Bridging the Gap between IEEE 1471, Architecture Description Languages and UML

Mohamed Mancona Kandé,Valentin Crettaz, Alfred Strohmeier, Shane Sendall

Swiss Federal Institute of Technology Lausanne (EPFL)
Software Engineering Laboratory
1015 Lausanne EPFL, Switzerland
email: {Valentin.Crettaz, Mohamed.Kande, Alfred.Strohmeier, Shane.Sendall}@epfl.ch

ABSTRACT *A lot of attention has been paid to software architecture issues in both the software engineering research community and standardization organizations working in the software area. On one hand, IEEE 1471 makes a clear distinction between the architecture and the architectural description of a software system. The software architecture research community, on the other hand, has focused on the creation and improvement of special-purpose languages, architecture description languages (ADLs). ADLs have the advantage of being mathematically founded, facilitating analysis of architectural models, but they have also the disadvantage of lacking adequate support for separating various kinds of stakeholders' concerns along different viewpoints. ADLs do not address the clear difference between software architecture and its representations, as does the IEEE 1471. To help improve the situation, we propose a UML-based approach to software architecture that instantiates the conceptual framework defined in IEEE 1471 and complements the abstractions and mechanisms found in current ADLs.*
In this paper, we introduce the ConcernBASE approach to software architecture description and discuss how to integrate it with SADL, a particular ADL. We validate the mapping in ConcernBASE Modeler, a UML-based tool prototype, by integrating SADL tools.

KEYWORDS *Software Architecture, Architecture description, UML, ADL, IEEE 1471, SADL, Advanced Separation of Concerns, MDSOC, Views, Viewpoints, Concern space.*

1 Introduction

Software architecture is concerned with understanding and describing complex software-intensive systems at different levels of abstraction. The attention paid to issues of software architecture is increasing in both the software engineering research community and standardization organizations working in the software area. However, in spite of the number of books and research papers found in the literature, architectural concerns still continue to be one of the most contentious issues in the construction of complex systems.

IEEE 1471 defines an *architecture* as "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". In addition, it refers to an *architecture description* as "a collection of products to document an architecture"[1]. By these definitions, IEEE 1471 makes clear the distinction between the architectural description and architecture of a software-intensive system.

On the other hand, the software architecture research community, essentially academics, has focused on the creation and improvement of special-purpose languages, known as architecture description languages (ADLs) [7][8][9]. In the past few years, numerous ADLs have been specially designed to represent different aspects of architectures of software systems. ADLs have the advantage of being mathematically founded, facilitating analysis of architectural models, but they have also the disadvantage of lacking adequate support for separating various kinds of stakeholders' concerns along different viewpoints. Due to their formal nature, ADLs can be hard to understand and to use, as developers in need of ADL-based software architectures will have to learn the mathematical models of software systems. This is perhaps a reason why ADLs are not so widely used in industry [14]. In addition, ADLs do not address the clear difference between software architecture and its representations, as does the IEEE 1471.

To help improve the situation, we propose a general approach, which aims at integrating fundamental issues addressed in IEEE 1471 with some effort accomplished in the software architecture research community. We refer to this approach as ConcernBASE[1], a particular UML-based approach to software architecture that instantiates the conceptual framework defined in IEEE 1471 and complements the abstractions and mechanisms found in current ADLs, allowing for simultaneous separation of possibly overlapping concerns. To support users, ConcernBASE has also focused on providing tools for supporting architecture-centered software development.

1.ConcernBASE stands for ***Concern-B***ased and ***A***rchitecture-centered ***S***oftware ***E***ngineering [20]

In this paper, we introduce the ConcernBASE approach to software architecture description and discuss how to map a ConcernBASE architectural description, written in UML, onto an architectural description developed in a particular ADL, called SADL (Structural Architecture Description Language). Our motivation for doing this mapping was to make available the verification capabilities of SADL tools for ConcernBASE. The mapping has been validated in ConcernBASE Modeler, a UML-based tool prototype that supports the ConcernBASE approach and its integration with SADL tools [21].

The paper is organized as follows: section 2 gives some background on our work and the related work done by others. Section 3 introduces the ConcernBASE approach. Section 4 illustrates the application of the ConcernBASE approach on a compiler example, which is based on the reference model for compiler construction. Section 5 briefly presents the key concepts of SADL. Section 6 presents a method for translating ConcernBASE models to SADL specifications. Section 7 gives a concise overview of the tool for the ConcernBASE approach. Finally, section 8 summarizes the paper and discusses future work.

2 Background and Related Work

The Unified Modeling Language (UML) is a widely used standard and a general-purpose language, which provides a large number of well-known techniques and concepts for modeling various kinds of software artifacts from different perspectives or viewpoints. Unfortunately, UML, in its current state, is not sufficient for an explicit software architecture description as argued in [11]. To gain the benefit of software architecture description with UML, UML needs to provide first-class support for some key ADL concepts, such as connectors and styles.

Several strategies have been proposed to map ADL constructs to UML elements [10][14]. To make use of the expressive power of a specific ADL, each of these strategies has focused on analytical evaluation of architecture descriptions. Although the result of such mappings is an important means to provide an overall view of the system at hand, they often have no associated tool support. Therefore, it is very difficult to make any judgement about the feasibility of such approaches. On the other hand, the strategies used have focussed on increasing the popularity of existing ADLs, rather than addressing some key issues that motivate advances in UML 1.x, allowing UML 2.x to be a better standard. Moreover, despite the significance of the notion of multiple views in software architecture, as a fundamental principle for structuring architectural descriptions [15][16][17], ADLs and their mapping to UML do not address this issue. Fortunately, the standard IEEE 1471 [1] has proposed to improve the situation by introducing the concept of multiple viewpoints in software architecture descriptions. These viewpoints allow one to separate the stakeholders' concerns into different sets of related concerns; each set represents a certain aspect of the system that can be "seen" from a particular viewpoint.

In previous work [2], we proposed a UML-based approach to software architecture description using the IEEE 1471, which focused on incorporating key abstractions, found in nearly all-existing ADLs, into UML. This resulted in the definition of a structural viewpoint of software architecture, whose viewpoint language we called a UML profile for the structural description of software architectures.

This profile has been integrated into ConcernBASE to define its Structural Viewpoint. ConcernBASE is centered around the mechanisms of multi-dimensional separation of concerns (MDSOC) [6][12], an advanced form of separation of concerns. MDSOC is a conceptual framework that allows one to identify, simultaneously separate and manipulate software concerns along multiple dimensions (kinds of concerns). It includes some mechanisms for composing and decomposing software concerns and addresses the ability to handle new concerns, and new dimensions, dynamically, as they arise throughout the software lifecycle. However, unlike other implementations of MDSOC, ConcernBASE uses the standard UML notation whenever possible. In addition, it addresses some fundamental limitations of UML1.x by providing necessary extensions to UML to enable a "concern-driven" approach to software architecture description.

The Structural Architecture Description Language (SADL) [4] is a particular ADL that focuses on understanding, specifying and refining the representation of structural concerns in complex software systems. SADL is different from other ADLs, such as Wright [13], as it supports structural decomposition at multiple levels. This is called refinement of high-level system structures in the SADL terminology. However, SADL is only capable of providing support for structural decomposition along a limited number of dimensions (e.g., components, connectors, configurations, as introduced in section 4). The SADL support for behavioral modeling is very restricted.

