the relativity feature can play a role in fixing and coping with this situation. Dependency analysis of a component-based architecture could assist in questions relevant to relative effects of components on others such as the following [21]:

1. Are there components that are never needed?
2. What are other components that a particular component relies on?
3. What are the effects of dynamic replacement (at runtime) of a component?
4. What are the effects on other components on changes of a component?
5. What are the components that contribute to a specific functionality? If I don't need this functionality of the system, what are the components to remove?

Relativity of architectures and dependency analysis tools built for Rapide or on top of Rapide could assist in answering the above questions. Aladdin is a tool for dependence analysis [21]. Aladdin is an ACME compatible tool (see the section below). "The portion of Aladdin that builds the Abstract Syntax Trees for the Rapide and ACME variants were obtained from the Rapide Design Team at Stanford University and the ACME research group at Carnegie Mellon University, respectively, and then directly incorporated into Aladdin" [21]

6.8 The Poset Model versus CSP

The calculus for sequential processes goes back the work done by Hoare. This work has been extended to the Pi-Calculus or the calculus for mobile processes to cover some required functionalities related to mobile systems. The CSP and Pi Calculus was used in

different ADLs and considered for a foundation of describing the temporal behavior of interactions among components. While Wright relies on CSP and Darwin relies on Pi-Calculus, Rapide relies on its unique event model. Rapide relies on the Poset powerful model for component connections that were considered for several reasons provide some additional properties than CSP [18]. CSP is another alternative; it was used in Wright ADL.

6.9 Summary of Rapide Features for CBD

The above mentioned features of Rapide provides usefulness for our objective of describing component based designs and support the process of creating architectures of COTS components. This set of features can be summarized in the table 6.1 below.

Rapide Feature	**Benefit in CBD**	**Method**
Component Abstraction	Encapsulation and replace ability + description	(Requires, Provides) and (in, out), behavior
Communication Abstraction	Description of component topology	Interfaces + Connections
Deriving new interfaces from previous ones	Hierarchical composition of components building larger components – simplifying the synthesis of large systems	Poset Model (Check)
Component Gluing	Stronger usage than CSP	Event Patterns and Poset
Dynamism	Adding and Removing components on the fly - plug and plug and play feature	Creation Rules [component synthesis from behavior]
Architecture Execution / Simulation , Communication Integrity	Dependency Analysis, correctness analysis	Architecture Execution Language and Poset
Causality and Time	Compatibility study among componenets Description	Poset
Relativity of Architectures	Relative COTS evaluations and inspection of new versions. Replace ability, extensibility, and deployment	MAPPING & event Patterns

Table 6.1: Rapide features and its usage in CBD

7 ACME

This section presents ACME approach to describe software components and component communication as an architectural design and interchange language. ACME is a project for building a toolkit which easily enables architectural description languages to interchange. One of the most important purposes of interchanging description is to achieve a well-suited environment for testing, and to formally document the intended behavior of software components from different architecture description point of view [6,13].

ADL can be viewed as a platform designed for formal specification of software components. It is well-suited for many purposes, such as design, document, implement and test software components [6]. Architectural description interchange language defines several components and component communication specification concepts by providing a component abstract view focusing on the different descriptions generated from each ADL. The generated component description can be reused straightforward or translated to other ADL formats. Component specifications can be extended to cover the overall component properties. The generated interchanged description has the ability to be used as a software component testing. Component descriptions are viewed and compared within other component specifications [13].

ACME defined as "a simple, generic software architecture description language that can be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools" [13]. This project started by Garlan, Monroe, and Wile early 1995 with main goal of providing a generic, extensible infrastructure for describing, representing, generating, and analyzing software architecture descriptions, using a common language. ACME analyses software components from different directions. First, it is an architectural interchange platform since it provides "architectural tool developers to readily integrate their tools with other complementary tools. In other words, ACME users are able to view, analyze and design components described using different ADLs as single homogenous ADL" [13]. Second, ACME is also used as an extensible foundation for new architecture design and analysis tools. Many architectural design and analyzes tools require a representation for describing, storing, and manipulating architectural designs. Since it is considered as difficult to download, install and understand the functionality of each ADL separately, it is important to provide a platform that can be a representation foundation for building tools. "ACME provides a solid, extensible foundation and infrastructure that allows tool builders to avoid needlessly rebuilding standard tooling infrastructure" [13]. Finally, "ACME has emerged as a useful architecture description language in its own right. It provides a straightforward set of language constructs for describing architectural structure, architectural types and styles, and annotated properties of the architectural elements. Although not appropriate for all applications, the Acme architecture description language provides a good introduction to architectural modeling, and an easy way to describe relatively simple software architectures" [13].

