# A Logic Programming-based Environment for Integrating Architecture Description Languages*

**Qian Wang and Gopal Gupta**

Department of Computer Science
University of Texas at Dallas

**Abstract.** Numerous architectural description languages(ADLs) have been developed in the last decade. However, none of the ADLs and their tool-set are expressive enough to cover all the requirements that may be specified while developing a software system. An ADL based approach to software engineering will be more useful and powerful if ADLs can share architectural descriptions and if their analysis tools can be integrated. In this paper, we propose a semantics-based approach to integrating ADLs. A general, abstract executable form is developed for representing architectural information expressed in ADLs. A uniform query language, called ADLQL, is also defined that can be used to retrieve architectural information from this abstract form. The semantics of both the ADL and the query language is given in terms of this abstract executable form. The executable semantics of a query can then be run on the executable semantics of an architectural description to check for desired properties that the architectural description should satisfy. The semantics is expressed using Horn clause logic. There are at least three benefits of our framework. First, software designers and analysis tools can use a uniform query language to retrieve architectural information from descriptions written in different ADLs. Second, implementations and tool-sets for ADLs can be developed extremely quickly. Thus, as an ADL rapidly evolves, its implementation infrastructure can be developed at the same pace. Third, an architecture description written in one ADL can be readily translated into (subset of) another ADL.

## 1 Introduction

For successfully developing any complex software system, one of the critical tasks is to clearly describe the *architecture* of the software system. At the architectural level of design [7, 8], a software system is typically described as a composition of high-level, interacting components. The choice of an appropriate architecture can lead to a software product that satisfies its requirements and that can be easily modified as new requirements arise, while an inappropriate architecture can lead to cost over-runs and possibly project failure. Errors and changes of the architecture are the most expensive mistakes if they are discovered in later phases

* Authors are partially supported by grants from the US National Science Foundation, the Department of Education, and the Environmental Protection Agency.

of software development. As architectural design plays a very important role in the software engineering process, it becomes increasingly important to support architectural description and analysis with appropriate tools and environments.

Over the last ten years considerable research has been done in the field of software architecture. As a result, numerous architectural description languages (ADLs) have been developed; these include Aesop, Wright, ACME, C2, UniCon, Rapide, MetaH, Darwin (a survey of current ADLs can be found in [5]). However, research on ADLs is still primarily an academic endeavor. One of the main reasons is that each ADL just provides a specific set of capabilities for architectural design and analysis. None of the ADLs (and their affiliated tool-set) are expressive enough to cover *all* the requirements that may be specified while developing a software system. For example, software designers can use Aesop's toolkit to describe architecture style; Aesop has a graphical editor [4, 5] that can be specialized with visualizations appropriate to different styles. Wright [3, 5] models architecture using components, connectors, ports and roles. Using a variant of CSP it formalizes the behavior of each component and connector and provides capabilities for statically checking the consistency and completeness of the design. Rapide [6, 5] describes an architecture as a composition of components. Component behavior is specified in terms of the way outgoing events are produced in response to incoming events. Rapide allows architectural designs to be simulated, and provides tools for analyzing the result of those simulations.

Ideally, a software architect would like to have as many capabilities as possible in the ADL. Similarly, a software architect would like as much tool support as possible for that ADL. Thus, the designer may want to graphically specify the software architecture, analyze it statically for design errors, and then run a few simulations to understand the software's runtime behavior [1]. However, given the current situation, designers will have to use multiple ADLs and their associated tool-sets to accomplish this. Thus, they will have to use Aesop's graphic interface to draw the software architecture, use Wright's modeling and analysis tool-set [1, 2] to analyze the architecture, and use Rapide's simulation facility to study sample execution traces. Clearly, the ADL-based technology will be more useful and powerful if architectural descriptions as well as the tools that have been developed for various ADLs can be shared. Unfortunately, there are lots of differences that exist among various ADLs:

1. There is no universally-accepted definition of software architecture, nor is there any universal standard [2, 5]. Each ADL has a somewhat different conception of architectural structures. Different ADL use different notations to describe software structures.
2. There are some significant differences in the way ADLs specify architectural properties, For example, Wright uses CSP to formalize the behavior of connector, while Rapide uses *POSETs* to describe a similar behavior. We must deal with these gaps between ADLs in order to integrate them.

Thus, to facilitate the task of specifying the architectures of a software system, we need a framework that will integrate various ADLs. The integration framework should be such that different parts of a design can be specified using different ADLs, yet the architect should be able to reason about the whole design. Additionally, the integration framework should also allow one ADL specification to be translated into another so that ADL specific tool-sets can be cross-used. Thus, an ideal integration framework should

1. allow us to reason over the whole architectural specification, even though it may have been written using multiple ADLs.
2. permit cross-use of ADL specific tool-sets.
3. allow ADLs to be quickly integrated as new ones are designed or as old ones evolve and are changed.