3 The ConcernBASE Approach

According to IEEE 1471, a *viewpoint* is a specification of the conventions for constructing and using a view, while a *view* is a representation of a whole system from the perspective of a related set of concerns. *Concerns*, as defined in [1], are those interests which pertain to the system development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders. Concerns can be logical or physical concepts, but they may also

include system considerations such as performance, reliability, security, distribution, and evolvability. These are important and standard definitions that are considered in the remainder of this paper. IEEE 1471 helps us understand intuitively the need to separate the set of all concerns involved in a software system into different viewpoints, but it does not specify how an architect should identify, categorize, and encapsulate the concerns that pertain to individual viewpoints. Furthermore, it remains intentionally silent on how to represent concerns in architectural views.

ConcernBASE provides a particular approach to software architecture description that aims at addressing these issues, by using UML and instantiating both the MDSOC and IEEE 1471 conceptual frameworks. One important goal of ConcernBASE is to provide a UML-based instantiation of both conceptual frameworks. Another goal is to provide mechanisms to produce software architecture descriptions in a flexible and incremental way, allowing one to identify, separate, modularize and integrate different software artifacts based on various kinds of concerns.

Throughout the approach, we take the premise that software architecture is multidimensional in nature. That is, when constructing complex software, an architect will have to represent the system in many different ways in order to be able to understand, communicate and reason about its high-level properties, from different viewpoints. Each way of representing the system may be considered as a different view of the architecture of the system that consists of one or more models. Each model allows us to reflect some aspects of the system, while hiding others from view.

3.1 Key Concepts in ConcernBASE

In ConcernBASE, a viewpoint is defined in a template called *viewpoint schema* that fulfills the requirements of IEEE 1471. To fulfill these requirements, a viewpoint schema:

- defines a unique identifier for the viewpoint at hand;
- identifies a set of stakeholders along with a set of the various kinds of concerns that pertain to those stakeholders;
- provides an approach to facilitate the choice or definition of modeling elements that any associated representation language needs to support;
- identifies the associated architectural view that represents the stakeholders' concerns in one or more architectural models;
- and provides the sources for key information used in or related to the viewpoint definition.

In addition to the requirements of IEEE 1471, ConcernBASE allows one to define the rationale for a viewpoint and provide some relationships between the different kinds of concerns to be addressed, using a viewpoint schema. One example of an architectural viewpoint defined in ConcernBASE is the structural viewpoint.

A new concept introduced by ConcernBASE is the notion of *concern spaces*. A concern space represents a conceptual repository that contains all relevant information related to a particular viewpoint. The concern space takes the viewpoint schema as an input and refines the information it contains. ConcernBASE allows us to structure the set of concerns into different kinds of concerns (or dimensions), to specify the relationships between these categories and maintain changes in the concern structure. A concern space can be considered as a "multi-dimensional model of system considerations" that pertains to a software architect from a particular perspective.

Different elements of the set of concerns addressed by the viewpoint can be represented in different models by means of projections. A *projection* is an architectural abstraction that defines the relationship between a viewpoint and a view (i.e., models of a view). It consists of a set of rules that specifies how to encapsulate (one or more) concerns into (zero or more) model elements, taking into account that some concerns might not have adequate representations in the language at hand. A model element can be simple or composite. A simple model element can be, for example, any basic UML elements, such as a link, attribute, parameter, etc. Examples of composite model elements include classes, subsystems, packages, and any type of a UML diagram. Basically, a projection can be any set of rules that specifies how to decompose, organize, structure software according to a specific dimension. Different projections along different dimensions result in different models, but also different projections along the same dimension may result in the same or different models. All sets of rules defining projections are defined and maintained as parts of the concern space.

A *view* of the software architecture of a system is a partial architecture description of that system that may have one or more architectural models. The architecture description of the whole system may be considered as a set of different architectural views. In the same way, we consider the system concern space as the union of all concern spaces of individual viewpoints.

Introducing viewpoint schemas in ConcernBASE has two main benefits:

- *Viewpoint schemas allow us to define concern spaces*. A viewpoint schema provides a large amount of information that can be further refined, structured and stored in a conceptual repository for the viewpoint.
- *Concern spaces improve our understanding of the relationship between architecture and its description*. ConcernBASE, by introducing concern spaces, provides a

means to understand and illuminate, to some extent, the relationship between a viewpoint and a view. Using a concern space, a viewpoint can be characterized as an approach or an architectural mechanism for separating, analyzing and using software concerns. On the other hand, a view associated with a viewpoint can be characterized as a particular work product that can be manipulated. A view consists of a set of architectural models that each represent some set of software concerns. Taking into account the definitions of architecture and architecture description given in [1], and considering both a viewpoint as a set of abstractions, rules and guidelines and a view as a work product, we decided to recognize viewpoints as parts of the architecture, while recognizing views as parts of the architecture description. Thus, given a concern space and a model (view) that represents a set of concerns (viewpoint), it should be possible to define a set of rules (projection) that specifies the relationship between the viewpoint and its associated view. This reasoning can be generalized to improve our understanding of the relationship between architecture and its description. However, it does not always work the other way, because some concerns cannot be represented in a given language.

3.2 Structural Viewpoint

The structural viewpoint is an example of a particular ConcernBASE viewpoint, whose specification is shown in Figure 1. The structural viewpoint addresses concerns related to static, behavioral and configurational structure. In this example, we first show how to use the ConcernBASE viewpoint schema; then we discuss how the information it contains can be used as an input for defining a viewpoint language and the structural concern space. We choose this example as it allows one to understand the key idea behind the ConcernBASE approach to software architecture description, while introducing some concepts needed for mapping ConcernBASE to SADL.

The structural viewpoint schema, depicted in figure 1, consists of 8 sections (titled in bold). The first section gives the name of the viewpoint, structural viewpoint. The second and third sections identify the lists of concerns and stakeholders to which the concerns pertain, respectively. The structural concerns listed in the second section correspond to those typically found in most ADLs. They all pertain to the architect. But some of them can be of interest to developers, for instance, the realization of a specific computation can be assigned to a particular developer. The assignment of the computation to the developer represents an example of information that can be taken from the schema, refined, structured and stored in the concern space. The fourth section provides the rationale for the structural viewpoint. Section 5 is the most important part of the viewpoint schema. It provides an approach to specifying the key characteristics of the viewpoint, by considering the viewpoint as a module of the architecture whose concerns can be affected by and/or affect other concerns. Each concern addressed by the viewpoint can be used in one of the following three situations: motivating the need for new decisions (as an incentive factor); allowing one to specify what decisions to take (as a decisional factor); and describing the degree of satisfaction with the decisions taken together with their impacts on the architecture (as a resultant factor).

To identify the incentive factors in practice, it is often useful to ask a number of questions [20]. For example, the incentive factors listed in figure 1 have been identified from questions, such as: What carries the interactions among computational elements? What characteristics does it need to have? What axioms and design principles can be reused? In the subsection for decisional factors, an answer to each of these questions will require some stakeholders to make decisions and document them in the schema. In the example, looking for a response to the first question led us to separate different kinds of interactions and to modularize them into a Connector dimension. Having defined a dimension as a decisional factor does not require it to be a first-class model element in the viewpoint language. For instance, a viewpoint language does not need to support connectors as a first-class model element. Depending on the impact of the Connector (as a dimension) on the architecture and the background of the architect (or an ADL designer), connectors can be added to the resultant factors, as a candidate meta-type that needs to be either selected from an existing description language or defined in a new one.

Some relationships between particular concerns (without curly brackets) or between the kinds of concerns (with curly brackets) are described in section 6 of the schema. Section 7 of the schema provides some important information sources related to the viewpoint; and finally Section 8 of the schema identifies the architectural view whose models need to be supported by the viewpoint language.

