Integrating the ConcernBASE Approach with SADL

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

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

ABSTRACT We describe ConcernBASE, a UML-based approach that is an instantiation of the IEEE's Conceptual Framework (Std 1471) for describing software architectures. We show how the approach supports advanced separation of concerns in software architecture by allowing one to identify and define multiple viewpoints, concern spaces and views of an architecture. Our work focuses on integrating the ConcernBASE approach with the Structural Architecture Description Language (SADL) in order to make the verification capabilities of SADL available to those who develop in UML. The result is a UML profile for structural description of software architecture. The paper also presents a prototype tool that supports this UML profile.

KEYWORDS Software Architecture, Unified Modeling Language, UML, Structural Architecture Description, SADL, Advanced Separation of Concerns.

1 Introduction

The fact that numerous software systems are becoming increasingly complex, distributed, and deployed in heterogeneous environments leads us to think that modeling the software architecture of a system is a very important part of the development life cycle. But, software architecture modeling should not be seen as a separate activity that is limited to a particular "phase" of the software life cycle, as one might deduce from the limited scope of existing architecture description languages (ADLs) [7][8][9]. ADLs have the advantage of being mathematically defined and deeply rooted in formal methods, but also the disadvantage of lacking the flexibility for modeling systems from various viewpoints. In addition, due to their formal nature they can be hard to understand and to use [2].

Unlike ADLs, the Unified Modeling Language (UML) is a widely used general-purpose language, which provides a large number of well-known techniques and concepts for modeling various kinds of software artifacts from different perspectives. Unfortunately, UML, in its current state, is not sufficient for an explicit software architecture description [2][10][11]. To gain the benefit of software architecture description with UML, the core UML needs to natively define some key ADL-concepts as first-class modeling elements, such as connectors and styles.

In previous work, we proposed a UML-based approach to software architecture descriptions, which focused on extending the UML by incorporating key abstractions found in most existing ADLs into a profile [2].

This work has been integrated into a new approach called ConcernBASE[1]. ConcernBASE is centered around the principle of *multi-dimensional separation of concerns* (MDSOC). MDSOC is an advanced form of separation of concerns that allows one to identify and simultaneously separate multiple kinds of software concerns, including

---

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

principles for composition and decomposition of those concerns [6]. However, unlike other approaches based on MDSOC [12], ConcernBASE uses the standard UML notation and addresses some fundamental limitations of UML for supporting software architecture descriptions.

The Structural Architecture Description Language (SADL) [4] has been developed to support the specification of structural aspects of complex software systems. Unlike other ADLs, e.g. Wright [13], it also focuses on the refinement of high-level system structures. In addition, SADL provides an assortment of tools that support both refinement and verification of different structural aspects of complex software.

This paper discusses how to map ConcernBASE architectural descriptions, written in UML, to SADL architectural descriptions, making available the verification capabilities of SADL tools for ConcernBASE. In addition, we present a UML-based tool that supports the ConcernBASE approach and its integration with SADL, and implements the UML profile for structural architectural descriptions.

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

2 The ConcernBASE Approach

ConcernBASE is a software architecture approach that aims at providing support for software (component) development and software (system) construction, by combining the capabilities of both MDSOC and UML. It addresses techniques for developing individual software components and particularly focuses on understanding, reasoning and specifying mechanisms for gluing those components together.

ConcernBASE is a UML-based instantiation of the IEEE's Recommended Practice for Architectural Description (Std 1471) [1], that is augmented with support for advanced separation of concerns [3]. Thus, it supports 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. According to the IEEE Std 1471, *Concerns* 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 include system considerations such as performance, reliability, security, distribution, and evolvability [1].

Throughout the approach, we take the premise that *software architecture is multidimensional*. That is, when constructing complex software, an architect represents the system in many different ways in order to be able to understand, communicate and reason about its high-level properties, from different perspectives or viewpoints. Each way of representing the system may be considered as a different view of the architecture of the system, that takes into account *multiple dimensions* of concern (i.e., different kinds of architectural concerns).

In the IEEE's conceptual framework for architectural description (IEEE Std-147), a *viewpoint* is a specification of the conventions for constructing and using a view. While this is an important and precise definition that helps understand how to separate different concerns, it does say how those concerns can be identified, encapsulated and represented in architectural views. In ConcernBASE, viewpoints are characterized by two essential architectural abstractions that we refer to as concern space and projection.