In this paper, we propose a denotational semantics [19, 20] based framework to integrate architecture description languages. The process of integration is divided into two phases. In the first phase, each ADL specification is semantically mapped to a *set of relations* which captures the architectural information inherent in the ADL specification. This set of relations is represented as a database of predicates. In the second phase, we define a uniform query language (called ADLQL) to retrieve architectural information, that is either explicitly or implicitly coded in the ADL specification, from this database of predicates. The retrieved information can be (optionally) output using the syntax of any ADL. The framework can be used in a number of ways:

- Software architects can map their architectural specification to this database of predicates mentioned above, and then query it using ADLQL to understand the specification better, especially those aspects that are implicit in the design. Properties of the ADL specification that are of interest can also be verified in this manner.
- Different components of a large system can be specified using different ADLs by different architects. These multiple components can then be integrated using our framework, and various system properties queried and verified.
- Software architects can specify the target language of the ADLQL query to be another target ADL, in effect, translating the specification written in the original ADL to the target ADL. Tools that have been developed for the target ADL can then be used to further analyze the original specification. The translation capability can also be used to cross-compare two or more architecture descriptions written in different ADLs.

The cornerstone of our integration framework is denotational semantics and Horn clause logic (Horn logic for brevity). Essentially, the denotational semantics of an ADL, in terms of a database of predicates, is specified in Horn logic. If this semantics is executable, the *denotation* of a specification written in this ADL is also "executable", and can be used to automatically infer the architectural information implied in that specification. Essentially, this semantic specification can be executed to map a software architecture to its corresponding database of predicates. This database of predicates itself is represented in Horn logic (pure Prolog) [21].

Recall that denotational semantics of a language consists of three components: *syntax specification*, *semantic algebras*, and *valuation functions* [19, 20]. All three components can be specified in Horn logic [14]. The same approach is used to make ADLQL queries executable, i.e., denotational semantics of ADLQL is also specified in Horn logic. The denotation of an ADLQL query program can be obtained and is executable (this denotation is a query in Horn logic).

The main benefit of our Horn logic based approach is that the semantics of a given ADL as well as the semantics of ADLQL, both of which are executable, can be specified very quickly. For example, it took us only one man month to develop a semantic interpreter for ACME [2,1]. Likewise, the development of ADLQL and its semantic interpreter took an additional man month (note, however, the design of ADLQL is still evolving, however, these changes can be incorporated in its semantics very rapidly). Thus, new ADLs can be rapidly integrated into our framework. Additionally, as an ADL rapidly evolves, its implementation infrastructure can be developed rapidly at the same pace.

The novel contribution of this paper is the use of a semantics based framework for integrating ADLs as well as a query language for querying architectural descriptions. Our framework also allows software architects to understand their design better and verify properties of their design by posing appropriate queries. The framework provides considerable flexibility as the query language is decoupled from any specific ADL, allowing components of a system to be specified in different ADLs. Thus, our whole integration framework is language independent.

## 2 Semantics-based Approach to Integration

### 2.1 A Framework for Integrating ADLs

There are a number of approaches that can be adopted for integrating multiple ADLs and coping with the problem of language heterogeneity. These include: (i) select one of the languages, and translate specifications written in all other languages to this language; (ii) design a language that is a union of all the languages in consideration; (iii) design a language that is an intersection of all the languages in consideration. However, the above approaches cannot effectively deal with integration of ADLs for obvious reasons (detailed discussion on this issue can be found elsewhere [1, 2]).

In our opinion, a better approach to integration is one in which each ADL is *semantically* mapped to a common *executable* format that is powerful enough to capture the semantics of all the ADLs. Translators can then be built from each ADL to this common format. In fact, these translators can be obtained with relative ease from the formal semantic description of the ADLs [14]. Indeed this is the approach we adopt. We select Horn clause logic (or pure Prolog) as the common executable format that captures the architectural information coded in the specification. Since Horn clause logic can be regarded as a Turing complete programming language [21], any ADL can be translated into it. The common executable format is based on a set of relations (represented as a database of predicates in the Horn logic notation). Thus, for each ADL that is to be integrated, we denotationally specify its semantics in terms of a database of predicates. These denotational definitions in turn are also coded in Horn logic. These denotational definitions are executable themselves and thus serve as translators that automatically translate an ADL specification into a database of predicates.

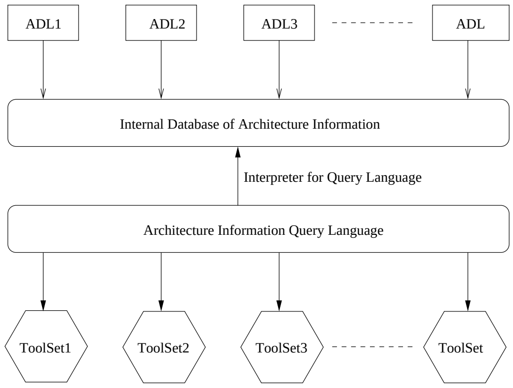


**Fig. 1.** Semantics-based Framework for Integration of ADLs

Thus, in our integration framework, we first design a semantic structure (in denotational semantics terminology, these are called *semantic algebras* [19, 20]) that represents architectural information as a series of ground predicates (these predicates can be thought of as database records). Each ADL is denotationally mapped to this semantic structure. Next, we design a query language, called ADLQL, that can be used to query the information coded in these predicates. The query processor, in fact, is also realized by using a denotational approach in which the denotational semantics of the query language is specified using Horn logic. The denotational semantics maps a query into a Horn clause query, which can then be executed against the semantic description of an ADL using a logic programming interpreter (see Figure 1).