ACME provides platform to construct and illustrate software components. Components are specifies and categorized as one of ACME element, which consist of the following constitutes: The *component type constituent*, which defines its type and name, the *component object constituent*, which is an object and associates it with a type and *the component function constituent*, which is a function and specifies its parameter and result types. Additionally there are *component bindings*, which association between expressions and names [13] and c*omponent assertions* which is basically a boolean expression that must be true whenever the control returns from the function constrained by the semantic description [13].

Components represent the primary computational elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line description of software architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards and databases [13]. Connectors represent interactions among components. In other words, connectors mediate the communication and coordination activities among components. Informally they provide the "glue" for architectural designs. Examples include simple forms of interaction such as pipes, procedure call and event broadcast. But connectors may also represent more complex interactions such as a client-server protocol or a SQL link between a database and an application [13]. Systems represent configurations of components and connectors.

Components' interfaces are defined by a set of ports. Each port identifies a point of interaction between the component and its environment [13]. A component may provide multiple interfaces by using different types of ports. A port can represent an interface as simple as a single procedure signature, or more complex interfaces, such as a collection of procedure calls that must be invoked in certain specified orders, or an event multi-cast interface point [13].

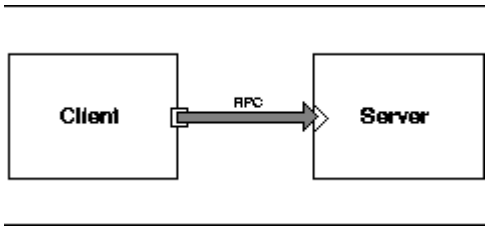

Figure 7.1: Simple Client-Server Diagram, see [13]

Connectors also have interfaces that are defined by a set of roles. Each role of a connector defines a participant of the interaction represented by the connector. Examples for the different roles are reading and writing roles of a pipe or sender and receiver roles of a message passing connector [13].

```
System simple_cs = {
Component client = { Port send-request; };
Component server = { Port receive-request; };
Connector rpc = { Roels { caller, callee}};
Attachments {
client.send-request to rpc.caller;
server.receive-request to rpc.callee;
}
```

Figure 7.2: Elements of an Acme Description, see [2]

As a simple illustrative example,
Figure 7.1 shows an architectural drawing containing a client and server component, connected by an RPC connector. Figure 7.2 contains its Acme description. The client component is declared to have a single send-request port, and the server has a single receive-request port. The connector has two roles designated `caller` and `callee`. The topology of this system is declared by listing a set of attachments.

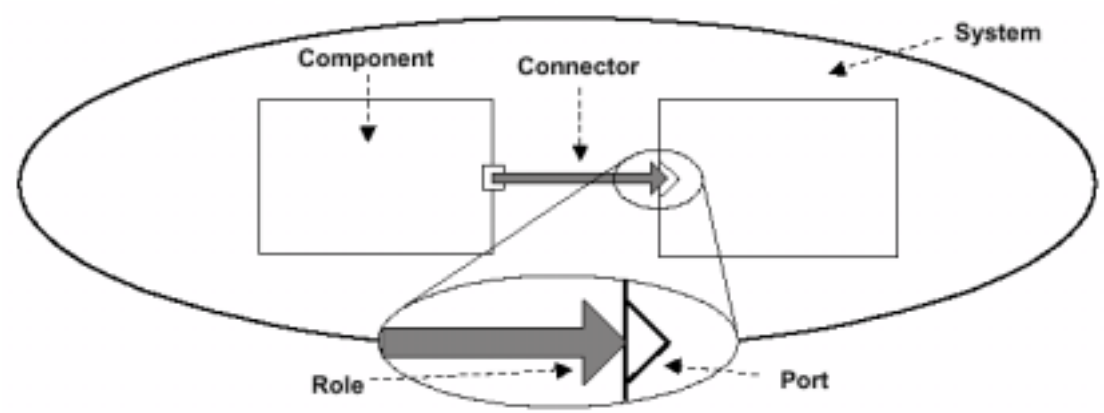

Figure 7.3: Elements of an Acme Description, see [13]