3.3 Structural View

The structural view is the most abstract representation of all significant structural concerns that are relative to the structural viewpoint. It focuses on what kind of architectural components, connectors, constraints and styles are needed to understand and reason about the system's structure. The structural view abstracts from many details of the system components and connectors and does not provide any information on how the communication among the architectural components is implemented or on the internal structure of those elements. The elements represented at this level often need to be refined in other models. To get

more details, the structural view needs to establish some refinement techniques that support 3 types of model: static, behavioral, and configurational.

Viewpoint name		Structural viewpoint
Concerns		Computation, data store, interaction, configuration, constraints, reuse (of axioms and design principles)
Stakeholders		Architects, developers, maintainers, acquirers
Rationale		To identify, understand, specify, represent and reason about the structural characteristics of a software-intensive system
Approach	Incentive factors	▪ The computational and data elements that make up the system ▪ The static and dynamic organization of computational and data elements ▪ The protocols/roles used for communication ▪ The carrier of interactions among computational elements ▪ The axioms and design principles that are reused
	Decisional factors	1. Separation/modularization of architectural concerns into dimensions of interest, expressed as: *{set of various kinds of concerns}* → *Dimension*. This means, for structural concerns, ▪ {computation, data store} → Component, ▪ {multiple kinds of interactions} → Connector, ▪ {configurations} → System, ▪ {axioms, design principles} → Style, ▪ {constraints} → Property 2. Organization of groups of structural concerns into models (here 3 different models): ▪ {static structures, dynamic structures, configurational structures} → model
	Resultant factors	A set of architectural units characterizing the viewpoint language: ▪ {components, connectors, system, style, properties} *Satisfaction*: each structural concern can be modularized *Impacts*: support for decoupled units of computation/data store *Noteworthy*: one dimension per architectural unit *Unsatisfactory*: Insufficient support for modeling crosscutting concerns (e.g., extra-functional aspects)
Relationships among Concerns		configuration *IsMotivatedBy* ⟨{computation}, {interaction}, {constraint}⟩ {constraint} *AppliesTo* {{computation}, {interaction}, {configuration}} ⟨{computation}, {interaction}, {constraint}⟩ *MemberOf* style
Source		ADL [8], IEEE-Std-1471 [1], MDSOC [12], Quality Attributes [18]
Resulting View		Structural view::{Static, behavioral, configuration}

Figure 1: Structural Viewpoint Schema

3.3.1 Static Model

The static model describes the static structure of the components and connectors composing the system. *Computational components* represent subsystems, system-level reusable modules with well-defined interfaces, or plug-in capabilities[1]. A computational component is a locus of definition of some computation and data concerns, which usually do not crosscut the boundaries of a single subsystem or module. Some components may have internal structures that can be represented at subsystem or lower levels using a number of representation units. Thus, the representation units that compose a specific component must pertain to those computation and data concerns which are modularized by the same component.

The UML Profile for SADL defined for ConcernBASE supports the specification of computational components by using a class-like notation. To visually distinguish computational components from other components, such as classes, the keyword <<computational>> or the computational icon (placed in the upper right hand corner of the class name compartment) are used. LexicalAnalyzer, shown in figure 4 is an example of a computational component.

The interface of a component is specified as a collection of several interface element types, each of which defines a

1.As described later, dynamically attaching and detaching connection points to components, as defined in system configurations, enable our component model to describe plug-in capabilities.

logical interaction point between the component and its environment. The interface elements of a component can be of three different types: operational, signal or stream. An <<operational>> interface element type of a component describes a set of operations that can be required by or provided to other components, whereas a <<signal>> interface element type specifies a set of signals that can be sent to or received from other components. A <<stream>> interface element type enumerates a collection of streams that can be consumed by or produced for other components, as well as a set of quality of services to be guaranteed by those streams. There is a composition association between a component type and its interface element types.

A connector is a locus of modularization for component interconnections and communication protocols. Basically, the static structure of a connector consists of connection points and a connection role. A *connection point* describes a point at which a component can join a connector to communicate with other components. Thus, it represents an element of the connector interface through which the participation of a component in an interconnection can be defined. A *connection role* is an abstract representation of the channel between compatible connection points. It also specifies the protocol of interactions between connection points.

3.3.2 Behavioral Model

The behavioral model describes the dynamic (or behavioral) properties of all architecturally significant elements of the system under development. The behavior of a computational component is specified by the component interface protocol (CIP). A CIP defines the temporal ordering of data flows, call events, and signal events that can be received or sent by the component. It is defined by composing the protocol statemachines of all interface elements. Composition is defined by "anding" all statemachines of the interface, i.e. the statemachine of each interface element runs concurrently to all the others.

The behavior of a connector type is defined by specifying the protocol of interactions for each connection role and the behavior associated to the connection points. Both of these are described using UML protocol statemachines.

3.3.3 Configuration Model

The configuration model describes the organization of the system in terms of component and connector type instances. An instance of a connector type has two categories of elements: dynamic ports and links between these ports. The *dynamic ports* are instantiations of connection points, whereas the links are instantiations of connection roles. Similarly, when a component type is instantiated, its interface element types are instantiated as *static ports* that are parts of the boundary of the component instance. Two or more component instances can then be interconnected to define a configuration of the system by attaching dynamic ports of the connector instance(s) to the component instances. Before a dynamic port is attached to a component, we have to check that its contract is fulfilled.

4 Compiler Example

This section presents an example that illustrates the benefits of the ConcernBASE approach by applying its techniques to a well known compiler example. Figure 2 depicts an informal representation of a Level-3 Compiler architecture taken from [4], which uses the reference model for compiler construction.

Despite the box-and-arrow architecture representation, figure 2 shows that the compiler has a batch-sequential architectural style. The main component coordinates the correct execution sequence of the components composing the compiler system. First, it transfers the control to the LexicalAnalyzer, then to the Parser, then to the AnalyzerOptimizer, and finally, to the CodeGenerator. The rounded-edge components, SymbolTable and Tree, are shared-memory components. The former holds binding information and makes it available to the LexicalAnalyzer and AnalyzerOptimizer. The latter keeps abstract syntax trees and is accessed by the Parser, AnalyzerOptimizer and CodeGenerator. Note that some components have read and write access, while others are only granted read or write access. The Parser component is directly receiving tokens from the LexicalAnalyzer via the unidirectional pipe relating them and not through shared-memory components.

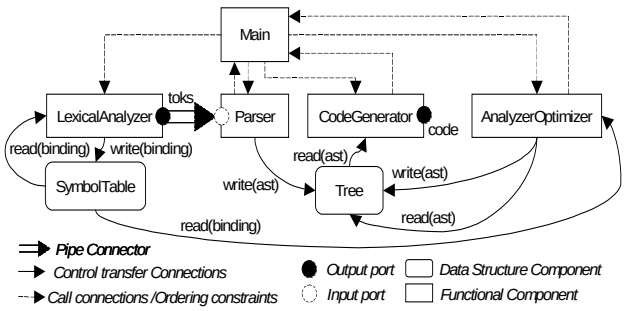


Figure 2: Compiler Architecture: take

Figure 3 depicts the set of significant concerns that define the structural view of the compiler system. It contains six components: LexicalAnalyzer, Parser, AnalyzerOptimizer, CodeGenerator, SymbolTable and Tree, which are all connected via a complex connector, named CompilerConnector. As shown below, the connector plays a central role in this example. It mediates different kinds of communications between the components of the system and encapsulates all the communication paths. The CompilerConnector

also coordinates the interactions among participant components. Therefore, it may enforce a particular communication protocol among the components.