More concretely, consider a software architecture specification, $\mathcal{S}$, written in an ADL, $\mathcal{A}$. Consider an ADLQL query program, $\mathcal{Q}$, that queries this specification. Using the semantic specification of $\mathcal{A}$, the denotations of $\mathcal{S}$, say $\mathcal{D}_s$, can be obtained. Using the semantic specification of ADLQL, the denotation of the $\mathcal{Q}$, say $\mathcal{D}_q$, can be obtained. Both $\mathcal{D}_s$ and $\mathcal{D}_q$ are represented in Horn logic. Thus, using a logic programming interpreter, the denotation $\mathcal{D}_q$ can be *executed against* the denotation $\mathcal{D}_s$ of the architectural specification to obtain the answers to the query $\mathcal{Q}$.

The denotational semantics of the ADLs as well as of ADLQL can be rapidly specified using our approach. This is due to the *declarative* nature of denotational semantics. For example, it took us less than 1 man month of work to specify the semantics of entire ACME. Thus, given the rapidity with which these executable semantic specifications can be developed, if a new ADL is designed, it can be quickly integrated. Likewise, if the design of an ADL evolves, the changes to the ADL can be quickly integrated.

Note that some language specific information may be lost if a general semantic based approach is used. As an example consider the case of ACME which supports multiple inheritance, i.e., ACME allows an architectural component to inherit properties from multiple components. Thus, if we consider a `Component C={Port p;}` which inherits `{Property number:int=5;}`, then while semantically translating ACME to the database of predicates, the information about who inherited from whom will be lost. The database will, however, record the property for each component that inherits it, so the information recorded will be "computationally equivalent." Therefore, internally the component will be translated as if the property was defined within it: `Component C={Port p; Property number:int=5;}`. Also, as noted earlier, some architectural information described by one ADL may not be expressible in another. For example, it is hard to express CSP part of Wright in Aesop. In our integration framework, such parts that do not have a counterpart in the target ADL are omitted from the output.

The query language, ADLQL, has been designed with two purposes in mind. The first is to retrieve both extensional as well as intensional information from the semantics of an architectural specification. The second is to format the result of the query using the syntax of the ADL desired. This formating facility can also be used to simply translate an architectural specification written in one ADL $A_1$ to another ADL $A_2$. Of course, translation of an $A_1$ specification to $A_2$ specification is possible only if, semantically speaking, $A_1$ is a *subset* of $A_2$. Thus, ADLQL permits an architect to extract information of interest from a specification, and outputs the results in the syntax of another ADL. The output can then be fed to tools specific to the target ADL for further analysis. This approach is more flexible than the one where a literal translation of a specification written in one ADL to another one is done.

Note that the query language, ADLQL, that we have designed to extract architectural information from the documents is independent of any ADL. The use of a uniform query language for retrieving architectural information has the advantage that software architects don't have to be conscious of the syntax of original specification, information is retrieved from predicates that capture the semantics of the specification. Also, due to the decoupling between the query language and any specific ADL, the limitations of any specific ADL do not affect the information that can be inferred from an architectural description.

### 2.2 Horn Logical Denotational Semantics

Denotational semantics of a language consists of three components: *syntax specification*, *semantic algebras*, and *valuation functions* [19, 20, 18, 21]. The syntax specification of a language is usually given as a context free grammar. Semantic algebras are the basic domains along with associated operations. The meaning of a program is expressed in terms of these basic domains. Semantic valuation functions are mappings from syntax trees to corresponding semantic values in the semantic algebras.

Traditionally syntax specification in denotational definitions has been given in the BNF format, while the semantic algebras and valuation functions are described in the lambda-calculus. We use Horn clause logic (pure Prolog or logic programming) [21] instead for specifying all three components [14]. The main advantage offered by logic programming is that even syntax can be expressed in it at a very high level, and a parser for the language is immediately obtained from the syntax specification. Generation of the syntax trees requires a trivial extension to the syntax specification. The semantic algebras and valuation functions are also expressed in Horn logic quite easily. All three components of the semantics are executable, and thus the Horn clause semantic description provides a complete interpreter for the language under consideration.

Horn logical semantics also provides a basis for language translation. Essentially, the meaning of a source language $L_s$ can be expressed in terms of the constructs of the target language $L_t$. The meaning consists of both syntax and semantic specifications. If these syntax and semantic specifications are executable, then the specification itself acts as a translation system, providing a provably correct translator. Translating a program/specification written in $L_s$ to $L_t$ consists of three parts: DCGs for parsing programs written in $L_s$ and $L_t$ respectively, and the appropriate valuation predicates which map parse tree of $L_s$ to parse tree of $L_t$. Suppose $P_s(S_s, T_s)$ to be the top level predicate for the DCG of $L_s$ that takes a sentence $S_s$ of $L_s$, parses it and produces the parse tree $T_s$. Let $P_t(S_t, T_t)$ be the top level predicate for the DCG of $L_t$ that takes a sentence $S_t$ of $L_t$, parse it and produces the parse tree $T_t$. Let $M_{st}(T_s, T_t)$ be the top level valuation predicate that relates parse trees of $L_s$ and $L_t$, then the `translate` predicate can be used for obtaining $S_t$ given $S_s$, and *vice versa.*

$$\texttt{translate}(S_s,\ S_t)\ \texttt{:-}\ P_s(S_s,\ T_s),\ M_{st}(T_s,\ T_t),\ \texttt{Pt}(S_t,\ T_t).$$