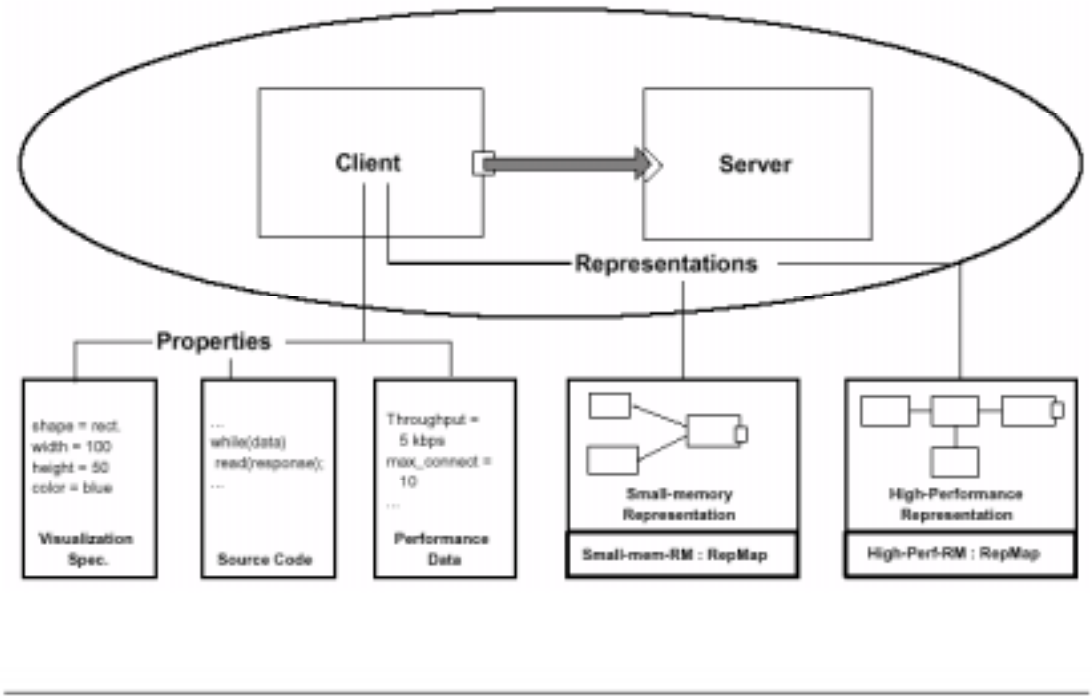

Figure 7.4: Representations and Properties of a Component, see [2]

Acme supports the hierarchical description of architectures. Specifically, any component or connector can be represented by one or more detailed, lower-level descriptions. Figure 7.4 illustrates the representation of elements. Each such description is termed a representation in Acme. The use of multiple representations allows Acme to encode multiple views of architectural entities (although there is nothing built into Acme that supports resolution of inter-view correspondences).

Representation maps illustrate the interaction between system elements and others. It provides the mapping between the internal and the external system components [13].

8 UML as an ADL

UML is a standardized modeling language for software development. It provides a graphical system to support the whole development live cycle of a software project. It has a semi-formal notation that makes it readable for human and computers. In contrast to the ADLs we presented earlier, the UML has a different concept of modeling software. Instead of an abstract architecture with components, connectors and constraints on their topology, the UML is based on the object-orientated paradigma. This allows UML to model the physical relationship of interacting objects. That means that each object and each modeled communication is implemented in the described project. In an ADL only logical components and communications are modeled with no need to an explicit implementation.

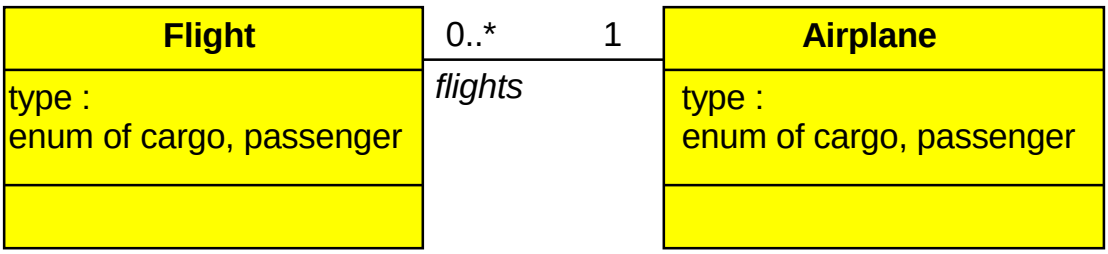

Figure 8.1: UML representation for objects.