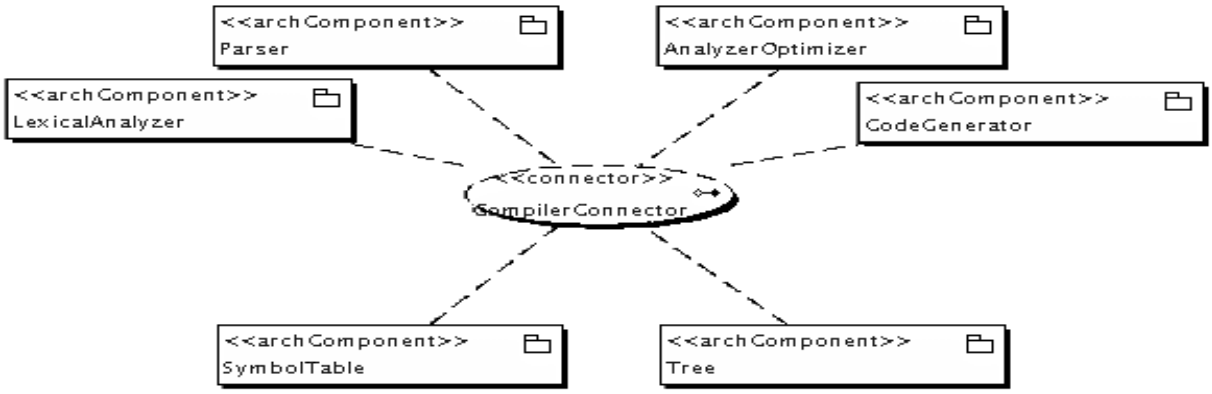


Figure 3: Structural View of the Compiler System

Figure 4 illustrates the static structure of the LexicalAnalyzer component. Its component interface is composed of five interface elements, where each element defines a logical interaction point between the component and its environment. The ExecutionControl interface element provides the operation start with the meaning that another component can activate the LexicalAnalyzer, i.e. starts it by implementing this interface. The MemoryAccessControl interface element requires two operations: read and write. This means that the LexicalAnalyzer requires these operations to be provided by another component. The ControlFlowSignaling interface element declares incoming and outgoing signals necessary to control the execution of the LexicalAnalyzer, while the MemoryFlowSignaling interface element enumerates signals needed for communication with the shared-memory components.

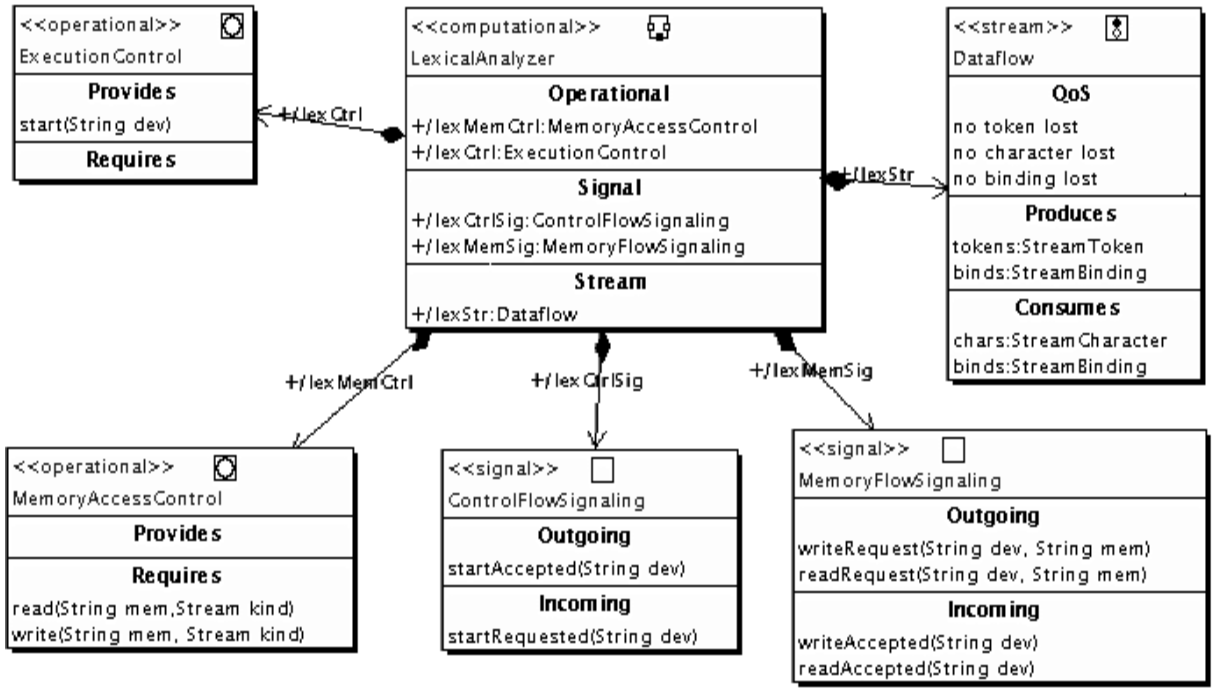


Figure 4: Static Structure Model of the LexicalAnalyzer

Lastly, the Dataflow interface element defines two streams produced by the LexicalAnalyzer, namely a stream of tokens and a stream of bindings, as well as two consumed streams conveying characters and bindings. It is important

to remark that bindings are both produced and consumed by the component, showing the similarity with figure 4, where the LexicalAnalyzer component reads and writes bindings, i.e. produces and consumes them. As shown below, all these interface elements are involved in a composition relationship with the component that realizes them. Furthermore, the interface elements are externally visible parts of the component.

The use of communication-specific interface elements clearly exhibits separation of concerns when defining specialized interaction points (referred to as static ports in the configuration model), since each interface element type is responsible for a particular communication type.

To illustrate a portion of the configuration model of the compiler system, we instantiate the LexicalAnalyzer and Parser components and the simple connectors. The resulting configuration is shown in figure 5, which depicts a part of the configuration model of the compiler system. In figure 5, we can see one instance of the LexicalAnalyzer component and one instance of the Parser component. Each interface element owned by the component is shown as a static port on its boundary. We distinguish three connectors instances, which are used to mediate the communication between components. One connector links the <<operational>> static ports of ExecutionControl together, another relates the <<stream>> Dataflow ports, and another the <<signal>> ControlFlowSignaling ports.

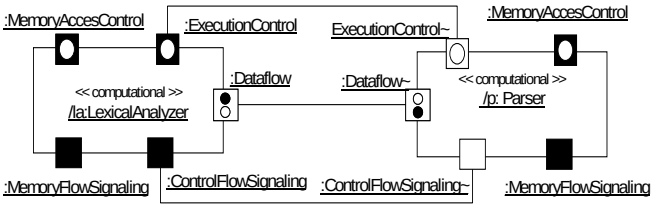


Figure 5: Configuration View of the Compiler System

5 Overview of SADL

This section gives a brief introduction to the concepts of SADL. Figure 6 shows a portion of the architecture description of the compiler_L1 example in SADL. The topmost section of an SADL architectural description is called ARCHITECTURE; it encloses other lower-level SADL sections. We can see that an architecture section is referenced by the identifier compiler_L1. The architecture description given after the ARCHITECTURE keyword includes exchanged data with its environment using input and output ports. The compiler_L1 has an input port, named char_iport, and an output port, called code_oport. char_iport receives a sequence of characters (SEQ(character)), and code_oport sends code data. To apply SADL to definitions that are externally defined, an architecture description must first import them. This is achieved by the using the keyword IMPORTING, indicating where the definitions can be found. In our example, IMPORTING Function FROM Functional_Style tells us that Function is imported from an SADL style named Functional_Style. In order to be imported into an SADL architecture, an SADL definition has to be exported using the EXPORTING statement. For instance, the declaration EXPORTING start specifies that the start function is made available to other architectures wanting to utilize that function.