We do not directly translate one ADL to another ADL in our framework, rather we introduce an intermediate structure, the database of predicates. This database captures the semantics of the ADL as discussed above. Also, the ADLQL queries are translated into queries over this database of predicates to extract information. Thus, in our framework, the `translate/2` predicate transforms to:

$$\begin{aligned}\texttt{translate}(S_s,\ S_t)\texttt{:-}\ &\texttt{Pr}(S_s,\ T_s),\ \texttt{BuildDatabase}(T_s,\ \texttt{Database}),\\ &\texttt{BuildDatabase}(T_t,\ \texttt{Database}),\ \texttt{Pt}(S_t,\ T_t).\end{aligned}$$

where the variable `Database` refers to the database of predicates. The predicate $\texttt{BuildDatabase}(T_s,\ \texttt{Database})$ refers to the top level valuation predicate that maps parse trees of $L_s$ to this database. Likewise $\texttt{BuildDatabase}(T_t,\ \texttt{Database})$ refers to the top level valuation predicate that maps parse trees of $L_t$ to the database.

## 3 Structure of the Database of Predicates

We have used the integration framework above to develop the semantics of ACME [2, 1], an interchange language for ADLs. ADML [9], the XML rendering of ACME, has also been integrated. A preliminary design of ADLQL has also been completed, and its semantics developed. The definition of ADLQL will evolve as we integrate more and more ADLs into our framework. At present, a complete semantics for ACME, ADML and our preliminary design of ADLQL have been developed. In the near future, we plan to integrate Wright, then Rapide and Aesop.

In this section we discuss the structure of the predicate database. To explain the details we first consider a very simple example written in ACME [2]. The example describes a simple client/server structure [1] as two components, connected by a single connector called `rpc`.

```
System simple_cs =
 {
  Component client =
   {
    Port send-request;
    Properties {request-rate:float = 17.0;
    source-code:external-file = 'Code-Lib/client.c'}
   }
  Component server =
   { Port send-request;
     Properties
      {idempotence:boolean = true;
       max-concurrent-clients:integer = 1;
       source-code:external-file = Code-Lib/server.c'
      }
   }
  Connector rpc =
   {
    Role caller;
    Role callee;
    Properties
      {synchronous:boolean = true;
       max-roles:integer = 2;
       protocol:Wright = 'omit'
      }
   }
  Attachments
   {
    client.send-request to rpc.caller;
    server.receive-request to rpc.callee
   }
 }
```

The semantic interpreter that we have developed for ACME will map the information in this specification to a database of predicates. These predicates essentially represent the relationship between various entities specified in the document. Each architectural entity is represented by a specific relation name in this database of predicates. The whole system is described by a relation called "system". The "system" relation has three arguments. The first argument is a list of string names used for identifying the system in the hierarchy (software architecture are hierarchically specified in ACME), i.e., one element maybe contains several other elements. The list of names can be thought of as uniquely identifying the element in the name space of elements. This list also helps us to find the relationships among elements. For instance in the example above, the identifier `[simple_cs, server]` indicates that that `server` is contained in an element called `simple_cs`. The second argument indicates the type of system, while the last argument is a reference to the semantic description of the system's body. Each system body has a unique identifier, and is described by a relation called `systemBody`. `SystemBody` in turn has three arguments: the identifier, body type, and elements that constitute the system body. In this example, we can see there are four elements in the system body: `client`, `server`, `rpc` and `attachment`.

The structure of the semantic predicates for the simple client server example above generated by the Horn logical semantic description of ACME is shown below.