In UML, different views are provided of the software system. Many scenarios from use cases over collaboration diagrams to deployment diagrams can be modeled within UML. But, there is a discussion in the literature that these views are insufficient to model that more abstract architecture [14]. A reusable component is not supported and the different architectural styles (i.e. pipelined, hierarchical or tuple space style) cannot be represented sufficiently in UML. There is also no connector as a first class entity in UML that would be needed to represent the architecture (see definition of architecture: “components + connectors + topology”, section 2).

Other shortcomings are that the semi-formal semantics are considered as insufficient for tools analysis and there is no automatic update mechanism that can guarantee the consistency of the different views [22].

The software engineering community has proposed mainly three different approaches to integrate UML into the architectural development. The first approach is to use one of the usual ADLs such as DRADEL and than refine the result for the architectural development by UML standard tools [9]. This allows using both, the standard tools with design and analysis capabilities of the ADL and the tools for design and analysis capabilities of UML. There are also promising efforts to implement a bridge from C2SADL to UML [12].

Another capable approach is the simulation of an excising ADL in UML [22]. There are basically two facts that allow working with such a simulation. One is based on the UML constraint language, the Object Constraint Language (OCL) [23] to represent limitations and restrictions following form the architecture chosen. The OCL has been part of UML

from the beginning and there is tool support for this feature. The other fact that allows a simulation of an ADL in UML is that the UML is based on a meta-model that itself is defined in UML. A developer is allowed to change this meta-model and gets again a valid UML description. That means that one can alter the UML meta-model to one similar to an ADL. The problem with this approach is that the standard tools do not support an altered meta-model. One would have to use special tools to design and handle the architectural descriptions.

So, people want to simulate ADLs in UML only based on OCL and so called stereotypes which allow aggregation and renaming of different objects. This admits the use of the standard tools to design and handle the architecture. An example for this procedure is given in Figure 8.2. The components and connectors of the architecture in C2 are represented by stereotypes in UML. The constraints like to property that a component cannot talk to another component without using a connector is defined in the OCL.

