Introduction to SADL 1.0
A Language for Specifying Software Architecture Hierarchies*

Mark Moriconi and R. A. Riemenschneider
Computer Science Laboratory
SRI International

Technical Report SRI–CSL–97–01
March 1997

Abstract

This report describes a new architecture description language called SADL, intended for the expression of software architecture hierarchies that are to be analyzed formally. The SADL language can be used to specify both the structure and the semantics of an architecture, but our main focus has been on the former. The interesting features in SADL include support for explicit mappings between architectures, generic architectures, architectural styles (including well-formedness constraints), and architecture refinement patterns that provide routine solutions to common design problems. SADL is programming language independent, but can be tailored to model programs in most conventional programming languages.

This document describes SADL version 1.0. Related technical papers and the public SADL version 1.0 tools are available at web site

`http://www.csl.sri.com/sadl`

Language updates and releases of new or updated tools will be made available at this site.

*This research was supported by the Defense Advanced Research Projects Agency under Rome Laboratory contracts F30602–93–C–0245 and F30602–95–C–0277.

Contents

List of Figures

1 SADL and Software Architecture

The term "software architecture" has been used to refer to various aspects of system structure and system behavior. Our research is focused more narrowly on specifying and reasoning about ***structural properties*** of software systems — namely, the form of an architecture and the important structural relationships between and among architectures that are intended to describe the same software family. Semantic constraints are introduced only to support structural reasoning. Our work complements the more traditional research in formal methods that is concerned with behavioral properties of systems.[1]

Traditionally, software architectures have been described only informally using the ubiquitous box-and-arrow diagrams, severely limiting their utility in practical system development. The SADL language[2] provides a precise textual notation for describing software architectures while retaining the intuitive box-and-arrow model. To this end, we make a clean distinction between several kinds of architectural objects (e.g., components and connectors) and make explicit their intended uses.[3] The SADL language provides facilities not only for specifying architectures, but also for specifying well-formedness constraints on particular classes of architectures. For example, it is possible to specify not only the kinds of components and connections in, say, client-server and blackboard systems, but also the intended configurations of the components and connections. SADL can be used to describe the architecture of a specific system or a family of systems.

A unique aspect of our research centers around the representation of and reasoning about ***architecture hierarchy***. A ***vertical hierarchy*** serves to bridge the gap between abstractions in architectures and the more primitive structural concepts in conventional programming languages. Each level in a vertical hierarchy typically is described using a different vocabulary, reflecting a change in representation. For example, a pipe-and-filter architecture would be described using a different vocabulary than an event-based architecture. A ***horizontal hierarchy*** is analogous to "bubble decomposition" in dataflow modeling, where the same vocabulary is used to describe every level in the decomposition.

The SADL language is intended to be used in describing both vertical and horizontal hierarchy and in relating different levels of representation by means of mappings. Precise mappings enable inter-level traceability. They also make it possible to reason about the relationship among architectures in a hierarchy and the relation between the hierarchy and its implementation. For example, we can determine whether the architectural objects in one architecture are present in the other even if there is a change in representation. We also can show that, if a particular communication path is not allowed in one architecture, it is not allowed in others in the hierarchy. An architecture hierarchy may describe a specific system or a family of systems.

[1] A number of existing formalisms can be used to define the behavior of components and the communication protocols in an architecture. Standard proof techniques can be used to reason about the behavior of an architecture and to check whether two architectures conform with respect to behavior.

[2] The acronym SADL stands for *Structural Architecture Description Language*.

[3] Some other models of architecture blur, or even eliminate, the distinction between components and connectors. We believe that this leads to a less familiar, and possibly less intuitive, model of architecture.

SADL can be used to specify the following elements of architectural design:

- **Specific architectures**. An architecture in which every architectural object has a particular name and type.
- **Generic architectures**. An architecture can have parameters that denote an indeterminate type or object. This makes it possible, for example, to specify an architecture that contains an arbitrary number of interfaces and connections. A specific architecture is an instance of a generic architecture.
- **Mappings**. Two architectures are related by a syntactic interpretation mapping, part of which is architecture-specific and part of which applies to any architecture of a certain class.
- **Architectural styles**. Styles encode closely related, reusable design information. A style consists of a vocabulary of object types and constraints on how objects of those types must be related in specific architectures. Examples of architectural styles include pipe and filter, client server, and dataflow.
- **Architecture refinement patterns**. Patterns represent reusable solutions to design problems. They are much smaller than patterns studied in the object-oriented research community and are compositional with respect to a useful property of mappings.

SADL patterns support ***structural refinement***, the systematic translation of one architecture into another architecture containing different architectural concepts. An example is the realization of a collection of interacting clients and servers in terms of an architecture containing only procedures and procedure calls. In general, a structural refinement may require the introduction of behavioral constraints. In fact, we have studied several systems for which desirable structural refinements are correct (in a precise mathematical sense) only if certain constraints are placed on system behavior. Behavioral constraints are expressed in terms of predicate symbols that are axiomatized in the SADL constraint language.

The subsequent sections present an overview of the SADL language, illustrate the basic features of the language, and give the SADL grammar. Most SADL features are illustrated by means of a multi-level compiler architecture, which was chosen because it is simple and familiar. The use of generics is illustrated by means of a client-server architecture in which the server is connected to an arbitrary number of clients.

After reading this document, you should be able to read and write SADL descriptions of most architectures, mappings, architectural styles, and refinement patterns. The mathematical underpinnings for our approach to hierarchy can be found in two publications [1, 2]. The purpose of this document is to explain the SADL language and how to use it. It is not intended to be a formal treatment of the SADL semantics.

2 Overview of SADL

2.1 Syntax

The traditional box-and-arrow diagrams are represented in SADL as follows: boxes correspond to components, arrows correspond to connectors, and the locations where arrows are attached to boxes correspond to ports. This basic model is extended to include architectural styles, heterogeneous architectures, semantic constraints, and mappings between architectures. The entire SADL framework is formally defined and can be used to support formal reasoning about individual and multiple architectures.

SADL can be used to represent the following architectural elements.

1. **Architecture**. An architecture is a, possibly parameterized, collection of the following items.

   (a) **Component**. A component represents a locus of computation or a data store. The various types of components include a module, process, procedure, or variable. A component has a name, a type (a subtype of type `COMPONENT`), and an interface, the ports of the component. A *port* is a logical point of interaction between a component and its environment. A port has a name, a type, and is designated for input or output.

   (b) **Connector**. A connector is a typed object (a subtype of type `CONNECTOR`) relating ports. Every connector is required to accept values of a given type on one end and to produce output values of the same type on the other.

   (c) **Configuration**. A configuration constrains the wiring of components and connectors into an architecture. A configuration can contain two kinds of elements.

      i. **Connections**. A connection associates type-compatible connectors and ports.

      ii. **Constraints**. Constraints are used to relate named objects or to place semantic restrictions on how they can be related in an architecture.[4]

2. **Mapping**. A mapping is a relation that defines a syntactical interpretation from the language of an abstract architecture to the language of a concrete architecture.

3. **Architectural style**. A style consists of a vocabulary of design elements, well-formedness constraints that determine how they can be used, any semantic constraints needed for refinement, and a logical definition of the semantics of the connectors associated with the style. A constraint is declarative and might say, for example, that clients initiate communication with servers, but not vice versa. A given architecture may be *homogeneous*, involving one style, or *heterogeneous*, involving multiple styles.

[4]Components can be related directly with constraints, but only interfaces can be related with connectors. This is useful for abstract modeling in which the specifics of interfaces are unimportant, and for low-level modeling in which it is desirable to closely model the target programming language.

4. **Refinement pattern**. A refinement pattern consists of two architecture schemas, an association of the objects in the two schemas, and possibly constraints on one or both schemas. An instance of a pattern is formed by matching schema variables against the appropriate portions of SADL specifications.

Components, interfaces, connectors, and constraints are treated as ***first-class objects*** — i.e., they are named and typed objects that can appear as parameters. They can be refined into (decomposed, aggregated, or eliminated) objects in more concrete architectures.