---

```
acmeADL(acmeADL9, [none_import_define], [[simple_cs]]).
system([simple_cs], [none_type_appoint], systemBody8).
systemBody(systemBody8, [none_type_appoint], [[simple_cs, client],
            [simple_cs, server], [simple_cs, rpc], [simple_cs, attachment5]]).
component([simple_cs, client], [none_type_appoint], componentBody2).
component([simple_cs, server], [none_type_appoint], componentBody3).
componentBody(componentBody2, [none_type_appoint],
            [[simple_cs, client, 'send-request'], [simple_cs, client, 'request-rate'],
               [simple_cs, client, 'source-code']]).
componentBody(componentBody3, [none_type_appoint], [[simple_cs, server,
    receive-request'], [simple_cs, server, idempotence], [simple_cs, server,
        'max-concurrent-clients'], [simple_cs, server, 'source-code']]).
port([simple_cs, client, 'send-request'], [none_type_appoint], [none_body_define]).
port([simple_cs, server, 'receive-request'], [none_type_appoint], [none_body_define]).
property([simple_cs, client, 'request-rate'], float, 17.0, [none_body_define]).
property([simple_cs, client, 'source-code'], 'external-file',
                 'CODE-LIB/client.c', [none_body_define]).
property([simple_cs, server, idempotence], boolean, true, [none_body_define]).
property([simple_cs, server, 'max-concurrent-clients'], integer, 1, [none_body_define]).
property([simple_cs, server,'source-code'],'external-file',
                 'CODE-LIB/server.c',[none_body_define]).
connector([simple_cs, rpc], [none_type_appoint], connectorBody4).
connectorBody(connectorBody4, [[simple_cs, rpc, caller], [simple_cs, rpc, callee],
            [[simple_cs, rpc, synchronous], [simple_cs, rpc, 'max-roles'],
               [simple_cs, rpc, protocol]]).
role([simple_cs, rpc, caller], [none_type_appoint], [none_body_define]).
role([simple_cs, rpc, callee], [none_type_appoint], [none_body_define]).
property([simple_cs, rpc, synchronous], boolean, true, [none_body_define]).
property([simple_cs, rpc, 'max-roles'], integer, 2, [none_body_define]).
property([simple_cs, rpc, protocol], 'Wright', omit, [none_body_define]).
attachment([simple_cs, attachment5], [none_binding_define],
            [attachBody6, attachBody7]).
attachBody(attachBody6, ([client, 'send-request'], [rpc, caller]),
```

```
               [none_property_define]).
attachBody(attachBody7, ([server, 'receive-request'], [rpc, callee]),
               [none_property_define]).
```

The structure of the database predicates is obvious from the example above, however, more details can be found elsewhere [16].

## 4 The Query Language ADLQL

Once an ADL specification has been semantically transformed into a database of predicates, its properties can be checked by the architect. The checking of properties is done by posing a query in the query language called ADLQL that we have designed.

ADLQL is inspired by structured query languages for databases. ADLQL is a declarative language. Thus, architects just need to express the information that needs to be obtained. ADLQL's declarative nature also makes it easy to use and learn. It is unrealistic to assume that a user will have complete knowledge of the content and structure of software architecture. Thus, ADLQL allows queries based on zero or partial knowledge.

ADLQL has been designed to query architectural information (that is explicitly or implicitly present in the design) from the predicate database. Thus, architects can retrieve the basic information about an architectural specification written by them or some one else. For example, they can get a list of components contained in the current system or a list of subcomponents of a particular component. They can retrieve properties that are included in a component, etc. They can also retrieve information related to the structure of the software architecture. In fact, the main intended use of ADLQL is for checking the structural integrity of software architectures.[1] Structural integrity pertains to checking for things such as whether two components are directly or indirectly connected or not, or if there is a component that sends a message then there is at least one other component that receives a message, or whether port and the component are consistent or not, or whether a component and affiliated port are compatible or not, etc. Some of these integrity checks will be standard (similar to design rules), while some of them will be specific to a particular software architecture.

ADLQL also allows the results of a query to be formatted according to user's requirements. That, the user can format the output in the syntax of another ADL. Currently, the output can be formatted in ACME or ADML because our current system at present integrates these two ADLs only. This formatting capability can also be used to translate one ADL specification into another, as illustrated later. The general form of an ADLQL query is as follows:

[1] Here one can invoke the analogy to a building's architecture. Once an architect designs a building s/he will be interested in checking that the design satisfies all sorts of design rules, e.g., all rooms have a door that opens to a corridor and not to another room, or all rooms have a toilet/bath in their proximity, etc.

```
SELECT ArchitectureExpression
  FROM ADL_DOCUMENT
  WHERE WhereClauseExpression
  CONSTRUCT OutputFormat
  USING NameSpace
```

where `SELECT` clause describes the type of architectural information to be retrieved. Currently there are eleven predefined types. These are: *system*, *component*, *connector*, *role*, *port*, *property*, *attachment*, *path*, *contain*, *substitution* and *connect*. The `FROM` clause denotes the source of internal database to be queried. The `WHERE` clause indicates the constraints that the information retrieved should satisfy. The `CONSTRUCT` clause defines the format to be used for presenting the results of the query (currently only ACME and ADML are supported). Finally, the USING clause declares the name space that will be used for the query language. The current EBNF definition of ADLQL can be seen in Appendix I.

ADLQL queries are also converted to Horn Logic queries, using our denotational semantics and Horn logic based method described earlier. A syntax specification for ADLQL as a DCG is developed. Given a query of ADLQL, its denotation can be viewed as a query coded in logic programming. The logic programming coded query can be executed on top of logic programming coded predicate database obtained denotationally from the ADL specification.

## 5 Implementation and Examples

We have realized our integration framework starting with ACME. A complete denotational semantics of ACME has been developed. A preliminary design of ADLQL, that is powerful enough to verify structural properties, has also been completed and its Horn logical denotational semantics developed. Note that both the semantics have been fully implemented using Horn logic. The Horn logical semantics of (a subset of) ADML has also been developed. Thus, our integration framework that incorporates ACME, ADML, and ADLQL is fully operational (it will be publicly released shortly; in the mean time the system can be obtained by writing to the authors), and can be used to verify the structural integrity of architectural designs.

We omit the details of the semantics of ACME, ADML, and ADLQL due to lack of space. The details can be found elsewhere [16, 15]. We next illustrate the usage of our framework via examples.