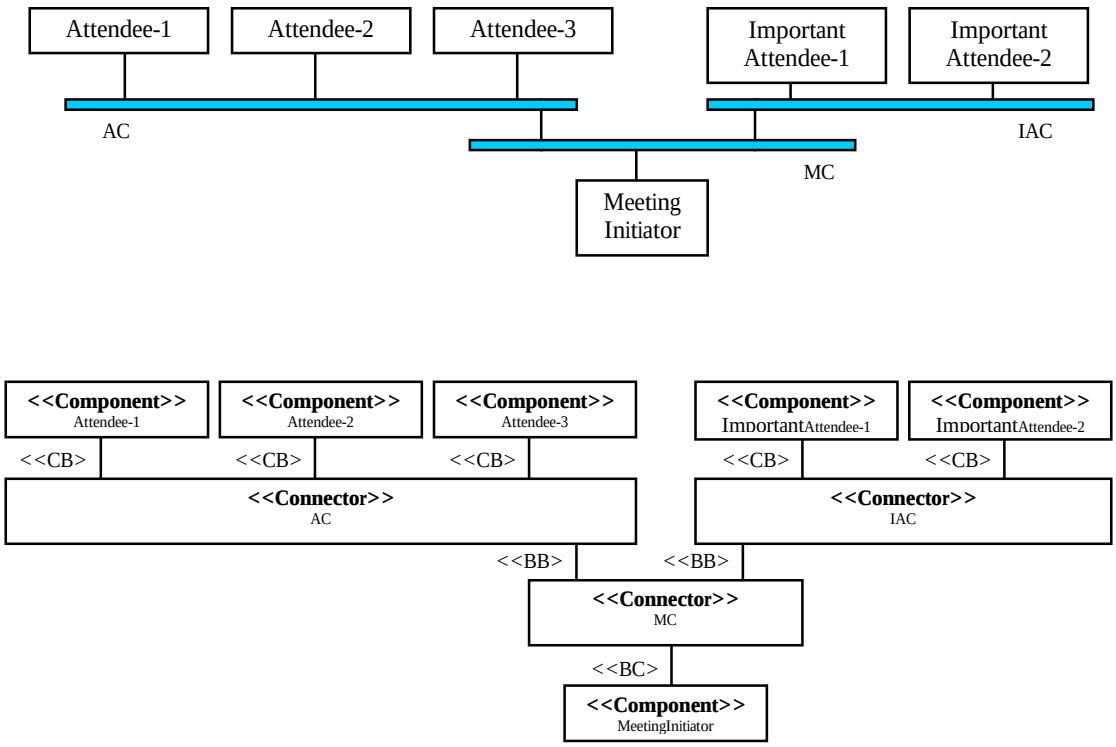

Figure 8.2: Example for a C2 architecture simulated in UML with stereotypes.

The third approach to have an architectural concept in UML is proposed by the OMG, the developer of UML. They want to extend the UML concept by a new architectural view in the version 2.0 [124]. The current version is 1.4 and it will take some time until the version 2.0 will be released and some more time until there are sophisticated tools to deal with it.

Altogether, we can summarize that UML has no component concept as it is needed for architectural development. UML can be extended to implement architectural constraints or it can complement an existing ADL and refine the architecture to an implementation.

People want to use UML because it is widely accepted in the industry and tool support is provided [25].

9 ADLs and Ptolemy

Ptolemy is a model of computation that concentrates on describing components in terms of concurrency and time. Ptolemy consists of two perspectives. Theoretical perspective which is RE, FA/DFA, CFG, Pares trees, Turing Machines, and system perspective which is the "laws of physics" governing the interaction between components and a modeling paradigm. Ptolemy has been created to describe the design of embedded system components and simulate heterogeneous embedded systems. In other words, design components that can be used under different models of computations. Figure 9.1 illustrates typical elements found in Ptolemy description.

Ptolemy intersects with ADLs in different ways. ADLs focus on providing a description to system components and their interactions. On the other hand, models of computation define the interactions characteristics in the system. Examples for different modules of computation are: rendezvous style, synchronize and asynchronous, event-based, time-driven, data flow (stream) based systems. Investigating similar issues in ADLs, it is found that ADLs also provide a component and connector characteristics for a system. For example, Wright works for CSP, and Awkward works for FIFO channel communication systems.

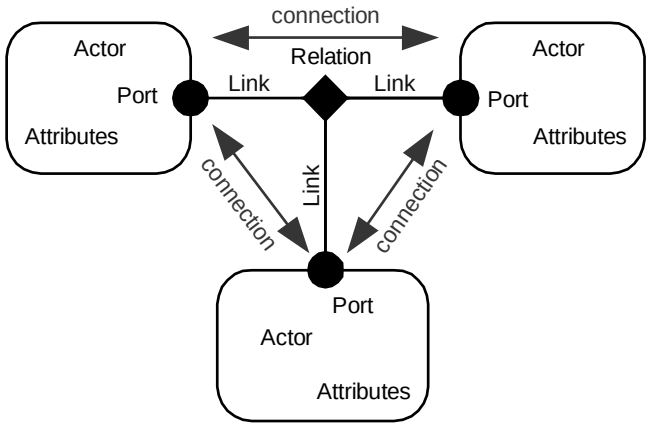

Figure 9.1: Ptolemy description elements, see [26]

The question is whether ADL is part of Ptolemy or vise versa? Or can both be integrated to provide a more comprehensive component design description.

Ptolemy is divided into two types, classic type and system-level type. The Classic type defines the token type, port type, and static and dynamic type checking and converting. The System-level type defines static aspects of an interface such as data-types, extends and captures the dynamic interactions of component types. In Ptolemy each domain has a corresponding *Receiver* interface (*Receiver* could be get(), put(), hasRoom(), hasToken()). To describe this *Receiver*, interface automata is used to capture the dynamic behavior; as a result a lattice structure is generated. To generate actor interface automaton, each domain polymorphic uses automaton to describe dynamic interaction behavior. Furthermore, each interface automaton is checked for compatibility against domain interface automata.

ADLs have similar concepts, ADLs architecture blocks have the ability to adopt Ptolemy requirements, but Ptolemy focuses only on component connectors. For example, ACME has seven classes of design element. These seven architecture blocks can be extended for more architectural description information. To accommodate the wide variety of auxiliary in formation ACME supports annotation of architectural structure with lists of properties. Each property has a name, an optional type and a value [6,13]. Any of the seven kinds of ACME architectural design entities can be annotated. Back to Figure 7.4 which shows several properties attached to a hypothetical architecture. Properties become useful only when a tool makes use of them for analysis, translation and manipulation. In ACME the "type" of a property indicates a "sublanguage" with which the property is specified [6,13].