An SADL architecture description contains three different sections dealing with various aspects of its software architecture, namely COMPONENTS, CONNECTORS and CONFIGURATION. The first and the second sections contain the declaration of the components and connectors, respectively, whereas the third section defines constraints on the configuration of the architectural elements defined in the first and second sections.

The COMPONENTS section contains mainly elements like ARCHITECTURE, Function, Variable and Operation. In SADL, all of those elements are considered as being components. The ARCHITECTURE section allows us to define sub-architectures that can be contained in a higher-level architecture. For instance, in figure 6, lexicalAnalyzerModule is a sub-architecture contained in the compiler_L1 architecture. Note that through this feature, SADL provides a support for modularization.

Functionality of architectures can be expressed through the definition of Function components. As an architecture element, a Function component may have input and output ports through which data can be received or sent. In figure 6, the sub-architecture lexicalAnalyzerModule contains a function called lexicalAnalyzer representing the main functionality of the sub-architecture.

Operation and Function components have similar meanings. The difference between them lies in the fact that the input ports of an Operation are seen as the parameters and the output port as the return value of the operation. However, the number of output ports of a Function component is limited to one.

Variable components are used to hold different types of data and make them available to other components in the sense of shared-memory, which is local to a sub-architecture. One component is only able to keep a single type of data, which means that we need different Variable components for different types of data. For instance, the lexicalAnalyzerModule contains three different Variable components (character-, token- and bindingVariable), the only three that are used by the sub-architecture.

The CONNECTORS section contains the definitions of different kinds of connectors, these are, e.g., Pipe and Enabling_Signal. Connectors enable communication among components. A Pipe connector carries data from an output

port of one component to an input port of another. The transmitted data must be of the same type supported by the related output and input ports. An Enabling_Signal connector mediates signal communication that is likely to occur between two components.

```
IMPORTING Function FROM Functional_Style
...
compiler_L1 : ARCHITECTURE [ chars_iport : Finite_Stream(Character) -> code_oport : Finite_Stream(code)]
BEGIN
  COMPONENTS
    lexicalAnalyzerModule : ARCHITECTURE
       [chars_iport : Finite_Stream(Token), bind_iport: Finite_Stream(Binding) ->
        bind_oport: Finite_Stream(Binding), token_oport : Finite_Stream(Token)]
    BEGIN
       COMPONENTS
          lexicalAnalyzer : Function
                 [chars_iport : Finite_Stream(Token), bind_iport: Finite_Stream(Binding) ->
                  bind_oport: Finite_Stream(Binding), token_oport : Finite_Stream(Token)]
          characterVariable : Variable(Character)
          tokenVariable : Variable(Token)
          bindingVariable : Variable(Binding)
       CONNECTORS
          ...
       CONFIGURATION
          token_read  : CONSTRAINT = Reads(lexicalAnalyzer, tokenVariable)
          token_write : CONSTRAINT = Writes(lexicalAnalyzer, tokenVariable)
          ...
     END
     ...
  CONNECTORS
     tokenPipe : Pipe[Finite_Stream(Token)]
     ...
  CONFIGURATION
     tokenFlow : CONNECTION = Connects(tokenPipe,lexicalAnalyzerModule!token_oport,parserModule!token_iport)
     ...
END
```

Figure 6: Extract of the Level-3 Compiler SADL Specification

The CONFIGURATION section defines the configuration constraints on the previously defined components and connectors. These constraints may state, for instance, which Function or Operation component has read/write access to a Variable component, which component sends a signal, which component receives it, the direction of the data flow between two components, and from which component an Operation is called. We use two different types of statements, namely CONNECTION and CONSTRAINT. The former defines data flow connections and the latter specifies all other kinds of constraints.

6 Mapping ConcernBASE to SADL

This section presents our approach for translating a ConcernBASE architectural description written in UML into a textual form written in SADL.

The mapping consists of 5 steps. The first step identifies all data types utilized in the ConcernBASE architectural description and maps them to SADL. The second step requires that all the architectural components be found and mapped to SADL. The third step requires that all the interface elements of each architectural component be found and mapped to SADL. The fourth step identifies data flow connections and maps them to SADL. And finally, the fifth step puts the pieces together.

6.1 Mapping Data Types

To perform this task, we use an SADL feature that allows SADL styles to be defined anytime [4]. Figure 7 shows an SADL style which defines the data types used in the level-3 compiler (see section 5).

Basically, we define a new style that consists of all data types contained in the current architectural description. To do this, we have to look at every stream interface in the static model of all the components and connectors. Then, we build up the data types list by gathering every data type supported by the different streams. Then, we simply define a new style having the name of the current architecture appended with the suffix Types in a file having the name of the style with the extension ".sadl".