**Example 1:** Consider a very simple query first where we are interested in simply retrieving some architectural information from the example architectural specification `simple_cs` shown in Section 3:

```
SELECT component(?ComponentName)
  FROM   simple_cs
  WHERE (?ComponentName = client).
  CONSTRUCT ADML
  USING simple_cs.
```

The query above asks for all the information of a component called `client` from the specification `simple_cs` with the output being formatted in ADML (note that variable names are preceded by a question mark). The semantic interpreter of ADLQL will map this query to a Horn logic query, which when run against the property predicate produced by the semantic interpreter of ACME for the `simple_cs` specification will produce the result shown below:

```
<AcmeComponentDeclaration id = 'client'>
    <AcmeComponentDescription>
        <AcmePort id = 'send-request'>
        </AcmePort>
        <AcmeProperties>
        <AcmeProperty id = 'request-rate'
            type = 'float'> 17.0
        </AcmeProperty>
        <AcmeProperty id = 'source-code'
            type = 'external-file'>
            'Code-Lib/client.c'
        </AcmeProperty>
        </AcmeProperties>
    </AcmeComponentDescription>
</AcmeComponentDeclaration>
```

**Example 2:** Next, we consider an example that translates the whole `simple_cs` architectural specification from ACME to ADML. To accomplish this, one can pose the following ADLQL query:

```
SELECT System(?SystemName)
   FROM simple_cs
   WHERE ?SystemName = simple_cs
   CONSTRUCT ADML
   USING root.
```

This query is quite self explanatory. The constant `root` denotes the top element in the name space that is above every name; its presence implies that the query will search the system information starting from the top level of name space.

**Example 3:** Next, we illustrate how ADLQL can be used to query structural information implicit in a specification. Consider a system with a client-server architecture; the following is an example of a query that checks if the server connects to multiple clients with a star like configuration (with the server at the center of the star).

```
SELECT component(?Server)
     FROM example3
     WHERE (forall ?Client in System.components |
       exist  ?Server in System.components |
       connected(?Server,?Client))
     CONSTRUCT ADML
     USING root.
```

`Forall` and `exist` denote universal and existential quantification respectively. The predicate of `connected(C1,C2)` denotes that the component `C1` is directly connected to component `C2` by a connector (full BNF of ADLQL is given in the appendix). The output of the query will be the ADML formatted definition of the server component that satisfies this structure. If no such server exists, the query will report a failure.

**Example 4:** Similarly, we can check if one component in the architecture is reachable from another. Consider the architecture of a compiler that consists of five pipelined components: lexical analyzer, parser, semantic analyzer, optimizer and code generator. The query below can be used to check if there is a direct or an indirect path from the component `lex` component to component `codegen` via connectors:

```
SELECT path(?C1,?C2,?Path)
   FROM example2
   WHERE ?C1= lex and ?C2=codegen and reachable(?C1,?C2,?Path)
   CONSTRUCT ADML
   USING root.
```

The constraint `reachable(C1,C2,Path)` holds if there exists a direct or indirect connection between components `C1` and `C2`. Per the semantics of ADLQL, a list of intervening components between the components `lex` and `codegen` will be output.

From these examples, one can see the utility of ADLQL. Once an architecture has been specified, the software architect can query the design in all sorts of ways to confirm the structural integrity of the design. The design of ADLQL is still evolving; the goal is to design a language that can express most properties that a software architect would be interested in verifying.

## 6 Related Work

ACME itself is an effort in the direction of integrating ADLs [2, 1]. ACME is an interchange ADL developed by CMU; it provides a simple structural framework for representing architectures, together with a "liberal annotation mechanism" [1]. ACME does not impose any semantic interpretation on an architectural description, rather it provides syntactic structures to represent semantic descriptions. This semantic information is to be interpreted by analysis tools. Our work can be regarded as a step further in this direction, except that we use

Horn logic to hold the semantic representations, and we build semantics based translators, that are also specified in Horn logic, to derive the semantic representations from ADL specifications. Since Horn logic is equivalent to a Turing complete programming language, any architectural feature can be semantically captured (even timing related and concurrency related properties can be represented and reasoned with by using advanced logic programming techniques such as constraint and tabled logic programming [13, 17]).

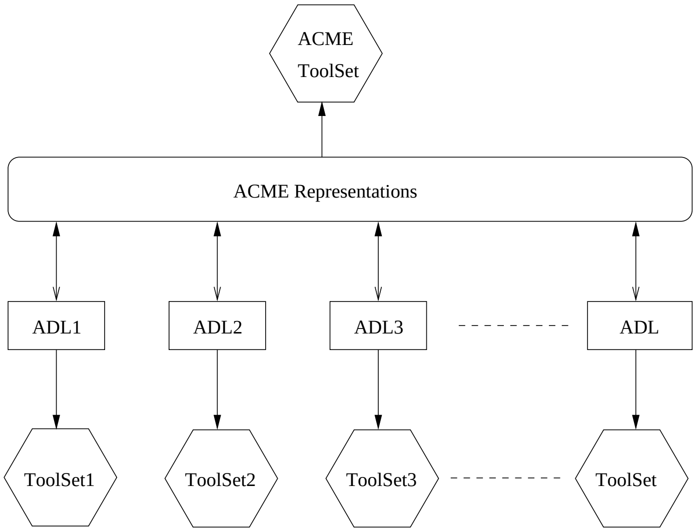


**Fig. 2.** ACME Based Integration

In ACME, an architectural design is shared among several ADLs by first translating the design into an ACME representation. This representation can be read by other ADLs that understand ACME, or it can be manipulated by tools that operate on ACME directly (see Figure 2). ACME can be thought of as an common interchange language which attempts to provide all the common architectural features. However, designing a language that captures all architectural features is hard, as it can result in a language with a very large definition. In contrast, by using Horn logic—a very general notation—any architectural feature can be represented. There are three main differences between ACME and our approach:

1. ACME defines an interchange ADL, in contrast, we define a general semantic representation. The definition of ACME is inspired by Wright, Rapide and Aesop, and thus integrating these ADLs is easy in the ACME framework. However, integrating other ADLs may require extensions to ACME.
2. We develop a general and uniform query language that can be used to query architectural information implicit in a design. While this query language is

still evolving, it can be made powerful enough so as to capture the functionality of most analysis tools. Thus, rather than building a custom tool for each kind of analysis, a query program can be developed instead.
3. By using Horn logic, translators that work both ways can be implemented with relative ease (provided translation is semantically possible). Thus, the semantic translator that is used to map an ADL to the database of predicates can also work in the reverse direction, i.e., this translator will also map database of predicates to the ADL. Thus, a single translator works in both ways, reducing the amount of effort involved in building translators.

Thus, w.r.t. the criteria for an ideal integration environment, developed earlier, an ACME-like approach only satisfies criteria 2, satisfies criteria 3 only partially, and does not satisfy criteria 1. Our semantics-based framework, in contrast, satisfies all 3 criteria.

Another effort in the direction of integration is the development of ADLs based on the Extensible Markup Language (XML) notation. These include xADL [11] and ADML[9]. xADL has been developed at the University of California at Irvine as an ADL-neutral interchange format. It is designed as a shared language for representing a variety of possible ADLs, and includes five kinds of basic tags: ⟨`Architecture`⟩, ⟨`Component`⟩, ⟨`Connector`⟩, ⟨`ComponentType`⟩, and ⟨`ConnectorType`⟩. xADL can be customized to different ADL styles; it has been customized to the style of C2 [10, 12], an architecture-centric integrated development environment. ADML, discussed earlier, is another XML based interchange notation [9]. ADML can be regarded as, essentially, an XML rendition of ACME, so the issues discussed above for ACME also apply to ADML.

Our effort bears some similarity to the evolving algebra approach espoused by Gurevich and extensively applied to practical scenarios by Börger and his research group [22]. Our semantic maps can be regarded as declarative specification of evolving algebras.

## 7 Conclusions

In this paper, we proposed a new framework for integrating multiple ADLs. This framework is based on denotational semantics and is considerably more flexible than other approaches developed so far. We developed a query language, called ADLQL, for querying architectural information that allows software architects to reason about the whole architectural design, even though different parts of the design may have been specified using multiple ADLs.

We demonstrated our approach by developing the Horn logical denotational semantics for ACME, ADML and ADLQL. An architecture specification written in one ADL can be readily translated into specification of another ADL due to the use of Horn clause logic for coding their semantics. The implementation infrastructure for integrating an ADL can be developed extremely quickly. Thus, as an ADL rapidly evolves, its implementation infrastructure can be developed at the same pace. This integration infrastructure that has been developed will be publicly released shortly; at present it can be requested from the authors.

## References


1. David Garlan and Zhenyu Wang. A Case Study in Software Architecture Interchange. In Proceedings of Coordination'99, Springer Verlag, 1999.
2. David Garlan, Robert T. Monroe and David Wile. Acme: An Architecture Description Interchange Language. In *CASCON'97*, pp. 169-183. 1997.
3. Robert Allen and David Garlan. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology, July 1997.
4. David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. In *SIGSOFT'94*, pp. 179-185.
5. Nenad Medvidovic and Richard N. Taylor. Architecture description languages. In *ESEC/FSE'97* Software Engineering Notes 22(6).
6. David C Luckham, et al. Specification and analysis of system architecture using Rapide. IEEE TSE, Special Issue on Software Architecture, 21(4):336-355. 1995.
7. D. Garlan and M. Shaw. "An Introduction to Software Architecture". In *Advances in Software Engineering and Knowledge Engineering.* World Scientific, 1993.
8. D. Garlan and D. Perry. "Introduction to the special issue on Software Architecture". IEEE Transactions on Software Engineering, 21(4), April 1995.
9. Open Group Technology Report. "Architecture Description Markup Language (ADML)", available online at http://xml.coverpages.org/adml.html, June 2000
10. N. Medvidovic, et al. Using object-oriented typing to support architectural design in the C2 style. SIGSOFT'96. pp. 24-32.
11. Rohit Khare et al. xADL: Enabling Architecture-Centric Tool Integration With XML. In *Proc. 34th HICSS*, 2001.
12. Taylor, R. N., Medvidovic, N., et al. A component and message-based architectural style for GUI software. IEEE Trans. On S/W Engineering, 22(6):390-406.
13. G. Gupta, E. Pontelli. "A Constraint-based Approach to Specification and Verification of Real-time Systems," *IEEE Real-time Systems Symposium '97*, pp. 230-239.
14. Gopal Gupta. Horn logic denotations and their applications. In The Logic Programming Paradigm: A 25 year perspective. Springer-Verlag, 1999:127-160.
15. Q. Wang, G. Gupta. A Semantics-based approach to integrating Architecture Description Languages. UT Dallas Computer Science Tech Report, June 2004.
16. Q. Wang. A Semantics-based approach to integrating Architecture Description Languages. Ph.D. thesis, forthcoming.
17. Y. S. Ramakrishnan, C. R. Ramakrishnan, I. V. Ramakrishnan et al. Efficient Model Checking using Tabled Resolution. In *Proceedings of Computer Aided Verification (CAV'97).*
18. C. Gunter. Programming Language Semantics. MIT Press. 1992.
19. D. Schmidt. Denotational Semantics: a Methodology for Language Development. W.C. Brown Publishers, 1986.
20. D. Schmidt. Programming language semantics. In CRC Handbook of Computer Science, Allen Tucker, ed., CRC Press, Boca Raton, FL, 1996. Summary version, CM Computing Surveys 28-1 (1996) 265-267.
21. L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, 1994.
22. E. Börger Linking Architectural and Component level system views by ASMs. In *Proc. Forum on Specification and Design Languages '03.*