SADL is *programming language independent*, intended for both abstract and concrete modeling of system architectures. The SADL type system allows structuring by means of the subtype relation, but does not provide explicit support for the object-oriented version of inheritance that involves overriding. Thus, SADL architectures can be refined directly into implementations constructed using most conventional programming languages. A programming language with overriding, such as C++ or Java, can be used in one of two ways. The lowest-level architecture in a SADL hierarchy contains components that are considered primitive with respect to that level of detail. It is not necessary to model overriding in the architecture provided we do not want to prescribe how to build these primitive components. On the other hand, if one wants to specify in the architecture exactly how to build a component from existing classes, it is necessary to introduce a new subtype relation.[5]

2.2 Type System

The SADL type system has several features particularly useful for describing architectures.

- SADL is *algebraically extensible* in that it is possible to introduce new types that are not derived from a predefined collection of base types. This is useful in defining new architectural styles, which typically contain new types of components and connectors.

- SADL is *polymorphic* in that types can be passed as parameters.

- SADL supports *subtyping* as a means of defining new architectural objects within a particular class of objects. For example, the declaration

  ```
  Client: TYPE <= COMPONENT
  ```

  says that `Client` is a subtype of the SADL built-in type `COMPONENT`. Similarly, the declaration

  ```
  CS_Channel: TYPE <= CONNECTOR
  ```

  says that `CS_Channel` (which stands for "client-server channel") is a subtype of `CONNECTOR`. In addition, it is possible to create a new type consisting of all members of an existing type that have some property. For example, the declaration

[5]A relation for overriding can be axiomatized in SADL, but an axiomatization is not provided in its current library.

```
Local_Client: TYPE = { c: Client | Local(c) }
```

says that `Local_Client` is the subtype of `Client` consisting of all clients that satisfy the predicate `Local`.

- SADL supports ***type constraints***. For example, a general ***type compatibility constraint*** on all components and connectors is stated informally as follows:

  > If connector c carries values from port p_1 of one component to port p_2 of another component, then the declared type of p_1 must be a subtype of the declared type of c, which in turn must be a subtype of the declared type of p_2.

  More specialized constraints may be associated with connector and component subtypes. For instance, we might specify that members of the type `Synchronous_CS_Connector` can only link a server to members of `Local_Client`.

Type checking in SADL is a matter of showing that type constraints are satisfied. Type correctness is, in general, undecidable because of the flexibility allowed in defining subtypes.[6] But in most cases of practical interest, type correctness can be efficiently decided.

2.3 Formal Semantics

A SADL architecture denotes a logical theory in an extended first-order logic that includes numerical quantifiers — often called ω-logic. This logic is equivalent in power to weak second-order logic. Ordinary first-order logic is not powerful enough for describing certain properties of generic architectures. Examples include "connectedness" and "finiteness."

A SADL mapping determines an ***interpretation mapping*** between two ω-logic theories. A mapping can be analyzed for various properties. For example, we would want to know that it is a ***theory interpretation***. We argue in [2] that it should also be ***faithful*** to prevent unintended connections from appearing in an implementation.

Meaning is assigned to connectors using axioms in the SADL constraint language. The correctness of these meaning constraints can be checked by showing that they are true in a mathematical model of the intended semantics of the connectors. Such models can be expressed in any of several conventional formalisms. In [1] we demonstrated the use of conventional formalisms CSP and TLA (Temporal Logic of Actions) for this purpose.[7]

[6] Consider subtypes of the form

$$\{x : T \mid C(x)\}$$

and note that whether a given member of type T satisfies $C(x)$ may be impossible to effectively determine.

[7] The TLA semantics was also used to establish the key safety and liveness relationships between certain abstract and concrete connectors.

3 Overview of Compiler Architecture

Consider the standard dataflow architecture for a compiler that is depicted at the top of Fig. 1. The diagram is intended to convey an intuitive feel for the architecture; it is not a formal description of the architecture. Boxes denote functional components and arrows denote directional dataflow between ports. The labels on arrows denote types or value domains. A value cannot be transmitted between ports unless its type is compatible with the types of the ports.

The dataflow model of the compiler fixes its functional units, their interfaces, and the direction, source, and destination of all of its flows. We assume that, if a fact is not explicit in (or deducible from) the dataflow model, it is not intended to be true of it. For example, we assume that the optimizer is not intended to transmit data to the parser because the dataflow is only one-way. This "completeness assumption" has important practical and theoretical consequences that are beyond the scope of this report. For present purposes, it is enough to observe that, without this assumption, a system developer could easily be misled into believing facts about a system architecture that are not true.[8] (The completeness assumption for architectures can be formalized in terms of theory interpretations that are faithful [2].)

A more concrete architecture intended to implement the dataflow model of the compiler is depicted at the bottom of Fig. 1. The concrete architecture is a hybrid that implements the dataflow style in terms of pipe-filter, batch-sequential, and shared-memory styles. Abstract-level signatures have been changed, dataflow connectors have been implemented in several ways, new components are introduced, and precedence relations are added to preserve the original flows in the presence of shared-memory communication.

4 Level 1 of Compiler Architecture

A SADL specification of the top-level compiler architecture is contained in Fig. 2. The first line

```
compiler_L1: ARCHITECTURE
```

says that `compiler_L1` is an architecture. The second line

```
[char_iport: SEQ(character) -> code_oport: code]
```

says that `compiler_L1` accepts a sequence of characters and produces code.[9] The sole input port is `char_iport`, and `code_oport` is the sole output port. A port is the architectural analogue of an input or output formal parameter of a procedure.

The next few lines of the top-level compiler architecture

[8] It is important to note that the completeness assumption does not preclude the use of "don't care" elements in an architecture. See Sec. 9.

[9] SADL is a case-sensitive language. `ARCHITECTURE` and `SEQ`, like other SADL built-ins, are written in upper case. Identifiers imported from a style are written, by convention, with an initial upper case letter only. Identifiers unique to a particular specification, such as `compiler_L1` and `char_port`, are conventionally written in lower case.