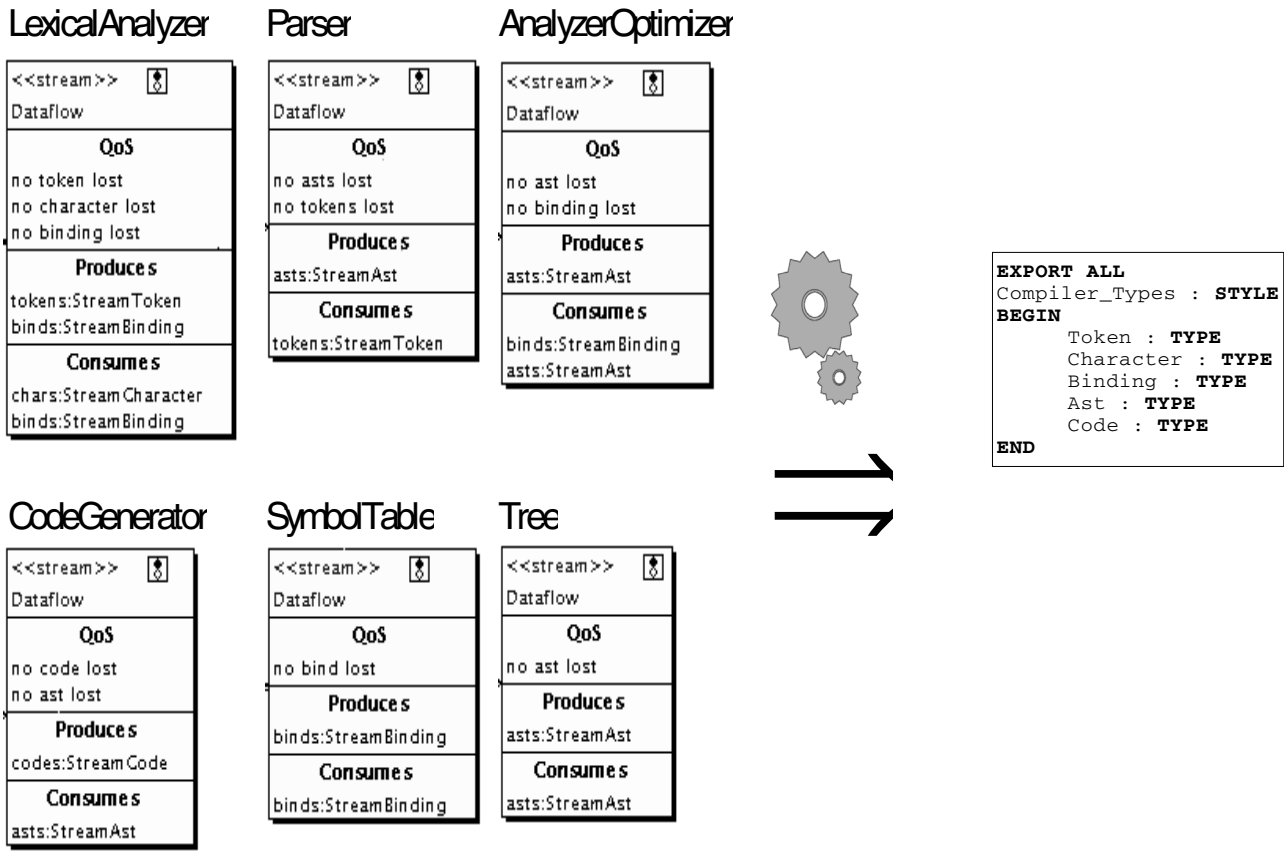


Figure 7: Compiler_Types.sadl

6.2 Mapping Architectural Components

Before mapping ConcernBASE components to SADL, we look at the structural view and identify all the architectural components that are contained in the system.

We translate every architectural component (subsystem) as an SADL sub-architecture with the suffix Module and declare it in the COMPONENTS section of the main architecture. We then declare a Function component with the same name as the component and the same input and output ports. The Function represents the main functionality of the sub-architecture and will be referred to as the sub-architecture's main component. However, this mapping strategy does not exclude that other UML artifacts (for instance, high-level connectors) can be modeled as components. Such artifacts will be discovered during the next steps. Figure 8 shows how the structural view is translated into SADL.

6.3 Mapping Component Interfaces

To translate the component interface, we have to look at its static model. The component interface is composed of three different interface element types, each of which supports a different communication pattern.

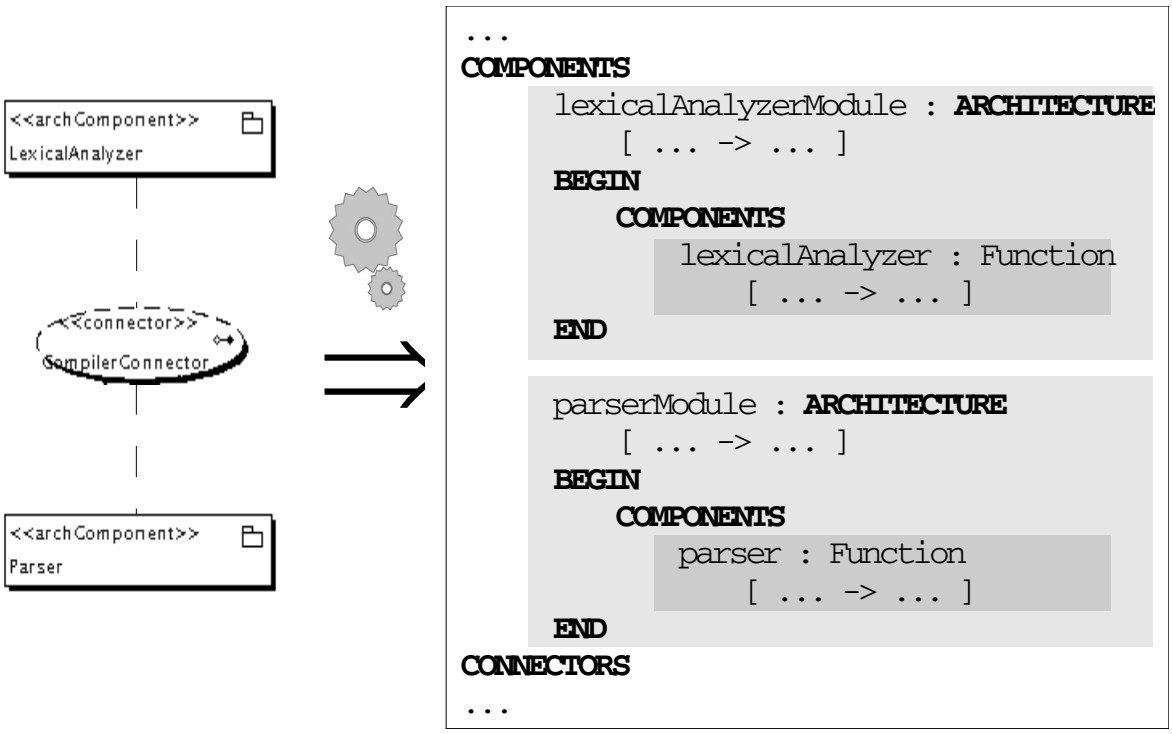


Figure 8: Translating Architectural Components

6.3.1 Stream Interface Type

Clearly, the <<stream>> interface element type is the easiest type to map, since it is equivalent to a SADL port. A stream interface element may produce and consume different kinds of streams, e.g., video and audio streams. Each stream declared in the Produces and Consumes compartments is translated into an output and an input port of the component, respectively. Figure 9 illustrates this idea.

Also, we declare a Variable component in the COMPONENTS section of the sub-architecture for every different type of stream. A Variable component simply holds the data and acts as a shared-memory component within the sub-architecture. Moreover, it should only be accessed by internal components of the sub-architecture that owns it, using Reads/Writes predicates. These are configuration constraints that need to be specified in the sub-architecture itself. The reason for doing so is to differentiate between functional and data-holding concerns of components. In this way, all data consumed by a component is directly stocked into a Variable component dealing with the corresponding data type.

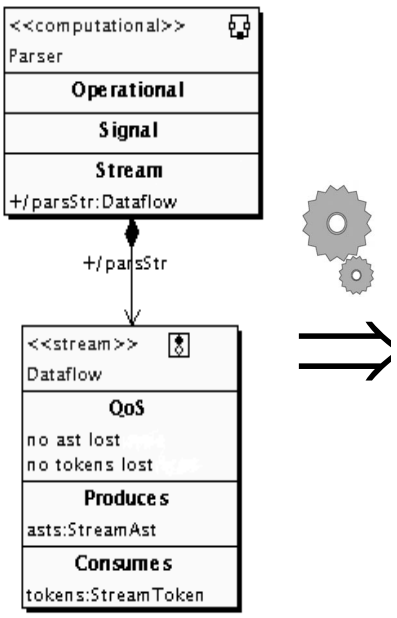
6.3.2 Operational and Signal Interface Types

SADL lacks precise formalism for the definition of operational connectors, i.e. connectors that mediate operation calls between two components. However, the SADL style, Procedural_Style, contains the definition of the Called_From predicate taking the invoked Operation and the calling COMPONENT as parameters. For instance, Called_From(B!start,A) means that the component A calls the operation start implemented by component B. Note that start is declared as an Operation in the COMPONENTS section of the sub-architecture B.

The Outgoing compartments of the <<signal>> interfaces of a component allow us to identify the set of signals defined by that component. We therefore declare the signals in the CONNECTORS section of the sub-architecture representing the architectural component. To retain the behavior, we have to translate the ordering constraints on the signals. To do this we analyze the behavioral model, which provides all information we need to get the correct sequencing of signals. Figure 10 shows the translation of the behavior of a component into SADL with respect to the mediation of signal and operational communication. The static model is helpful for identifying operations and signals, while the behavioral model helps discover the temporal ordering of signals and operation calls.