ACME itself predefines simple types such as integer, string, and boolean. Other types must be interpreted by tools: these tools use the "name" and "type" indicator to figure out whether the value is one that they can process. The default behavior of a tool that does not understand a specific property or property type should be to leave it un-interpreted but preserve it for use by other tools. This is facilitated by requiring standard property delimiter syntax so that a tool can know the extent of a property without having to interpret its contents [6,13].

10 ADLs & Actor Languages

Actor model was first described by Carl Hewitt (70's), it consists of was computationally active entities (actors) that are capable of receiving and reacting to messages.

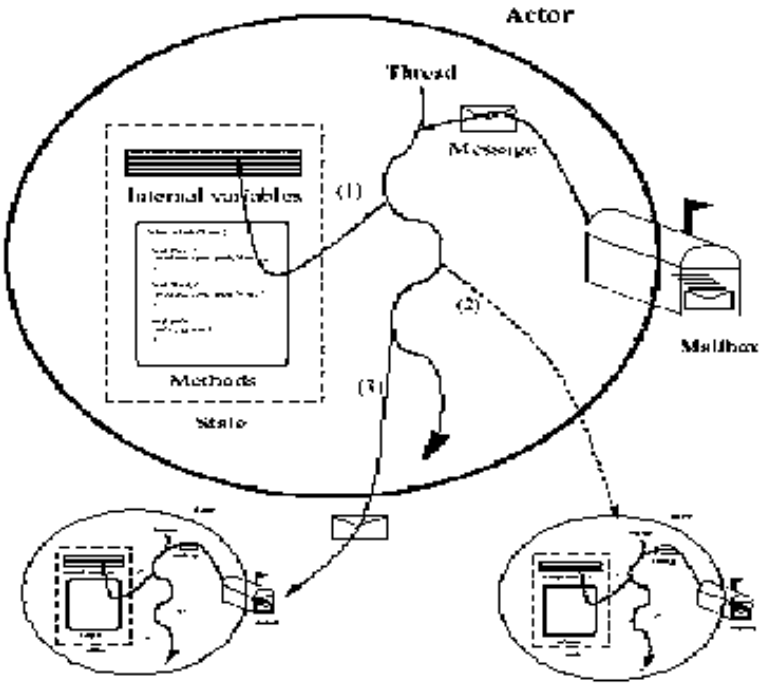

Figure 10.1: Actor Languages Structure, see [27]

Actors are independent concurrent objects that interact by sending asynchronous messages. Each actor has its mail address and behavior. Actors are capable of sending communication to other actors, create new actors, and define new behavior for itself. The

new behavior may be the same or different to its previous one. Based on the actor concept, Actor-based languages are created. These languages have many characteristics such as all procedures and declarative information are encapsulated into a single entity called actor. Actors share the main characteristics of objects in Simula and Smalltak [27]. Actors are considered being autonomous objects. A class is not integral to the actor model, inheritance in actors provide a conceptual organization of the system which is dynamically reconfigurable, sub-computation can be passed on by an actor to another actor which continue the processing. Delegation promotes modularity of the node.

As mentioned above, actors communicate using messages because it always knows the address. It receives the address within a communication from another actor, and it creates the address as part of creating another actor. Messages are represented in three tuples [27]. The information consists of a tag that distinguishes it from other tasks in the system, a target, which is the mail address to which the communication is to be delivered, and the actual message, which contains the information of the actor at the target. The system inherits events that are used to exchange or process information. However, the event arrival order is non-deterministic. Figure 10.1 illustrates the behavior in an actor-based language.

11 ADLs & Coordination Models and Languages

Coordination models and languages concentrate on concurrent and distributed systems. It is basically a provided mechanism for controlling interactions among components (component are represented as back boxes). Coordination is the process of building programs by gluing together active pieces. In other words, it is managing infrastructure for dependencies between activates. Linda is an example of a coordination model and language [28]. Coordination is expressed by the triple (E, L, M). E is the entity being coordinated (agents, processes, tuples), L is the media used to coordinate the entities (channels, shared variables, tuple spaces), and M is the semantic framework of the model (guards, synchr. constraints). Figure 11.1 illustrates the Linda Tuple space concept. The different processes P share the information in the tuple space and perform operations on this data (Rd(): read, Eval(): test value, …). Linda provides a separation between computation and coordination, it has a distribution features, it is dynamic, interoperable and implementation details are hidden from components [28].