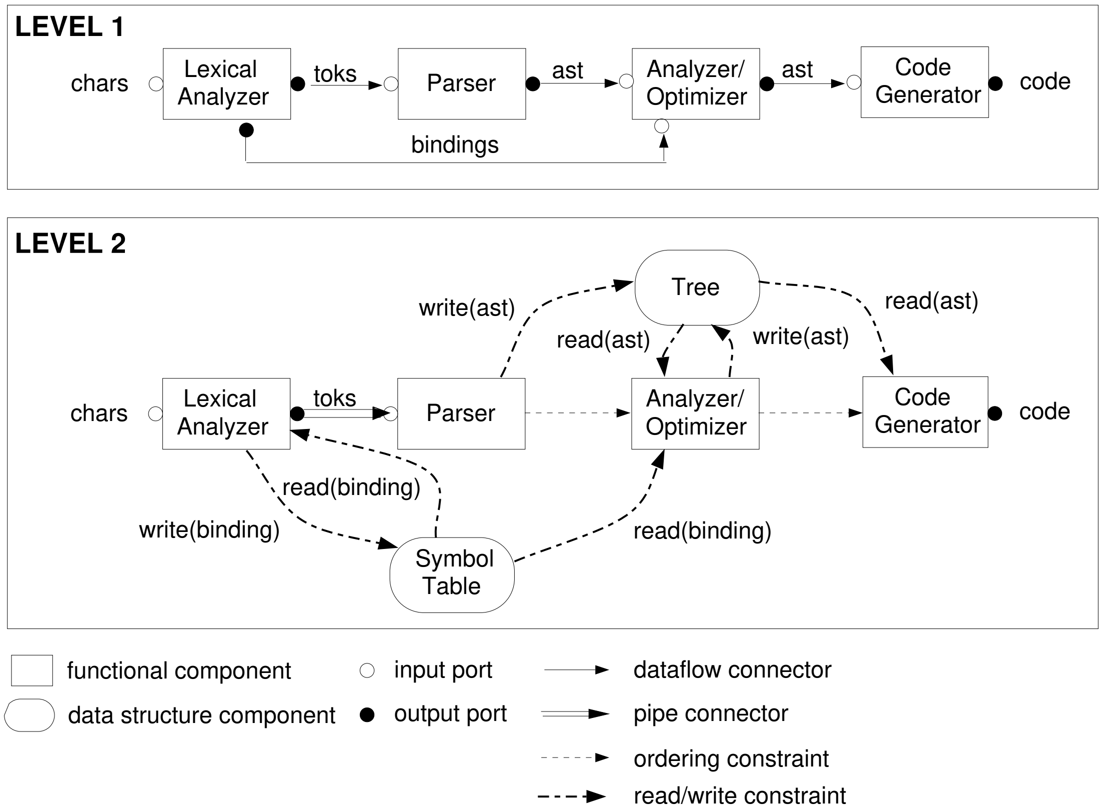

Figure 1: Two Architectures for a Compiler

```
compiler_L1: ARCHITECTURE
    [char_iport: SEQ(character) -> code_oport: code]
  IMPORTING character, code, token, binding, ast FROM compiler_types
  IMPORTING Function FROM Functional_Style
  IMPORTING Dataflow_Channel, Connects FROM Dataflow_Style
BEGIN
COMPONENTS
  lexical_analyzer: Function
     [char_iport: SEQ(character)
         -> token_oport: SEQ(token), bind_oport: SEQ(binding)]
  parser: Function
     [token_iport: SEQ(token) -> base_ast_oport: ast]
  analyzer_optimizer: Function
     [base_ast_iport: ast, bind_iport: SEQ(binding)
         -> full_ast_oport: ast]
  code_generator: Function [full_ast_iport: ast -> code_oport: code]
CONNECTORS
  token_channel:    Dataflow_Channel<SEQ(token)>
  bind_channel:     Dataflow_Channel<SEQ(binding)>
  base_ast_channel: Dataflow_Channel<ast>
  full_ast_channel: Dataflow_Channel<ast>
CONFIGURATION
  token_flow: CONNECTION
    = Connects(token_channel, token_oport, token_iport)
  bind_flow: CONNECTION
    = Connects(bind_channel, bind_oport, bind_iport)
  base_ast_flow: CONNECTION
    = Connects(base_ast_channel, base_ast_oport, base_ast_iport)
  full_ast_flow: CONNECTION
    = Connects(full_ast_channel, full_ast_oport, full_ast_iport)
END compiler_L1
```

Figure 2: Level 1 of Compiler Architecture in SADL

```
IMPORTING character, code, token, binding, ast FROM compiler_types
IMPORTING Function FROM Functional_style
IMPORTING Dataflow_Channel, Connects FROM Dataflow_style
```

say that the type predicates `character`, `code`, etc., are imported from another specification, called `compiler_types`, that `Function` is imported from the style `Functional_style`, and that `Dataflow_Channel` and `Connects` are imported from `Dataflow_style`. In general, an architecture may export items as well. In this specification, the internal architecture is not made visible to users of the compiler.

If we were to look in `Functional_style`,[10] we would find the declaration

```
Function: TYPE <= COMPONENT
```

which says that `Function` is a subtype of the SADL built-in type `COMPONENT`. Similarly, in `Dataflow_style` we would find

```
Dataflow_Channel: TYPE <= CONNECTOR
```

which says that `Dataflow_Channel` is a subtype of `CONNECTOR`, and

```
Connects: PREDICATE(3)
```

which says that `Connects` is a ternary predicate.[11] `Dataflow_style` also contains constraints, such as

```
connects_argtype_1: CONSTRAINT =
  (/\ x)(/\ y) (/\ z) [ Connects(x, y, z) => Dataflow_Channel(x) ]
connects_argtype_2: CONSTRAINT =
  (/\ x)(/\ y) (/\ z) [ Connects(x, y, z) => Outport(y) ]
connects_argtype_3: CONSTRAINT =
  (/\ x)(/\ y) (/\ z) [ Connects(x, y, z) => Inport(z) ]
```

that must be satisfied by any architecture of this style.[12] The constraints say that the first argument of `Connects` must be a dataflow channel, the second must be an output port, and the third must be an input port.[13]

The body of `compiler_L1` consists of three sections, labeled `COMPONENTS`, `CONNECTORS`, and `CONFIGURATION`. These labels are included to increase readability and to encourage

[10] The syntax of architectural styles is identical to that of architectures except that the reserved word `STYLE` is used in place of `ARCHITECTURE`.

[11] SADL syntax encourages implicit definition of predicates, using `PREDICATE` and a collection of constraints. Explicit definition would say exactly which interpretations of the predicate are allowed as part of the declaration. A similar approach will be taken to the definition of types.

[12] We mentioned earlier that the formal semantics of SADL requires extended first-order logic. The SADL constraint language is classical first-order logic with quantifiers that range over the natural numbers. This expressive power is required to describe certain topological constraints on an architecture.

[13] Our logical notation should be self-explanatory, with the possible exception of the use of '/\' and '\/' for the universal and existential quantifiers. This is an ASCII approximation to a quantifier notation, '$\bigwedge$' and '$\bigvee$', popularized by Polish logicians such as Tarski, Mostowski, and Kuratowski, and by some of their American students, most notably Richard Montague. Its principal advantage over '$\forall$' and $\exists$' is that it *has* a natural ASCII approximation. If you prefer, you can use '`FORALL`' and '`EXISTS`' for the quantifiers in SADL.

organizing the description in the way we have found easiest to maintain and extend. But the labels have no semantic significance, and you are free to omit them.

The first declaration in the COMPONENTS section

```
lexical_analyzer: Function
  [char_iport: SEQ(character)
     -> token_oport: SEQ(token), bind_oport: SEQ(binding)]
```

introduces a Function called lexical_analyzer. The lexer has a single input port called char_iport. Observe that the same port appears in the interface to compiler_L1. A constraint on architectures is that each input port of an architecture is an input port of some component of the architecture because the only way an architecture can accept a value is for some component of that architecture to accept the value. Similarly for output ports. Component lexical_analyzer has two output ports, token_oport and bind_oport, which supply a sequence of tokens and a sequence of bindings, respectively. The declarations of the other three components, parser, analyzer_optimizer, and code_generator, are understood in much the same way.

The first declaration in the CONNECTORS section

```
token_channel: Dataflow_Channel<SEQ(token)>
```

introduces a Dataflow_Channel called token_channel which carries sequences of tokens. Note the difference in the syntax for component types and connector types: the former has a signature of the form

[*port-declaration** -> *port-declaration**]

where a *port-declaration* has the form

identifier : *type-expression*

and a connector type has a signature of the form

< *type-expression* >

This distinctive syntax makes it easy to distinguish component types from connector types, and to distinguish both from other types.

The CONFIGURATION section contains the connection constraints that characterize the way the connectors wire the components together. In general, it also may contain additional constraints on the architecture. The first declaration in the CONFIGURATION section

```
token_flow: CONNECTION
  = Connects(token_channel, token_oport, token_iport)
```

says that token_channel connects output port token_oport of the lexer to input port token_iport of the parser. This is the dataflow connection that models the mechansism by which the lexer passes a token sequence to the parser for further processing. The other three connection constraints complete the description of dataflow among the four components via the other three connectors.

5 Level 2 of Compiler Architecture

The SADL architecture at Level 2 of the hierarchy is contained in Fig. 3. The architecture is heterogeneous in that it contains concepts from shared-memory, pipe-and-filter, and batch-sequential architectural styles. In the description of this architecture, we concentrate on the concepts and notation that we did not see in the Level 1 architecture.

The `lexical_analyzer` component at the dataflow level has been replaced by a *subarchitecture* called `lexical_analyzer_module` that encapsulates a function called `lexical_analyzer` and a redundant data structure called `symbol_table`.[14] Both are exported so that the global configuration constraints on `compiler_L2` can mention them. It next imports the identifiers it needs from other styles. (Note that, to simplify dependency tracking, identifiers imported by `compiler_L2` are *not* automatically made available to `lexical_analyzer_module`.)