Furthermore, C1 sends the signal sig1 and enters state B. The component C2 (not shown in the figure) receives sig1 and immediately sends sig2, which is in turn received by C1. Upon reception of sig2, C1 calls the operation op1 and sends the signal sig3. The ordering is translated by means

of SADL predicates (Sender, Receiver, Called_From) indicating the kind of relationship existing between the predicate's arguments. For instance, Sender(c1Module!sig1,c1Module) means that c1Module is the sender of the signal sig1. Outgoing signals are declared within the sub-architecture. The constraints that specify the correct sequencing of the signals are declared in the CONFIGURATION section of the main architecture.




```
...
COMPONENTS

        parserModule : ARCHITECTURE
            [tokens_iport : Finite_Stream(Token)->
             asts_oport : Finite_Stream(Ast)]
        BEGIN
            COMPONENTS
               parser : Function
                  [tokens_iport : Finite_Stream(Token)-
                   asts_oport : Finite_Stream(Ast)]

               tokenVariable : Variable(Token)
               astVariable : Variable(Ast)
            CONFIGURATION
               ...Reads(parser,tokenVariable)
               ...Writes(parser,tokenVariable)
               ...Reads(parser,astVariable)
               ...Writes(parser,astVariable)
        END
CONNECTORS
...
```

Figure 9: Translating stream interface type

Translating the behavior of connectors is another very important thing that has to be taken into account in order to retain the semantics of the source model. ConcernBASE and SADL differ on the fact that connectors may have behavior, too. We cannot specify the behavior of a connector in SADL. In section 6.2, we have mentioned that we may have to create an additional SADL component to represent a ConcernBASE connector with behavior. For instance, in the level-3 compiler, the CompilerConnector is responsible for controlling the execution flow of the components being part of the compiler system. In SADL, we would model this feature as a component that would transfer the control to each component in a sequential manner (see the main component in figure 2). This simply means that we create an SADL sub-architecture for each simple ConcernBASE connector that has behavior. To achieve this, we have to find all state machines of a connector that do not transfer signals and operation calls further. Such an SADL component, standing for a ConcernBASE connector, has no precise functionality and therefore does not own any internal component (Functions, Operation or Variable component). This new component is only responsible for transferring the control to other components, much like a main procedure calling other sub-procedures to delegate different sequential sub-tasks.

6.4 Putting It All Together

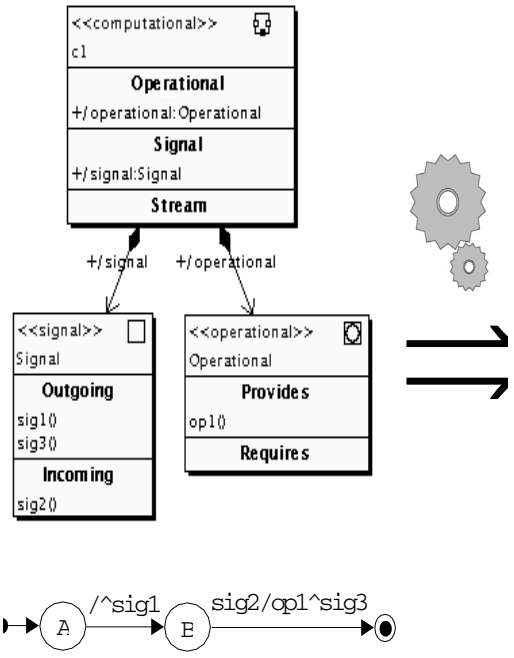
The last thing to do is to add IMPORTING and EXPORTING statements before the declaration of the main architecture as depicted in figure 12. An IMPORTING statement allows the use of architectural elements defined in other specifications and makes them available for the definition of the current architecture and sub-architectures. An EXPORTING statement allows an architecture to make its elements available to other architectural descriptions.

6.5 Mapping Connections

In the SADL formalism, a connection represents a data link between two components. It is further specified as being a CONNECTION constraint relating an output port of a component with an input port of another component via a data connector (e.g., a Pipe).

We have shown how to identify SADL ports in section 6.3.1, and now we show how to relate those ports together to allow data exchange between two components. The only thing we have to do is to look at the configuration model and identify the simple stream connectors between any two components. Figure 11 illustrates this concept by showing that C1 produces a finite stream of characters, C2 consumes this stream, and the connector between the <<stream>> static ports carries it. The connector and the

connection are respectively declared in the CONNECTOR and the CONFIGURATION sections of the main architecture.




```
...
COMPONENTS
        c1Module : ARCHITECTURE
            [ ... -> ... ]
        BEGIN
            COMPONENTS
                c1 : Function
                  [ ... -> ... ]
                op1 : Operation
                  [ ... -> ... ]
            CONNECTORS
                sig1, sig3 : Enabling_Signal
        END
CONNECTORS
CONFIGURATION
        ...Sender(c1Module!sig1,c1Module)
        ...Receiver(c1Module!sig1,c2Module)
        ...Sender(c2Module!sig2,c2Module)
        ...Receiver(c2Module!sig2,c1Module)
        ...Called_From(c1Module!op1,c1Module
        ...Sender(c1Module!sig3,c1Module)
...
```

Figure 10: Translating Behavioral Aspects

7 Tool Support

The ConcernBASE Modeler is an integrated tool for developing architectural descriptions using the ConcernBASE approach (described in section 3). The tool allows one to translate UML architectural models into SADL descriptions, providing at the same time a new and elegant way to supply verification support for UML models using the existing SADL tools. Tool pro-activeness supports the developer in modeling because it actively manages the consistency between different overlapping views. For instance, when the user wants to instantiate a component type in the configuration model, the tool proposes a list of components that have been defined in the structural view. When the user is modeling the behavior of architectural elements by means of state machines, the trigger and call event lists are populated with signals and operations that already exist, i.e. that have been defined in the corresponding interface elements. These features reduce user accidents and errors.

The software is single project-based, which means that it only allows one architecture to be modeled at a given time. One project may contain several model files depicting the architecture. The structural view is shown as a high-level model that can be refined by defining more detailed models; each architectural element declared in the structural view has its own static model and behavioral model in the same file; the configuration structure is also defined as separate model. All models are saved on disk using the standard XMI file format.