Concern space represents a conceptual repository that contains all relevant concerns related to a particular viewpoint. It allows us to structure different concerns into different categories (dimensions), to specify the relationships between these categories and maintain changes in the concern structure. Thus, a concern space can be considered as a "multi-dimensional model of system considerations" that pertains to a software architect from a particular perspective. A *projection* defines the relationship between a viewpoint, a concern space and a view. It is an architectural abstraction that specifies how to transform a set of concerns (sub-concern space) into a specific representation (view) relative to a particular perspective (viewpoint). Different projections along different dimensions of concern result in different views. A *view* of the software architecture of a system is a partial architecture description of that system that may consist of one or architectural models. Finally, the *architecture description* of the whole system may be considered as a set of different architectural views.

Without the notion of concern space, the definition of a viewpoint language, as proposed by the IEEE Std-1471, can be very difficult, since it becomes quickly unclear what the viewpoint language should define. Once we have the notion of concern space, one can use some UML extensions mechanisms to define a UML profile that represents the viewpoint language. In this case, the concrete syntax and semantics of the concern space will be fixed by the UML profile.

To discuss the mapping between ConcernBASE and SADL, we briefly present a ConcernBASE structural viewpoint and SADL, then we describe the mapping between them. Finally, we discuss some issues related the tool support.

2.1 Structural Viewpoint

The structural viewpoint is a particular ConcernBASE viewpoint that specifies the rules for constructing and using some structural views of the architecture of system. In addition, it defines a structural concern space, a UML Profile for structural architecture description and specifies three kinds architectural projections: a static structure projection, a behavioral projection and a configuration projection. These projections allows one to create three corresponding views, which together build the architecture description of the system.

2.2 Structural Concern Space

The structural concern space is the most abstract representation of all significant 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 concern space 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. A component represents a particular UML subsystem. Its type is defined by the UML stereotype <<archComponent>> that inherits properties of both a UML Class and a UML Package. In contrast, a connector represents a special kind of UML Collaboration, in which participant components are omitted. Instead, they are represented by connection points (their interactions points belong to the connector)[2].

2.3 Static View

The static view describes the static models 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 usu-

ally 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 computation 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 3 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 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 to 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.

2.4 Behavioral View

The behavioral view 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.

2.5 Configuration View

The configuration view 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 instanti-

---

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.

ations 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 be then 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.

3 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 1 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 1 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 them 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.
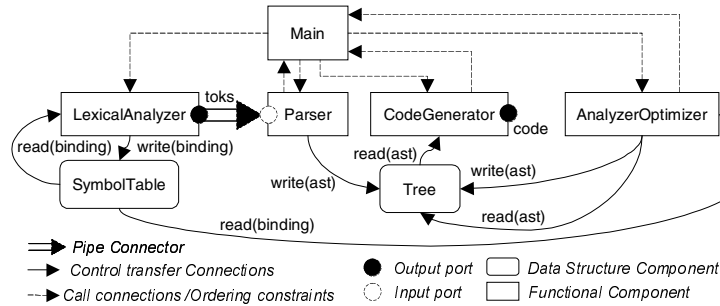
Fig. 1. Compiler Architecture: taken from [3]

Figure 2 depicts the set of significant concerns that define the structural concern space of the compiler system. It contains six components: LexicalAnalyzer, Parser, AnalyzerOptimizer, CodeGenerator, SymbolTable and Tree, which are connected together by 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 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
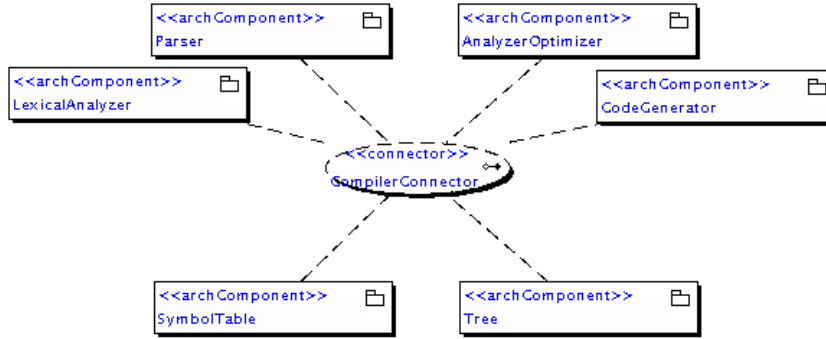
Fig. 2. Structural Concern Space Model of the Compiler System