Like any other architecture, the subarchitecture contains declarations of components, connectors, configuration constraints. In this example, there two components, no connectors, and two configuration constraints. The component `lexical_analyzer` is nearly identical to the top-level function of the same name, except that the binding port has been eliminated and the type of the token port has been changed from the abstract mathematical type `SEQ(binding)` to the implementation-level type `Finite_Stream(binding)`. The component `symbol_table` is a variable, which is a component type. It has no ports, as the component signature `[ -> ]` indicates. However, `Variable` does take an argument, a type expression that gives the type of values that it can hold. Variable `symbol_table` holds values of type `SEQ(binding)`. The two configuration constraints allow the `lexical_analyzer` to read and write the variable `symbol_table`.[15]

New notation appears in the second configuration constraint of `compiler_L2`:

```
read_bind: CONSTRAINT
  = Reads(analyzer_optimizer, lexical_analyzer_module!symbol_table)
```

The second argument to `Reads` is a *qualified name*, consisting of a module identifier, exclamation mark (`!`), and an identifier declared in that module. This mechanism provides a way of using the identifier `symbol_table` declared in `lexical_analyzer_module` without having to include it in an import list.

[14]The motivation for bundling the lexer and the symbol table has to do with keeping the derivation of this level from the higher level as straightforward as possible. Derivations will be discussed in Sec. 7.

[15]You might ask the following question at this point: Why are reading and writing, which would certainly be represented by arrows in a graphical notation, not modeled using connectors? While modeling these operations using connectors is certainly possible, we chose to base our style on a model closer to that used in programming languages, where there is a distinction between passing a value through a procedure's specified interface and reading or writing a global variable from within the body of the procedure. An advantage of our approach is that function signatures in the low level architecture correspond much more directly to code.

```
compiler_L2: ARCHITECTURE [char_iport: SEQ(character) -> code_oport: code]
  IMPORTING character, code, token, binding, ast FROM compiler_types
  IMPORTING Function FROM Functional_style
  IMPORTING Pipe, Finite_Stream, Connects FROM Process_Pipeline_style
  IMPORTING Variable, Reads, Writes FROM Shared_Memory_style
  IMPORTING Start_After_Finish_Of FROM Batch_Sequential_style
BEGIN
COMPONENTS
  lexical_analyzer_module: ARCHITECTURE
          [char_iport: SEQ(character) -> token_oport: Finite_Stream(token)]
     EXPORTING lexical_analyzer, symbol_table
     IMPORTING character, token, binding FROM compiler_types
     IMPORTING Function FROM Functional_style
     IMPORTING Finite_Stream FROM Process_Pipeline_style
     IMPORTING Variable, Reads, Writes FROM Shared_Memory_style
  BEGIN
  COMPONENTS
     lexical_analyzer: Function
        [char_iport: SEQ(character) -> token_oport: Finite_Stream(token)]
     symbol_table: Variable(SEQ(binding))[ -> ]
  CONFIGURATION
     write_bind: CONSTRAINT = Writes(lexical_analyzer, symbol_table)
     read_bind: CONSTRAINT = Reads(lexical_analyzer, symbol_table)
  END lexical_analyzer_module
  parser: Function[token_iport: Finite_Stream(token) -> ]
  analyzer_optimizer:   Function[ -> ]
  code_generator:       Function[ -> code_oport: code]
  abstract_syntax_tree: Variable(ast)[ -> ]
CONNECTORS
  token_pipe: Pipe<Finite_Stream(token)>
CONFIGURATION
  token_flow: CONNECTION
    = Connects(token_pipe, token_oport, token_iport)
  read_bind: CONSTRAINT
    = Reads(analyzer_optimizer, lexical_analyzer_module!symbol_table)
  write_base_ast: CONSTRAINT = Writes(parser, abstract_syntax_tree)
  read_base_ast: CONSTRAINT
    = Reads(analyzer_optimizer, abstract_syntax_tree)
  write_full_ast: CONSTRAINT
    = Writes(analyzer_optimizer, abstract_syntax_tree)
  read_full_ast: CONSTRAINT
    = Reads(code_generator, abstract_syntax_tree)
  precedence_1: CONSTRAINT
    = Starts_After_Finish_Of(analyzer_optimizer, parser)
  precedence_2: CONSTRAINT
    = Starts_After_Finish_Of(code_generator, analyzer_optimizer)
END compiler_L2
```

Figure 3: Level 2 of Compiler Architecture in SADL

6 Mapping Level 1 into Level 2

A mapping relates an abstract architecture to a concrete architecture that is intended to implement it.[16] Mappings are specified by a collection of pairs, called ***associations***, of the form

identifier `-->` *right-hand-side*

The identifier on the left-hand side of the association can be either the name of an architectural element (a component, connector, port, or other element), the name of a constraint, or a predicate (e.g., a type predicate). Let us call these ***element associations***, ***constraint associations***, and ***predicate associations***, respectively.

For element and constraint associations, the right-hand side consists of a (possibly empty) list of names. The intended interpretation of such an association is that the list of objects named on the right implements the object named on the left. For example, an association might say that some combination of concrete-level components and connectors provides an implementation of an abstract-level connector. In the case of constraints, the intended interpretation is that satisfaction of the constraints on the right guarantees satisfaction of the constraint on the left.

For predicate associations, the right-hand side is defined in terms of a new concept called a ***virtual predicate***. The syntax for virtual predicates is somewhat complicated, but the basic idea is simple. Associating a predicate with a virtual predicate is intended to say that objects denoted by the predicate on the left-hand side are implemented by the objects denoted by the virtual predicate on the right-hand side.

The next subsection explains the mapping from `compiler_L1` to `compiler_L2`, which consists entirely of element and constraint associations. The subsequent subsection discusses the mapping between the architectural styles employed by these architectures, which requires predicate associations. The two mappings are combined into a single mapping for purposes of formal analysis [1].

Typically, in well-organized SADL specifications of specific architectures, mappings between the architectures consist entirely of element and constraint associations, whereas mappings between architectural styles consist almost entirely of predicate and constraint associations.

6.1 Mapping `compiler_L1` to `compiler_L2`

By convention, we make the default assumption that, if a concrete-level object has the same name as an abstract-level object, then it implements that abstract-level object. So

[16] A mapping can be used for several purposes. We have studied in detail its use in determining whether or not a given architecture is implemented correctly by another architecture — even in the presence of a change in representation [2]. The essential idea is that a SADL mapping provides enough information to uniquely determine a mapping from sentences in the language of the abstract architecture to sentences in the language of the concrete architecture. Our correctness criterion is that this mapping on sentences must be a faithful interpretation of the theory of the abstract architecture in the theory of the concrete architecture.

```
compiler_map: MAPPING FROM compiler_L1 TO compiler_L2
BEGIN
  lexical_analyzer --> (lexical_analyzer_module)
  bind_oport       --> ( )
  base_ast_oport   --> ( )
  base_ast_iport   --> ( )
  bind_iport       --> ( )
  full_ast_oport   --> ( )
  full_ast_iport   --> ( )
  token_channel    --> (token_pipe)
  bind_channel     --> (lexical_analyzer_module!symbol_table)
  base_ast_channel --> (abstract_syntax_tree)
  full_ast_channel --> (abstract_syntax_tree)
  bind_flow        --> (lexical_analyzer_module!write_bind, read_bind)
  base_ast_flow    --> (write_base_ast, read_base_ast,
                              precedence_1, precedence_2)
  full_ast_flow    --> (write_base_ast, read_base_ast,
                              precedence_1, precedence_2)
END compiler_map
```

Figure 4: SADL Mapping Between Level 1 and Level 2

associations with a name on the left and a list consisting of that same name on the right can be omitted. Omitting default associations, the intended SADL mapping between the two levels of the compiler is shown in Fig. 4. The first association in the mapping

```
lexical_analyzer --> (lexical_analyzer_module)
```

says that second-level subarchitecture `lexical_analyzer_module` implements the first-level function `lexical_analyzer`. The next association

```
bind_oport --> ( )
```

says that `bind_oport` has been eliminated from the architecture `compiler_L2`. The object that implements `bind_oport` has not been explicitly declared in the second-level architecture because the dataflow connection that passes bindings from the lexer to the optimizer is implemented by having the optimizer read a data structure within the lexer. At the code level, the bindings will not be passed to the procedure that performs optimization through its explicitly declared interface. Instead, a procedure call (or method invocation, or similar mechanism) in the body of the optimization procedure will retrieve the data. Thus, `bind_oport` is implemented by the location in the component `lexical_analyzer` of subarchitecture `lexical_analyzer_module` that writes the bindings to `symbol_table`, and `bind_iport` is implemented by the location in `analyzer_optimizer` that reads the bindings from `symbol_table`. Neither of these locations is explicitly named in the second-level description. In that sense, these ports are implemented by "nothing". That `bind_oport` is mapped to a location in `lexical_analyzer` that writes to `symbol_table` can be inferred from the mapping of the connection in `bind_flow` to the constraints on writing and reading the symbol table expressed in `write_bind` of `lexical_analyzer_module` and `read_bind`.