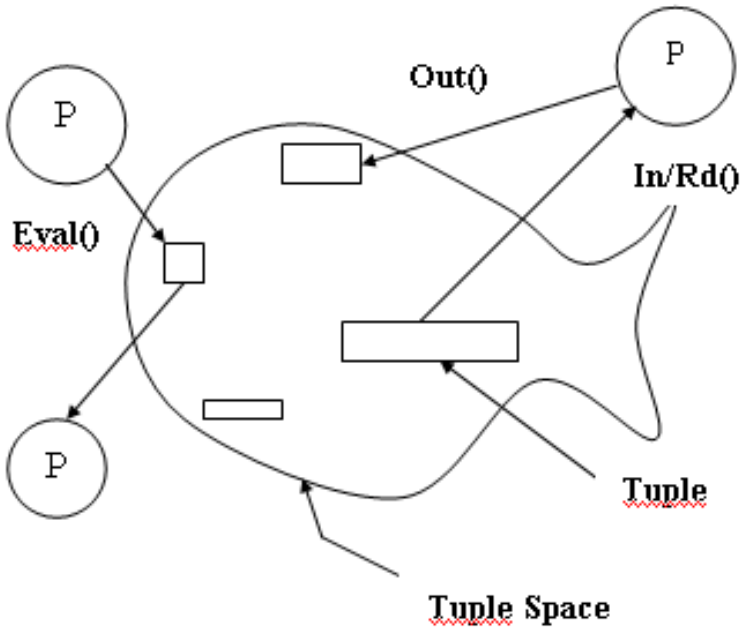

Figure 11.1 Linda Tuple space concept.

There are three other candidate technologies to work as interaction coordination. One is the concurrent OO-programming, which based on message-passing. Furthermore, it generates a process structure which is the number of processes and their relations. The next coordination technology is the concurrent logic programming. This technique concentrates on parallel systems, using policy-laden [28]. The last coordination system is the functional programming, which is based on a functionality description.

ADLs provide a similar coordination among components. For example, ACME uses representation maps [28]. A hierarchical description of architectures is supported by the description of encapsulation, i.e. components and connectors and by defining system boundaries. This is done at multiple refinement levels. When a component or connector has an architectural representation there must be some way to indicate the correspondence between the internal system representation and the external interface of the component or connector that is being represented. A representation map defines this correspondence.
This rep-map provides an association between distributed or parallel ports, or components among an enterprise distributed system [28]. Furthermore, rep-maps are essential when described extensible components, connector properties, location, and behavior.
Figure 7.3 illustrates rep-maps while Figure 7.4 illustrates component location and property list.

12 Conclusion

Component based design (CBD) is proposed as a key for the development of large software systems. Moreover, CBD is considered as a solution to recent critical system requirements (such as ..). Architecture Description Languages are high levels of abstractions that assist in the role of component based designs. They can assist in composing new architectures from either smaller architectures or components. They assist in the role of CBD by several means by working into creating paradigms that can automatically create architectures from components. The common view of an ADL is its emphasis on the definition of architecture as components, connectors and topology that define constraints. For the different ADLs presented there is different implementations that try to achieve those objectives using different techniques. Rapide uses the strong event model to glue components, UML uses the OCL to represent the constraints on objects and ACME can connect different architecture descriptions. Because ADLs are high-level abstractions but should give the techniques to analysis the behavior of interactions among components, CBD features are incorporated in them. Moreover, research trends are trying to incorporate these new features. A major obstacle for the success of an ADL is its universality. Unfortunately, the only universal hope found is in UML, which needs substantial work to make it a true ADL. Meanwhile, Ptolemy project achieved a step ahead by incorporating the universal component. An interesting additional research can be directed into comparing at two ADLs (such as Rapide and ACME) versus Ptolemy. Alternatively, once incorporating the different views of ADLs was successful under UML umbrella, we can then have further comparison.

13 Acknowledgements

We would like to thank Prof. Mavaddat for his insightful discussions, which led us to more understand the concept of component based design and many other topics discussed within this seminar.

14 References

[1] Ian Sommerville, “Software Engineering”, sixth edition, 2001, Addison-wesley.