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 the communication with the shared-memory components.
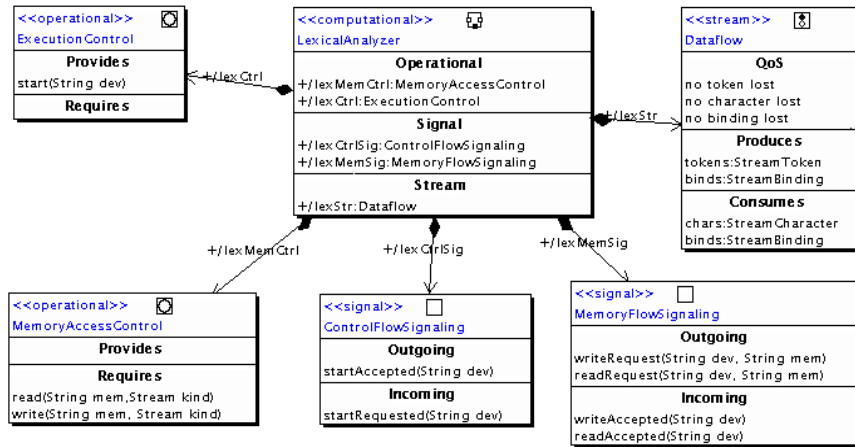

Fig. 3. Static Structure of the LexicalAnalyzer

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 2, 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 view), since each interface element type is responsible for a particular communication type.

To illustrate a portion of the configuration view of the compiler system, we instantiate the LexicalAnalyzer and Parser components and the simple connectors. The resulting configuration is shown in figure 4, which depicts a part of the configuration view of the compiler system. In figure 4, 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.
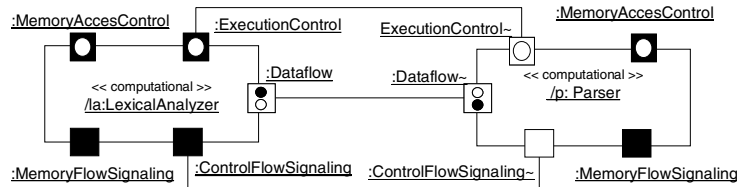

Fig. 4. Configuration View of the Compiler System

4 Overview of SADL

This section gives a brief introduction to the concepts of SADL. Figure 5 shows a portion of the architecture description of the compiler_L1 example in SADL. The top-most section of an SADL architectural description is called ARCHITECTURE; it encloses other lower-level SADL section. 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 use 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 5, 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 5, 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

```
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
```

Fig. 5. Extract of the Level-3 Compiler SADL Specification

the output port as the return value of the operation. However, the number of output ports of an Operation 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 connectors, these are, e.g., kind 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.

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.

5 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 map 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.

5.1 Mapping Data Types

To perform this task, we use an SADL feature that allows SADL styles to be defined anytime [4]. Figure 6 shows an SADL style which defines the data types used in the level-3 compiler (see section 4).

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 view 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".
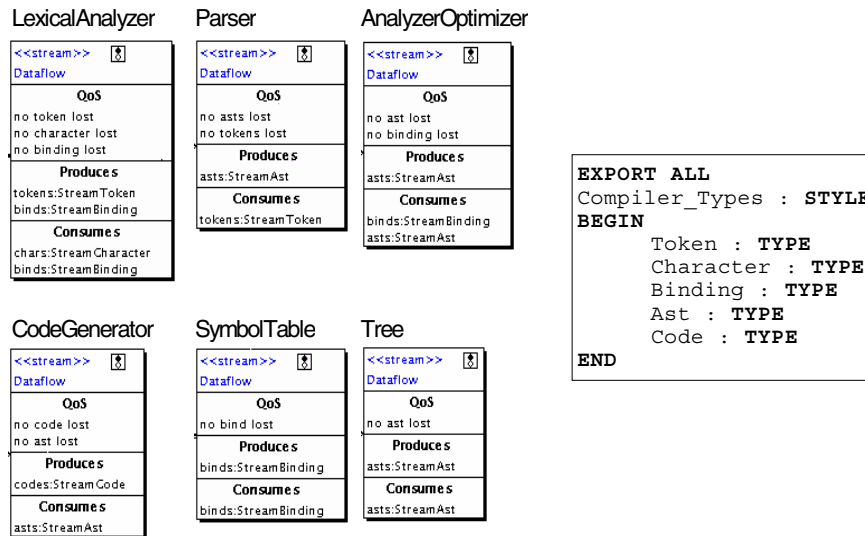