Another interesting aspect of this mapping is the use of a single variable `abstract_syntax_tree` to implement both channel `base_ast_channel` and channel `full_ast_channel`. To preserve the directionality of dataflow, the two precedence constraints must be satisfied. For example, the constraints do not allow a dataflow connection from the code generator to the parser. The two constraints are included in the list of constraints that implement `base_ast_flow` and `full_ast_flow`.

6.2 Style Mappings

6.2.1 Formalization

For purposes of illustration, we show how dataflow predicates are mapped to virtual predicates of the shared-memory language. From Fig. 4 we can see that some dataflow channels (in particular, `bind_channel`, `base_ast_channel`, and `full_ast_channel`) must be implemented as variables. This means that we must associate type predicate `Channel` with type predicate `Variable`. It would be tempting simply write something like

```
Channel --> Variable
```

but this mapping is too simplistic.

The first problem is that there may be no concrete-level predicate that denotes the set of objects that implement an abstract-level type. Consider, for example, the predicate `Function`. A function at the abstract level can be implemented by a concrete-level function or by a concrete-level subarchitecture ("bubble decomposition"). There is no single predicate at the concrete-level true of functions, architectures, and nothing else, and it is not at all natural to introduce one as part of the concrete-level specification.

A common notation in logic for predicate abstraction is

$$[\ x\ |\ \phi\ ],$$

which says that x is abstracted from ϕ. If we were to use this notation on the right-hand side of associations, we would get

```
Function --> [ y | Function(y) V ARCHITECTURE(y) ]
```

which is only slightly more complicated than the mapping above for type predicate `Channel`.

Unfortunately, a second problem remains. Suppose that a single abstract-level object is being implemented by several concrete-level objects. For example, a channel could be implemented as a pair of channels linked by a component that operates on the values being transmitted. An abstract-level assertion about the single object must then correspond a concrete-level assertion about the several objects. Suppose that we could specify an association of the abstract-level name a with the list $(a_1,\ a_2,\ \ldots,\ a_n)$ of concrete-level names, and the association of abstract-level predicate P with a virtual predicate based on a formula ϕ containing `n` free variables. This should be sufficient to determine an association of the atomic formula $P(a)$ to the appropriate substitution instance of ϕ.

The missing piece of information is which variable of ϕ corresponds to which of the a_i To determine the concrete-level equivalent of $P(a)$,

1. Find the association a `-->` `(`a_1`,` a_2`,` $\ldots$`,` a_n`)` that determines the implementation of a.

2. Find the association

   $$P \texttt{ --> [(} y \texttt{ --> (} y_1\texttt{, } y_2\texttt{, } \ldots\texttt{, } y_n\texttt{)) | } \phi \texttt{]}$$

   that determines the implementation of P. The right-hand side is what we call a ***virtual predicate***. This SADL notation is an extension of the notation for predicate abstraction.

3. Match the name association with the free variable association to determine that the result of simultaneously substituting $a_1, a_2, \ldots, a_n$ for $y_1, y_2, \ldots, y_n$ in ϕ implements $P(a)$.[17]

Multinary predicates are mapped to virtual predicates with a list of free variable associations of the proper length.

6.2.2 A SADL Style Mapping

The dataflow predicates used in the abstract-level specification of the compiler, some of which are built-in and some of which are imported from various SADL style specifications, are `Channel`, `Function`, `IN_PORT`, `OUT_PORT`, `CAN_CARRY`, `MIGHT_SUPPLY`, `CAN_ACCEPT`, and `CONNECTS`. We now show how these predicates are mapped to virtual predicates of the shared-memory language.[18]

The architecture mapping in Fig. 4 shows that we are implementing a dataflow channel by writing and reading a variable of the same type. In particular, it says that

- the Level 1 channel `bind_channel` is implemented by the Level 2 variable `symbol_table` of the `lexical_analyzer_module` subarchitecture,
- the Level 1 ports `bind_oport` and `bind_iport` that `bind_channel` connects are implemented by "nothing" (i.e., by objects not named in the Level 2 architecture), and
- the two Level 2 assertions that guarantee that the Level 1 assertion `bind_flow` is satisfied are `write_bind` of the `lexical_analyzer_module` subarchitecture and `read_bind`.

[17]Note that the variable y doesn't really play any role in the matching process. But the visual structural analogy to associations is suggestive.

[18]A complete specification of the mapping from Level 1 to Level 2 also would have to include the trivial mapping on the predicates of compiler types predicates that associates each predicate with the corresponding virtual predicate.

These mappings constrain the legal predicate mappings.

In our extended virtual predicate notation, the two compiler associations above are written in SADL as

```
Channel --> [ (x --> (y)) | Variable(y) ]
```

and

```
Function --> [ (x --> (y)) | Function(y) V ARCHITECTURE(y) ]
```

To see how this predicate association is used, consider the implementation of the dataflow type declaration

```
Function(lexical_analyzer)
```

The mapping contains the element association

```
lexical_analyzer --> lexical_analyzer_module
```

Therefore, we substitute `lexical_analyzer_module` for y in the formula

```
Function(y) V ARCHITECTURE(y)
```

to get

```
Function(lexical_analyzer_module) V ARCHITECTURE(lexical_analyzer_module)
```

which says that the lexical analyzer function can be implemented as a function or a subarchitecture.

Next, assume that, in the style mapping, ports are uniformly implemented as call sites (i.e., locations where calls of access functions are performed which are unnamed in the Level 2 architecture)

```
IN_PORT  --> [ (x_1 --> (y_1), x_2 --> (y_2))
                | Call(y_1, y_2) & (\/ y) Can_Get(y_1, y) ]
OUT_PORT --> [ (x_1 --> (y_1), x_2 --> (y_2))
                | Call(y_1, y_2) & (\/ y) Might_Put(y_1, y) ]
```

These associations mean that the formula

```
OUT_PORT(bind_oport, lexical_analyzer)
```

which says that `bind_oport` is an output port of `lexical_analyzer`, is mapped to the formula

```
Call(*a*, lexical_analyzer_module) & (\/ y) Might_Put(*a*, y)
```

which says that `*a*` (an arbitrary name introduced for the object that implements `bind_oport`) is a call within `lexical_analyzer_module` and that it potentially stores a value in some variable. The associations for the other predicates are similar.

```
CAN_CARRY    --> [ (x_1 --> (y_1), x_2 --> (y_2)) | Can Hold(y_1, y_2)
MIGHT_SUPPLY --> [ (x_1 --> (y_1), x_2 --> (y_2)) | Might_Put(y_1, y_2) ]
CAN_ACCEPT   --> [ (x_1 --> (y_1), x_2 --> (y_2)) | Can_Get(y_1, y_2) ]
CONNECTS     --> [ (x_1 --> (y_1), x_2 --> (y_2), x_3 --> (y_3))
                              | Writes(y_2, y_1) & Reads(y_3, y_1) ]
```

7 Refinement Patterns

SADL is the product of a continuing research effort to investigate architectural refinement. One of the objectives of this effort is to formalize expert knowledge of implementation techniques in the form of reusable architectural design refinement patterns. If a collection of architecture refinement patterns have been ***verified*** — i.e., if we have proven that, whenever any pattern in the collection is applied to an architectural design A, it produces a more detailed design A' such that any system whose architecture correctly implements A' also correctly implements A — then we can use repeated applications of those patterns to generate concrete implementations of abstract architectures that are guaranteed to be correct. This correctness guarantee is exactly what is needed to be sure that an architectural hierarchy is telling a consistent story about a particular system or a family of systems.