# Appendix I: EBNF for ADLQL

**The EBNF for ADLQL is included only for the convenience of the reviewers. It will not be included in the final paper, if accepted.**

```
adlQuery::=
    <SELECT> architectureExpression
    (<FROM>  adl_document)?
    (<WHERE> whereClauseExpression)?
    <CONSTRUCT> outputFormat
    (<USING>     (nameSpace|root))?
architectureExpression ::= system(element_expr)
    | component(element_expr)
    | connector(element_expr)
    | role(element_expr)
    | port(element_expr)
    | property(element_expr)
    | attachment(element_expr)
    | path(element_expr,element_expr,element_expr_list)
    | contain(element_expr,element_expr)
    | contain(element_expr,element_expr_list)
    | substitution(element_expr,element_expr)
    | connected(element_expr,element_expr_list)
    | connected(element_expr,element_expr)
    | reachable(element_expr,element_expr,element_expr_list)
element_expr::= var_name|id
element_expr_list ::= <[>(element_expr)*<]>
var_name ::= ? <IDENTIFIER>
nameSpace::=id <.> (id)*|id
var_list::=var_name|var_name,var_list
adl_document::= id
    |<component><(>id<)>
    |<connector><(>id<)>
    |<system><(>id<)>
    |<role><(>id<)>
    |<port><(>id<)>
outputFormat::= <acme>|<adml>
whereClauseExpression ::=   quantifiedExpression
                          | booleanExpression
quantifiedExpression ::=
    ( ( <FORALL> | <EXISTS> )
          (<IDENTIFIER>|var_name) <IN>
    SetExpression <|> WhereClauseExpression )
SetExpression ::= (<sys>|<IDENTIFIER>)<.>
    (<conponents>|<connectors>
       |<roles>|<ports>|<properties>)
BooleanExpression ::=
    OrExpression ( <AND> OrExpression )*
```

```
OrExpression ::= ImpliesExpression
    ( <OR> ImpliesExpression )*
ImpliesExpression ::=
    EqualityExpression
    ( <IMPLIES> EqualityExpression )*
EqualityExpression ::=
    RelationalExpression
    ( <EQ> RelationalExpression
      | <NE> RelationalExpression )*
RelationalExpression ::=
    AdditiveExpression
    ( "<" AdditiveExpression
      | ">" AdditiveExpression
      | <LE> AdditiveExpression
      | <GE> AdditiveExpression )*
AdditiveExpression ::= MultiplicativeExpression
    ( <PLUS> MultiplicativeExpression
      | <MINUS> MultiplicativeExpression )*
MultiplicativeExpression ::=
    UnaryExpression
    ( <STAR> UnaryExpression
      | <SLASH> UnaryExpression
      | <REM> UnaryExpression )*
UnaryExpression ::=
    <MINUS> UnaryExpression
    |PrimitiveExpression
PrimitiveExpression ::=
    "(" WhereClauseExpression ")"
    | LiteralConstant
    | Reference
    |var_name
Reference ::=
    <IDENTIFIER>  ( "." <IDENTIFIER> ) *
LiteralConstant ::= ( <INTEGER_LITERAL> )
       | ( <FLOATING_POINT_LITERAL> )
       | ( <STRING_LITERAL> )
       | ( <TRUE> )
       | ( <FALSE> )
       | ( <COMPONENT> )
       | ( <GROUP> )
       | ( <CONNECTOR> )
       | ( <PORT> )
       | ( <ROLE> )
       | ( <SYSTEM> )
       | ( <ELEMENT> )
       | ( <PROPERTY> )
       | ( <INT> )
       | ( <FLOAT> )
       | ( <ANY> )
       | ( <BOOLEAN> )
```