Fig. 6. Compiler_Types.sadl

5.2 Mapping Architectural Components

Before mapping ConcernBASE components to SADL, we have to look at the structural concern space and identify the architectural components that are contained in the system. It is possible that other UML artifacts (for instance, high-level connectors) might have to be modeled as components. Such artifacts will be discovered during the next steps.

We translate every architectural component (subsystems) as an SADL sub-architecture with the suffix Module and declare them in the COMPONENTS section of the main architecture. We then declare a Function component with the same name as the component
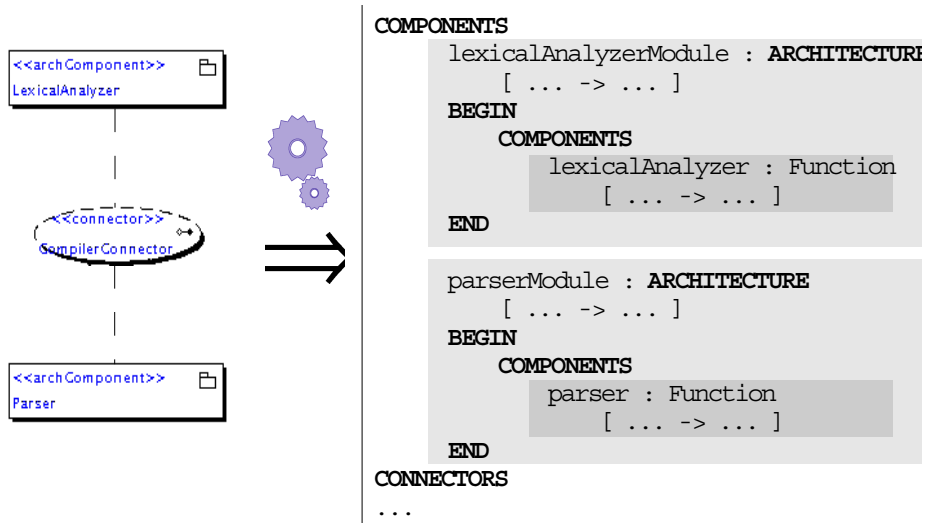
Fig. 7. Translating Architectural Components

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. Figure 7 shows how the structural concern space is translated to SADL.

5.3 Mapping Component Interfaces

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

5.3.1 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 are translated into an output and an input port of the component, respectively. Figure 8 illustrates this idea.

Also, we to 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 owning sub-architecture 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
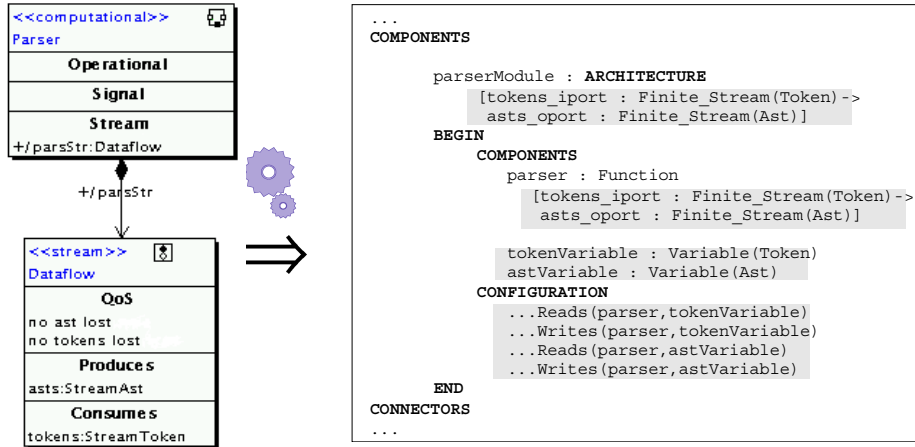
.