An example of a verified, reusable refinement pattern appears in Fig. 5. A refinement pattern consists of five parts: a pattern that the abstract architectural description must match, a pattern that the concrete architectural description must match, associations that determine a mapping from the abstract architecture to the concrete one, optional additional constraints on the abstract architecture, and optional additional constraints on the concrete architecture.[19]

Patterns are easy to read once pattern variables are understood. Any identifier that is preceded by the symbol `@` is a ***pattern variable***. A pattern variable beginning with a single `@` is bound to a single chunk of text — more precisely, a single subtree of a SADL abstract syntax tree — while a pattern variable beginning with a pair of `@`'s is bound to a collection of text chunks. Consider the abstract pattern part of the refinement pattern in Fig. 5:

```
@m: ARCHITECTURE [ @@ips -> @@ops ]
BEGIN
  @f1: Functional_style!Function[ @@ips1 -> @op: @t, @@ops1 ]
  @f2: Functional_style!Function[ @@ips2, @ip: @t -> @@ops2 ]
  @c:  Dataflow_style!Channel<@t>
  @a:  CONNECTION = Dataflow_style!Connects(@c, @op, @ip)
  @@rest
END @m
```

Fig. 6 gives a binding for pattern variables that matches the Level 1 compiler architecture seen earlier in Fig. 2.[20] It is easy to see how the concrete architecture part of the pattern can be bound to a more concrete compiler architecture, and how the associations part of the pattern can be bound to a mapping between the two levels.

[19]Patterns can be composed in an attempt to verify a mapping between two levels in an pre-existing architectural hierarchy, or can be used as rules to generate a hierarchy. In the former case, the optional constraints on the two architectures are simply checked; in the latter case, the constraints on the abstract architecture must be satisfied before the rule can be applied, and the constraints on the concrete architecture that are not demonstrably satisfied after the rule is applied must be added to the lower-level description.

[20]If we wanted to use the refinement pattern in Fig. 5 to begin to transform the Level 1 compiler architecture into the Level 2 compiler architecture by introducing the variable that holds the abstract syntax tree which implements communication between the parser and the optimizer, then this is the binding we would use in applying the rule.

```
channel_to_var: PATTERN =
ABSTRACT_PATTERN::
  @m: ARCHITECTURE [ @@ips -> @@ops ]
  BEGIN
    @f1: Functional_style!Function[ @@ips1 -> @op: @t, @@ops1 ]
    @f2: Functional_style!Function[ @@ips2, @ip: @t -> @@ops2 ]
    @c:  Dataflow_style!Channel<@t>
    @a:  CONNECTION = Dataflow_style!Connects(@c, @op, @ip)
    @@rest
  END @m
CONCRETE_PATTERN::
  @m: ARCHITECTURE [ @@ips -> @@ops ]
  BEGIN
    @f1: Functional_style!Function[ @@ips1 -> @@ops1 ]
    @f2: Functional_style!Function[ @@ips2 -> @@ops2 ]
    @v:  Shared_Memory_style!Variable(@t)[ -> ]
    @a1: CONSTRAINT = Shared_Memory_style!Writes(@f1, @v)
    @a2: CONSTRAINT = Shared_Memory_style!Reads(@f2, @v)
    @@rest
  END @m
ASSOCIATIONS::
  @op --> ()
  @ip --> ()
  @c  --> (@v)
  @a  --> (@a1, @a2)
END channel_to_var
```

Figure 5: SADL Pattern for Refining Level 1 of Compiler Architecture

Variable	*Value*
@m	compiler_L1
@@ips	{char_iport: SEQ(character)}
@@ops	{code_oport: code}
@f1	parser
@@ips1	{token_iport: SEQ(token)}
@op	base_ast_oport
@t	ast
@@ops1	{ }
@f2	analyzer_optimizer
@@ips2	{bind_iport: SEQ(binding)}
@ip	base_ast_iport
@@ops2	{full_ast_oport: ast}
@c	base_ast_channel
@a	base_ast_flow
@@rest	{lexical_analyzer:..., code_generator:..., taken_channel:..., bind_channel:..., full_ast_channel:..., token_flow:..., bind_flow:..., full_ast_flow:...}

Figure 6: Binding of Pattern Variables

8 Parameterized Architectures

The only variability in the compiler architecture is in the specific sequences of characters that are supplied as its input and the specific sequences that are produced as its output. Its components, connectors, connections, and constraints are fixed in number and kind.

In general, it is desirable to have more flexibility in defining architectures. For example, we may not want to fix the number of components and connections in an architecture. This occurs in the X/Open Distributed Transaction Processing (DTP) reference architecture [4]. To formalize X/Open DTP, we essentially need to build a client-server architecture in which the server is connected to a fixed *but undetermined* number of clients.

The ability to leave certain design decisions open is supported in SADL through three mechanisms.

- **Parametric architectures**. Architectures can be parameterized by totally or partially determined values. For example, `n:NAT` declares `n` to be a natural number, but does not say which natural number it is.
- **Parametric signatures**. These are signatures that do not fix the number of elements in the signature.
- **Parametric constraints**. These are constraints that do not fix the number of atomic formulae in an expression.

As an illustration of the three forms of parameterization, we show how to specify a client-server architecture with an undeterminate number of clients.[21] We begin by introducing a numeric parameter

```
n: NAT     % n is the number of clients
```

to represent the number of clients. Any constraints on this number can be included as well, for example,

```
client_limit: CONSTRAINT = [ n < 1024 ]
```

Suppose, for simplicity, that every client has the same interface: it passes a natural number to the server, which returns a natural number to the client. Then the type of all clients can be represented by

```
client_type: TYPE <= Function[in: NAT -> out: NAT]
```

That is, every client is a functional component with an input port of type `NAT` and an output port of type `NAT` (though perhaps not every functional component of that type is a client). Obviously, we cannot introduce an individual name for every client, but we can introduce a systematic way of referring to them.

```
client: {i: NAT | i < n} --> client_type
```

[21]The dynamic X/Open DTP architecture can be described in this way by pretending that all clients that will ever interact with the server "exist", and having new clients "register" prior to engaging in transactions.

declares `client` to be a function (in the mathematical sense) from the set of natural numbers less than `n`—a convenient set of `n` objects—into `client_type`. If we specify that `client` is one-to-one and onto, then it provides an enumeration of the clients, and we can use `client(i)` to refer to the `i`-th client.

The interface to the server is specified by means of a SADL parametric signature. We know that the server has `n` input ports and `n` output ports, but the value of `n` is not determined. We therefore want to allow signatures to contain not just port declarations, but indexed collections of port declarations. If we write

```
<< p(i): t |(i: NAT) ...i...  >>
```

for a collection of ports `p(i)` of type `t` indexed by the natural numbers `i` that satisfy the condition `...i...` on `i`, then the server can be declared as follows:

```
server: Function
   [ << in(i) : NAT |(i: NAT) i < n >>
          -> << out(i) : NAT |(i: NAT) i < n >> ]
```

The connections between the clients and the server cannot be declared individually. We could introduce a function for referring to them, or simply assert that connections exist by means of SADL parametric constraints. The connection constraints

```
dataflow_to_server: CONNECTION =
   (/\ i: NAT | i < n) (\/ c: Channel<NAT>)
       Connects(c, client(i).out, server.in(i))
dataflow_from_server: CONNECTION =
   (/\ i: NAT | i < n) (\/ c: Channel<NAT>)
       Connects(c, server.out(i), client(i).in)
```

say that there is a channel connecting the output port of the `i`-th client to the `i`-th input port of the server, and there is a channel connecting the `i`-th output port of the server to the input port of the `i`-th client.