[2] O. Nierstrasz, J.G. Schneider and M. Lumpe, “Formalizing Composable Software Systems — A Research Agenda,” Proceedings of 1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, Chapmann & Hall, 1996, pp. 271-282.

[3] Alan W. Brown, Kurt C. Wallnau: “An Examination of the Current State of CBSE”: A Report on the ICSE Workshop on Component-Based Software Engineering, 1998
International Workshop on CBSE, http://www.sei.cmu.edu/cbs/icse98/summary.html

[4] Kirby McInnis: “Component-based Development: The Concepts, Technology and Methodology”, CASTEK,
http://www.cbd-hq.com/PDFs/cbdhq_000901km_cbd_con_tech_method.pdf

[5] M. Shaw, D. Garlan: “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, April 1996.

[6] N. Medvidovic, R. Taylor, “A classification and Comparison Framework for Software Architecture Description Language.” IEEE Transaction on Software Engineering, Vol 26, NO.1, Jan 2000.

[7] David Garlan and Dewayne E. Perry: “Introduction to the special issue on software architecture.” IEEE Transactions on Software Engineering, 21(4):269–274, 1995.

[8] C. Williams, “Software Architecture: Implications for Computer Science Research.” Proc. First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, February 1999.

[9] SAAGE: “Software Architecture, Analysis, Generation, and Evolution”
http://sunset.usc.edu/~neno/Flyer.pdf

[10] P. Clements, L. Northrop: “Software Architecture: An Executive Overview” Technical Report, Carnegie Mellon University, February 1996

[11] J. Bishop: “Connecting components: theory and practice”, University of Pretoria, South Africa, http://www.csc.uvic.ca/news/events/resources/jbishop_00sep28.pdf

[12] Alexander Egyed, Nenad Medvidovic: “A Formal Approach to Heterogeneous Software Modeling.” FASE 2000: 178-192

[13] D. Garlan, R. Monroe, D. Wile, “Acme: An Architecture Description Interchange Language”, proceedings of CASCON 97.

[14] P. Clements: “A survey of architecture description languages.” In Eighth International Workshop on Software Specification and Design, Paderborn, Germany, March 1996.

[15] J. Magge and J. Kramer, “Dynamic Structure in Software Architecture,” ACM SIGSOFT, 1996 .

[16] D. Luckham and J. Vera, “An Event-Based Architecture Description Language, “ IEEE Transactions on Sofware Engineering, Vol. 21, No. 9, Sept. 1995.

[17] D. Luckham, J. Kenney and l. Augustin, “Specification and Analysis of System Architecture Using Rapide, “ IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995.

[18] D. Luckham et al, “Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems,” (1992), available at (http://citeseer.nj.nec.com/cs).

[19] C. Dabrowski and K. Mills, “Analyzing Properties and Behavior of Service Discovery Protocols using an Architecture-based Approach”, available at (http://citeseer.nj.nec.com/cs).

[20] D. Perry and A. Wolf, “Foundations for the Study of Software Architecture,” ACM SIGSOFT, Oct. 1992.

[21] J. Stafford and A. Wolf, “Architecture-Level Dependence Analysis for Software Systems,” Technical Report, CU-CS913-00, December 2000.

[22] Nenad Medvidovic, Alexander Egyed, David S. Rosenblum: “Round-Trip Software Engineering Using UML: From Architecture to Design and Back.” Proceedings to 2nd WOOR, Toulouse, France 1999, pp. 1-8

[23] OMG: “OMG Unified Modeling Language Specification.” Version 1.4, September 2001, http://cgi.omg.org/docs/formal/01-09-67.pdf

[24] A. Schürr, A.J. Winter: UML, the Future Standard Software Architecture Description Language? , in: H. Kilov, B. Rumpe, I. Simmonds (eds.): Behavioral Specifications for Businesses and Systems, Kluwer (1999), 193-206

[25] Rational Rose: http://www.rational.com

[26] Edward A. Lee, "What's Ahead for Embedded Software?," IEEE Computer, September 2000, pp. 18-26.

[27] Gul Agha, “An Overview of Actor Languages”, In Special Issue of the SIGPLAN Notices on the object-oriented programming workshop, pages 58--67, October 1986.

[28] N. Carriero and D. Gelernter, "Linda in Context," Comm. of the ACM, Vol. 32, No. 4, pp. 444458, April 1989.