Fig. 8. Translating stream interface type

5.3.2 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.
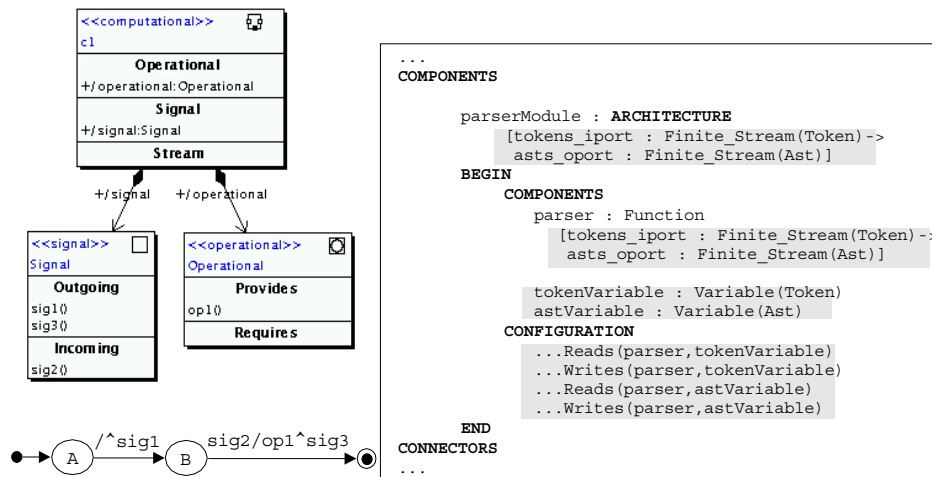

Fig. 9. Translating Behavioral Aspects

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 view, which provides all information we need to get the correct sequencing of signals. Figure 9 shows the translation of the behavior of a component into SADL with respect to the mediation of signal and operational communication. The static view is helpful for identifying operations and signals, while the behavioral view 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.

Another very important thing that has to be taken into account in order to retain the semantics of the source model is to translate the behavior of connectors. Concern-BASE and SADL differ on the fact that connectors may have behavior, too. We cannot specify the behavior of a connector in SADL. In section 5.2, 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 1). 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.

5.4 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 5.3.1, 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 view and identify the simple stream connectors between any two components. Figure 10 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.
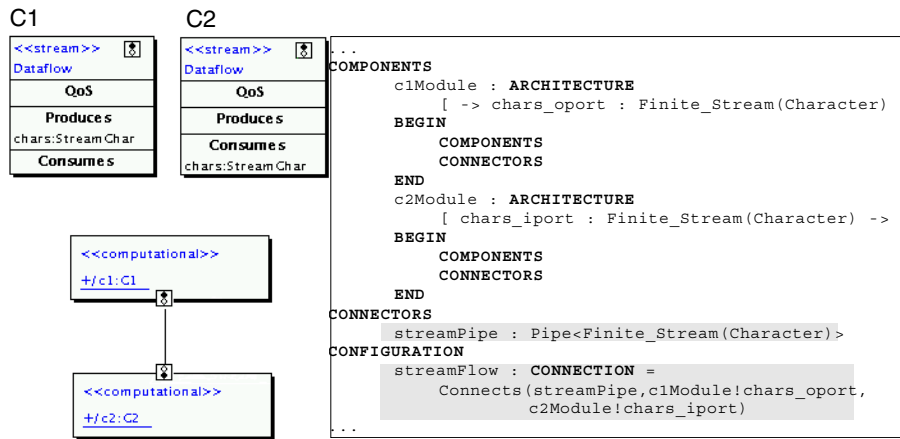
Fig. 10. Translating Data Connections

5.5 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 11. 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.

```
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
...
```

Fig. 11. Putting everything together

6 Tool Support

The ConcernBASE Modeler is an integrated tool for developing architectural descriptions using the ConcernBASE approach (described in section 2). 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 proactiveness 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 view, the tool proposes a list of components that have been defined in the structural concern space. 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 concern space is depicted by one model; each architectural element declared in the structural concern space model has its own static and behavioral view models in the same file; finally, the configuration view is defined by one model. All models are saved on disk using the standard XMI file format.

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).

7 Conclusion and Future Work

In this paper, we have presented the ConcernBASE approach and a method for translating ConcernBASE models into SADL specifications. 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 young. Both are 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. Support for dynamic reconfiguration, an important feature that allows one to dynamically change the configuration of a system, is planned as future work.

8 Acknowledgement

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 Development. Kluwer, 2000. (To appear.)

[13] Allen R. *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).