9 Optional Elements in an Architecture

It is sometimes useful to say that a particular architectural object is present *only under certain conditions*. Suppose, for example, that we want to define an optional dataflow connection between two components. We can begin by introducing a Boolean parameter

```
connected?: BOOL
```

which will be true when there is a connection and false when there is none. Hence the port collection

```
<< p(i): t |(i: NAT) i < 1 & connected? >>
```

contains one port if `connected?` is true and no ports if it is false. Thus an optional connection between functional components `a` and `b` can be specified as follows:

```
connected?: BOOL
a: Function[ -> << out(i): t |(i: NAT) i < 1 & connected? >> ]
b: Function[ << in(i): t |(i: NAT) i < 1 & connected? >> -> ]
df: CONNECTION =
   (/\ i: INT | i < 1 & connected?)(\/ c: Channel<t>)
       Connects(c, a.out(i), b.in(i))
```

10 Conclusion

The compiler example illustrates how SADL can be used to describe simple "textbook" architectures. Each level of detail is described by a SADL architecture specification, employing constructs from the relevant architectural styles. Multiple levels of description are linked by mappings which associate objects declared at one level with corresponding objects at the next level, creating a refinement hierarchy. A systematic way of creating consistent refinement hierarchies is to develop them transformationally by applying verified refinement patterns. Our experience suggests that simple, but real, architectures can be generated using a small number of patterns [2]. If you understand the compiler example, you understand SADL well enough to use it for describing simple architectures.

SADL has been applied to larger examples, requiring parameterized specifications and other advanced features. For example, SADL has been used to formalize the X/Open Distributed Transaction Processing (DTP) reference architecture [4] at multiple levels of detail. The highest level of the SADL DTP hierarchy corresponds to the informal dataflow diagram in the X/Open documents, a lower level corresponds to the C procedure call model documented in the hundreds of pages of X/Open documents, and still lower levels detail various implementation strategies. The complete formalization consists of a seventeen-level hierarchy with architectures ranging in length from 14 to 338 non-blank, non-comment lines. The architectures are linked by mappings to form a tree of alternative implementations of the X/Open specification. This example illustrates the feasibility of using SADL for real-world architectural specification.

Recently, we have demonstrated how our research on architecture hierarchy can serve as a basis for secure system design [3]. Our approach was demonstrated by incorporating security directly into the X/Open DTP architecture. It was necessary to build an architecture hierarchy containing four secure SADL architectures related by formal proofs of the kind described in our earlier papers.

References

[1] Mark Moriconi and Xiaolei Qian, "Correctness and Composition of Software Architectures", *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, New Orleans, Louisiana (December 1994), pp. 164–174.

[2] Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider, "Correct architecture refinement", *IEEE Transactions on Software Engineering*, vol. 21, no. 4 (April 1995), pp. 356-372.

[3] Mark Moriconi, Xiaolei Qian, R. A. Riemenschneider, and Li Gong, "Secure Software Architectures", *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California (May 1997).

[4] X/Open Company, Ltd., *Distributed Transaction Processing: Reference Model*, Version 2 (November 1993).

A The SADL 1.0 Grammar

The following is a formal reference grammar for SADL 1.0. It simplifies the grammar supported by the SADL 1.0 parser in three ways. First, the epicycles introduced to cope with a limit of one token lookahead were removed. Second, iteration constructs were expanded into simple recursions. Third, prettyprinting annotations were stripped.

A.1 Reserved words

The following identifiers should not be redeclared in SADL specifications, as their meanings are predefined.

`ABSTRACT_CONSTRAINTS`	`ABSTRACT_PATTERN`	`ALL`
`AND`	`ARCHITECTURE`	`ASSOCIATIONS`
`BEGIN`	`BOOL`	`BUT`
`COMPONENTS`	`CONCRETE_CONSTRAINTS`	`CONCRETE_PATTERN`
`CONFIGURATION`	`CONNECTORS`	`CONSTANT`
`CONSTRAINT`	`CONTAINED_IN`	`END`
`EXISTS`	`EXPORTING`	`FORALL`
`FROM`	`HAS_TYPE`	`IFF`
`IMPORTING`	`INT`	`MAPPING`
`NAT`	`NOT`	`ONLYIF`
`OR`	`PATTERN`	`PREDICATE`
`PROPERLY_CONTAINED_IN`	`SEQ`	`SET`
`STYLE`	`TO`	`TYPE`
`V`		

A.2 Comments

Comments are arbitrary character sequences. When a `%` character is encountered, the parser ignores the `%` and all characters following it on that line.

A.3 Notation for Productions

The notation used should be familiar. The production

$$A \longrightarrow \alpha \mid \beta \mid \gamma$$

means that non-terminal A can be rewritten to any one of α, β, and γ. Square brackets indicate options, so that

$$A \longrightarrow [B]C$$

is simply a slightly more compact way of writing

$$A \longrightarrow BC \mid C$$

Non-terminals are written in an *italic* font, while terminals are quoted and written in a `teletype` font.

A.4 Specification Unit

A SADL specification is a named, possibly parameterized, architecture, architectural style, mapping, or pattern.

$$
\begin{array}{rcl}
\textit{sadl-spec} & \longrightarrow & \textit{id-or-mid}\ [\text{`}\texttt{[}\text{'}\ \textit{formals}\ \text{`}\texttt{]}\text{'}]\ \text{`}\texttt{:}\text{'}\ \textit{spec-rest} \\
 & & \\
\textit{spec-rest} & \longrightarrow & \text{`}\texttt{ARCHITECTURE}\text{'}\ [\textit{comp-sig}]\ \textit{arch-rest} \\
 & \mid & \text{`}\texttt{STYLE}\text{'}\ \textit{styl-rest} \\
 & \mid & \text{`}\texttt{MAPPING}\text{'}\ \text{`}\texttt{FROM}\text{'}\ \textit{id}\ \text{`}\texttt{TO}\text{'}\ \textit{id}\ [\textit{map-rest}] \\
 & \mid & \text{`}\texttt{PATTERN}\text{'}\ \text{`}\texttt{=}\text{'}\ \textit{pat-rest} \\
 & & \\
\textit{formals} & \longrightarrow & \textit{formal}\ \text{`}\texttt{,}\text{'}\ \textit{formals} \\
 & \mid & \textit{formal} \\
 & & \\
\textit{formal} & \longrightarrow & \textit{id-or-mids}\ \text{`}\texttt{:}\text{'}\ \textit{formal-decl} \\
 & & \\
\textit{formal-decl} & \longrightarrow & \text{`}\texttt{ARCHITECTURE}\text{'} \\
 & \mid & \text{`}\texttt{TYPE}\text{'} \\
 & \mid & \textit{type-expr}
\end{array}
$$

A.5 Architectures and Styles

An architectures consists of a list of imports and exports followed by declarations of types, variables, constants, assertions, and architectures. Each of these elements of an architecture is optional. An architectural style is an architecture, but normally will not contain some of the elements, such as connections, in an architecture.

arch-rest ⟶ [*portings*] ‘`BEGIN`’ [*decls*] ‘`END`’ *id-or-mid*

styl-rest ⟶ [*portings*] ‘`BEGIN`’ [*decls*] ‘`END`’ *id-or-mid*

portings ⟶ *porting portings*
| *porting*

porting ⟶ ‘`EXPORTING`’ *ting*
| ‘`IMPORTING`’ *ting* ‘`FROM`’ *id-or-mids*

ting ⟶ ‘`ALL`’ [*ting-qualifier*]
| *id-or-mids*

ting-qualifier ⟶ ‘`BUT`’ *id-or-mids*

decls ⟶ *decl decls*
| *decl*

decl ⟶ *id-or-mids* ‘`:`’ *decl-body*
| ‘`{`’ *id-or-mid* ‘`(`’ *id-or-mids* ‘`)`’ ‘`:`’ *decl-body*
    ‘`|`’ [‘`(`’ *b-vars* ‘`)`’] *fmla* ‘`}`’
| ‘`@@`’ *id*
| ‘`COMPONENTS`’
| ‘`CONNECTORS`’
| ‘`CONFIGURATION`’

decl-body ⟶ *type-decl*
| *var-decl*
| *const-decl*
| *assert-decl*
| *arch-decl*

type-decl ⟶ ‘`TYPE`’ [*type-def*]

type-def ⟶ ‘`=`’ *type-expr*
| ‘`<=`’ *type-expr*

var-decl ⟶ *type-expr* [‘`=`’ *expr*]

const-decl ⟶ ‘`CONSTANT`’ *type-expr* ‘`=`’ *expr*

assert-decl ⟶ ‘`CONSTRAINT`’ ‘`=`’ *fmla*
| ‘`CONNECTION`’ ‘`=`’ *fmla*

arch-decl ⟶ ‘`ARCHITECTURE`’ ‘`=`’ [*signature*] *arch-rest*

A.6 Mappings

A mapping may relate two architectures or two architectural styles.