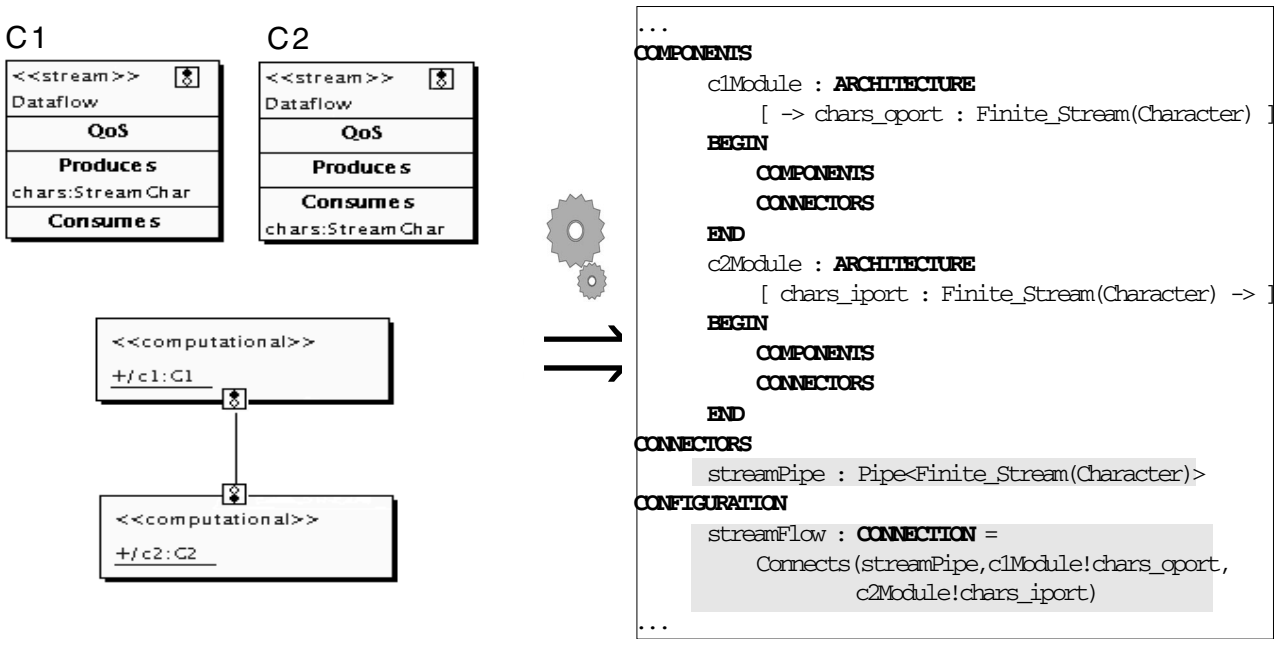


Figure 11: Translating Data Connections

The graphical user interface is simple, usable and intuitive. It has a menu bar that provides different options, a tool bar containing frequently-used functions, a left pane displaying a structured view of the architecture, a right pane allowing one to graphically and easily modify architectural diagrams, and a message pane keeping the user informed of what is going on within the system. The interface is completely event-driven and all resources, i.e. labels, texts, messages, images, etc., are internationalized; this means that the aspect of the interface can be changed and localized without having to rebuild the system. Finally, a complete built-in help system offers information on the system itself, its functionalities, and its application domain (ConcernBASE and SADL).

```
IMPORTING Character,Binding,Ast,Token,Code FROM Compiler_Types
IMPORTING Function FROM Functional_Style
IMPORTING Operation,Called_From FROM Procedural_Style
IMPORTING Sender,Receiver,Before,Enabling_Signal FROM Control_Transfer_Style
IMPORTING Pipe,Finite_Stream FROM Process_Pipeline_Style
IMPORTING Variable,Reads,Writes FROM Shared_Memory_Style
compilerL3 : ARCHITECTURE [ ... -> ... ]
BEGIN
COMPONENTS
      lexicalAnalyzerModule : ARCHITECTURE   [ ... -> ... ]
      BEGIN
          COMPONENTS
               lexicalAnalyzer : Function [ ... -> ... ]
               start : Operation [ ... -> ... ]
               tokenVariable : Variable(Token)
               ...
      END
CONNECTORS
...
```

Figure 12: Putting everything together

8 Summary and Future Work

ADLs provide expressive notations that many architects would like to integrate with UML. Therefore, different strategies have been proposed for mapping ADL constructs into UML. Then again, architectural viewpoints and views, as standardized in IEEE 1471, have been used in various architectural approaches to support the understanding and description of different aspects of the software architecture of systems. In this paper, we have proposed a particular way of establishing a bridge between ADLs, UML and the IEEE 1471. We have called this ConcernBASE and presented a method for translating into SADL specifications the IEEE 1471 concepts implemented in ConcernBASE using UML. The mapping discussed in this work enabled us to make use of SADL verification tools and integrate them with the ConcernBASE Modeler tool. The ConcernBASE approach and the tool supporting it are both undergoing refinement and improvement, but they are already being applied. Although the tool is not yet complete, one can already develop models, translate them to SADL, edit and syntax-check the resulting SADL descriptions and save the models to disk.

In future work, we plan to provide support for simultaneous separation of concerns at multiple levels of abstraction. Further, we plan support for runtime reconfiguration, an important feature that allows one to change dynamically the configuration of a system.

9 Acknowledgement

This work was partially supported by the Defense Advanced Projects Research Agency (DARPA) under contract F30602-00-C-0087. Valentin Crettaz would also like to thank the SRI System Design Laboratory and in particular Robert Riemenschneider for their support. Finally, the authors would like to thank Rich Hilliard for his detailed comments on the conformance of ConcernBASE to IEEE 1471.

References

[1] The Institute of Electrical and Electronics Engineers (IEEE) Standards Board. *Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE-Std-1471-2000)*. September 2000.

[2] M. Kande and A. Strohmeier. *Towards an UML Profile for Software Architecture Descriptions*. UML'2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, S. Kent, A. Evans, B. Selic (Ed.), LNCS (Lecture Notes in Computer Science)

[3] M. Kande and A. Strohmeier. *On The Role of Multi-Dimensional Separation of Concerns in Software Architecture*. Position paper for the OOPSLA'2000 Workshop on Advanced Separation of Concerns. (Online at http://lgl-www.epfl.ch/~kande/Publications/role-of-mdsoc-in-swa.pdf)

[4] M. Moriconi and R. Riemenschneider. *Introduction to SADL 1.0*. SRI Computer Science Laboratory, Technical Report SRI-CSL-97-01, March 1997.

[5] OMG Unified Modeling Language Revision Task Force. *OMG Unified Modeling Language Specification*. Version 1.4 draft, February 2001. http://www.celigent.com/omg/umlrtf/

[6] P.Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. Proceedings of the International Conference on Software Engineering - ICSE'99 (May 1999).

[7] D. Garlan, R. T. Monroe and D. Wile. *ACME: An Architecture Description Interchange Language*. Proceedings of CASCON '97 (1997).

[8] N. Medvidovic and R. N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, Vol. 26, No.1, January 2000.

[9] P. Clements. *A Survey of Architecture Description Languages*. 8th International Workshop on Software Specification and Design, Germany, March, 1996.

[10] D. Garlan and A. Kompanek. *Reconciling the Needs of Architectural Description with Object-Modeling Notations*. In UML 2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, S. Kent and A. Evans (Ed.), LNCS, York, UK, October 2-6, 2000.

[11] O. Weigert (moderator). *Panel: Modeling of Architectures with UML*. In UML 2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, S. Kent and A. Evans (Ed.), LNCS, York, UK, October 2-6, 2000.

[12] P. Tarr and H. Ossher. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Devel-opment. Kluwer, 2000. (To appear.)

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

[14] 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), pp. 209-218, Kyoto, Japan, April 19-25 (1998).

[15] P. B. Kruchten: *The 4+1 view model of architecture*. IEEE Software, 12(6):42-50, (1995).

[16] L. Bass, P. Clements, R. Kazman: *Software Architecture in Practice*. Addison-Wesley (1998).

[17] C. Hoffmeister, R. Nord, D. Soni: Applied Software Architecture. Addison-Wesley (1999).

[18] P. Clements, R. Kazman, M. Klein: *Evaluating Software Architectures*.Addison-Wesley (2002).

[19] R. Hilliard: *Viewpoint modeling*. ICSE Workshop on Describing Software Architecture with UML (2001).

[20] ConcernBASE: http://lglwww.epfl.ch/research/concernbase/index.html

[21] V. Crettaz, M. M. Kandé, S. Sendall and A. Strohmeier: *Integrating the ConcernBASE Approach with SADL.* UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, Fourth International Conference, Toronto, Canada, October 1-5, Martin Gogolla (Ed.), LNCS (Lecture Notes in Computer Science), no. 2185, Springer Verlag, 2001, pp. 166-181.