map-rest	$\longrightarrow$	'`BEGIN`' *associations* '`END`' *id*
assocs	$\longrightarrow$	'`ASSOCIATIONS`' '`::`' *associations*
associations	$\longrightarrow$	*association associations*
	$\mid$	*association*
association	$\longrightarrow$	*id-or-mid* '`-->`' *rhs*
rhs	$\longrightarrow$	'`(`' [*id-or-mids*] '`)`'
	$\mid$	'`[`' '`(`' [*var-assocs*] '`)`' '`|`' *fmla* '`]`'
var-assocs	$\longrightarrow$	*var-assoc* '`,`' *var-assocs*
	$\mid$	*var-assoc*
var-assoc	$\longrightarrow$	*id-or-mid* '`-->`' '`(`' [*id-or-mids*] '`)`'

A.7 Patterns

pat-rest	$\longrightarrow$	*abs-pat conc-pat* [*assocs*] [*abs-constrs*] [*conc-constrs*] '`END`' *id*
abs-pat	$\longrightarrow$	'`ABSTRACT_PATTERN`' '`::`' *id-or-mid* '`:`' '`ARCHITECTURE`' [*comp-sig*] *arch-rest*
conc-pat	$\longrightarrow$	'`CONCRETE_PATTERN`' '`::`' *id-or-mid* '`:`' '`ARCHITECTURE`' [*comp-sig*] *arch-rest*
abs-constrs	$\longrightarrow$	'`ABSTRACT_CONSTRAINTS`' '`::`' *fmlas*
conc-constrs	$\longrightarrow$	'`CONCRETE_CONSTRAINTS`' '`::`' *fmlas*

A.8 Interfaces

signature	$\longrightarrow$	*comp-sig*
	$\mid$	*con-sig*

comp-sig	⟶	‘`[`’ [*port-decls*] ‘`->`’ [*port-decls*] ‘`]`’
port-decls	⟶	*port-decl* ‘`,`’ *port-decls*
	|	*port-decl*
port-decl	⟶	*id-or-mid* ‘`:`’ *type-expr*
	|	‘`<<`’ *id-or-mid indices* ‘`:`’ *type-expr*
		‘`|`’ [‘`(`’ *b-vars* ‘`)`’] *fmla* ‘`>>`’
	|	‘`@@`’ *id*
con-sig	⟶	‘`<`’ *type-expr* ‘`>`’

A.9 Constraints

SADL constraints are first-order logic formulas with optional typing.

fmlas	⟶	*fmla fmlas*
	|	*fmla*
fmla	⟶	*atomic-fmla*
	|	‘`-`’ *fmla*
	|	‘`NOT`’ *fmla*
	|	*fmla* ‘`&`’ *fmla*
	|	*fmla* ‘`AND`’ *fmla*
	|	*fmla* ‘`V`’ *fmla*
	|	*fmla* ‘`OR`’ *fmla*
	|	*fmla* ‘`=>`’ *fmla*
	|	*fmla* ‘`ONLYIF`’ *fmla*
	|	*fmla* ‘`<=>`’ *fmla*
	|	*fmla* ‘`IFF`’ *fmla*
	|	*quantifiers fmla*
atomic-fmla	⟶	*fnam-or-mid* [‘`(`’ *terms* ‘`)`’]
	|	*fnam-or-mid bin-rel term*
bin-rel	⟶	‘`=`’
	|	‘`/=`’
	|	‘`<`’
	|	‘`>`’
	|	‘`CONTAINED_IN`’
	|	‘`PROPERLY_CONTAINED_IN`’
	|	‘`HAS_TYPE`’
terms	⟶	*term* ‘`,`’ *terms*

| *term*

quantifiers ⟶ *quantifier quantifiers*
| *quantifier*

quantifier ⟶ ‘`(`’ *q-type id-or-mid* [‘`:`’ *type-expr*] [‘`|`’ *fmla*] ‘`)`’

q-type ⟶ ‘`\/`’
| ‘`EXISTS`’
| ‘`/\`’
| ‘`FORALL`’

expr ⟶ *term*

term ⟶ *fnam-or-num-or-mid* [‘`(`’ *id-or-mids* ‘`)`’]

A.10 Type Expressions

Any named object is a member of a type. This includes components, connectors, constraints, architectures, and mappings. SADL includes constructors for forming subtypes (‘{...|...}’) function types (‘`-->`’), product types (‘`X`’), and power types (‘`^`’).

type-expr ⟶ *fnam-or-mid* [*simple-type-args*]
| ‘`ARCHITECTURE`’ *comp-sig* [*arch-rest*]
| ‘`{`’ *full-name* ‘`:`’ *meta-type-expr*
‘`|`’ [‘`(`’ *b-vars* ‘`)`’] *fmla* ‘`}`’
| *type-expr* ‘`-->`’ *type-expr*
| *type-expr* ‘`X`’ *type-expr*
| *type-expr* ‘`^`’ *number*

simple-type-args ⟶ *signature*
| ‘`(`’ *texpr-or-nums* ‘`)`’ [*signature*]

meta-type-expr ⟶ *type-expr*
| ‘`TYPE`’

b-vars ⟶ *b-var* ‘`,`’ *b-vars*
| *b-var*

b-var ⟶ *id-or-mid* ‘`:`’ *type-expr*

A.11 Odds and Ends

$$
\begin{array}{rcl}
\textit{texpr-or-nums} & \longrightarrow & \textit{texpr-or-num}\ \text{`}\texttt{,}\text{'}\ \textit{texpr-or-nums} \\
 & | & \textit{texpr-or-num} \\[1ex]
\textit{texpr-or-num} & \longrightarrow & \textit{type-expr} \\
 & | & \textit{number} \\[1ex]
\textit{fnam-or-num} & \longrightarrow & \textit{full-name} \\
 & | & \textit{number} \\[1ex]
\textit{full-name} & \longrightarrow & \textit{name} \\
 & | & \textit{id}\ \text{`}\texttt{!}\text{'}\ \textit{name} \\[1ex]
\textit{name} & \longrightarrow & \textit{id} \\
 & | & \textit{id}\ \text{`}\texttt{.}\text{'}\ \textit{id} \\[1ex]
\textit{id-or-mids} & \longrightarrow & \textit{id-or-mid}\ \text{`}\texttt{,}\text{'}\ \textit{id-or-mids} \\
 & | & \textit{id-or-mid} \\[1ex]
\textit{id-or-mid} & \longrightarrow & \textit{id} \\
 & | & \text{`}\texttt{@}\text{'}\ \textit{id} \\[1ex]
\textit{name-or-mid} & \longrightarrow & \textit{name} \\
 & | & \text{`}\texttt{@}\text{'}\ \textit{id} \\[1ex]
\textit{fnam-or-mid} & \longrightarrow & \textit{full-name} \\
 & | & \text{`}\texttt{@}\text{'}\ \textit{id} \\[1ex]
\textit{fnam-or-num-or-mid} & \longrightarrow & \textit{fnam-or-num} \\
 & | & \text{`}\texttt{@}\text{'}\ \textit{id} \\[1ex]
\textit{id} & \longrightarrow & \text{any string of letters, digits, and underscores that} \\
 & & \text{begins with a letter} \\[1ex]
\textit{number} & \longrightarrow & \text{any string of digits}
\end{array}
$$