LEADS: Language for Exploiting Architectural and Design Specifications

Bharath Kumar M, Y N Srikant
Department of Computer Science and Automation,
Indian Institute of Science
{mbk, srikant}@csa.iisc.ernet.in

Abstract

Precise specification of the architecture and design of software is a good practice. Such specifications contain a lot of information about the software that can potentially be exploited by tools, to reduce redundancy in software writing by automating routine tasks, as well as giving valuable feedback on the software. In LEADS, we propose a language and environment that is designed to make writing such tools a lot easier. LEADS is based on the Pattern-Action approach, where one specifies the pattern of information of interest and the actions to be taken when it is found. LEADS decouples itself from the specification environments/ formats thereby ensuring wide applicability. Here we discuss the language features of LEADS that make tool writing a lot more organized and modular.

Keywords:

Specifications, CASE Tools, Software Architecture, Knowledge based software engineering.

1. Introduction

With the advent of many standard notations and practices, many industry standard specification environments have come up. The software specifications[1] are typically stored as an object hierarchy. Specification environments often offer accessibility interfaces to the object hierarchy. The accessibility interfaces allow for extensions and customizations to the environment to suit specific needs, and also allow third party tools and power users to exploit the information content stored in them E.g.: [1]. In the context of architectural and design specifications, we highlight some of the possible ways in which such tools[2] can exploit the information.

- Automatic Code Generation – This can be done when we are dealing with small parts of the system that are understood and specified sufficiently enough to completely automate their code generation. This is normally possible when domain specific components and connectors are used, since such components and connectors are well defined and known.
  Other examples include automatic code generation for data structures; generation of IDL[3] specifications from component diagrams; and application of the singleton pattern over a class.
- Design Guidance – This can be done when we do not have enough knowledge to completely automate the generation of code, but have a good knowledge of the concerns in design and popular solutions to the same.
  Tools that suggest usage of certain design patterns to decouple change in the given contexts come under this category. Some other examples include tools that could assist the designer in identifying possible layers in a complex system of classes; and tools that analyze the design to give good test sequences.
- Analysis – The specifications act as a blue print for the software that is going to be produced. So, an early analysis could be very useful. Popular analysis techniques include calculating metrics or conducting simulations so as to give feedback about the software's robustness and feasibility.

Clearly, in all the above-mentioned means of exploiting information in specifications, the tools have to go through a two-phase process. They first search for information of interest in the specifications, and then take appropriate actions. The programming languages in which the tools are written do not offer any special constructs that make the search easy. Most often, the programs will have to work in and out of the complex object hierarchy in which the information is organized.

In LEADS, we use a Pattern-Action approach. In a LEADS specification, a tool writer has to simply specify the kind of information he is looking for, as a pattern, and the action to be performed when the information is found in a specification. The patterns are specified in a declarative language, using which one specifies the constraints that hold amongst the participants of the pattern. This declarative language offers special primitives that cater to the common needs of tools that work on architectural and design specifications. Since the potential ways of taking action on a pattern found could be very extensive, we generalize the actions as bits of C++ code. This approach is similar to that of yacc[2], where every time a grammar rule is parsed, a piece of code is executed.

In our design of LEADS, decoupling from the specification environment was of prime importance. Only this would ensure the applicability of LEADS over a wide domain of specification languages and environments. But the specification environment is where the facts, that are required for identifying the patterns, are provided. Hence we provide the type library, by which a mapping, from the types used in the LEADS specification to those present in the specification environment can be provided. Essentially, in a type library, one defines a new type of information and a means of obtaining it. This decoupled nature of

[1] When we use the term 'specifications', we refer to architectural and design specifications, unless mentioned otherwise.
[2] With 'tools' we refer to tools that exploit the information content in specifications.
[3] Interface Description Language for various Component Technologies like CORBA, COM

LEADS allows it to act as glue across specification environments thus helping exchange of information amongst various specification languages and environments.

The presence of the type library also ensures that LEADS specifications are reusable across specification environments that just differ in their syntax. Only the type library needs to be changed to make the LEADS specifications work with different specification environments.

We now discuss the design and languages features of LEADS through an example.

2. An Example

Large designs often have a large number of classes. A tool that could introduce some kind of organization and hierarchy is useful. Say, the tool analyses the class diagram, looks for dependencies amongst the classes, organizes the classes into layers and then organizes these layers themselves in a hierarchy based on relative dependencies. The tool puts classes that are interdependent to each other in the same layer. Then, the tool creates an ordering based on the dependency graph amongst the layers. We will now illustrate how this tool can be written using LEADS. Assume that the types used in the following script are from a specification environment that allows class diagram specifications in UML[3].

```
import "UML.tlb"

relationship BidirectionalDependency(c1 : Class,
                                     c2 : Class)
constraints
    (c2 in
        transitive_closure(c1.associations.role2.class)
    and c1 in
        transitive_closure(c2.associations.role2.class))
end

pattern Layer(m : Model)
participants
    classes : set(Class);
    position : integer
constraints
    range of classes = m.classes and
    classes = partition of m.classes based on
        relationship BidirectionalDependency
end

relationship LayerDependency(l1 : Layer, l2 : Layer)
constraints
    (exists c1, c2 : Class;
        range of c1 = l1.classes and range of c2 =
l2.classes
        and c2 in
        transitive_closure(c1.associations.role2.class))
end

pattern LayeredSystem(m : Model)
participants
    layers : set(Layer)
constraints
        layers = all Layer(m) and
        order layers on layers.position suchthat
            (l1,l2 : Layer; range of l1 = layers and
            range of l2 = layers and
            (l1.number < l2.number iff
             LayerDependency(l2,l1)))
action
    {
        // Code in C++ to list the layers
        for (set<Layer>::const_iterator i = layers.begin();
                            i != layers.end(); i++)
        {
          Layer *l = *i;
          cout << Layer position << l->position;
          for (set<Class>::const_iterator j
             = l->classes.begin(); j != l->classes.end(); j++)
          {
             Class *c = *j;
             cout << Class << c->name << endl;
           }
    }
end
```

LEADS provides two constructs for organizing the information searching. The **pattern** construct is used to specify a pattern of information as per its **participants**, the **constraints** that hold amongst them, and the **action** to be performed when the pattern is identified. The **relationship** construct is used to specify adequacy criteria for a relationship to exist amongst some elements.

In the example, our aim is to organize all the classes in a model into layers, and to number the layers based on their relative dependencies. We first define a *Layer*, to contain a set of classes and a number. This number (*position*) will essentially indicate the layer's position in the hierarchy. All interdependent classes form a single layer. So, each layer is essentially a partition of the entire set of classes. This is specified by the statement "classes = **partition of** m.classes **based on relationship** BidirectionalDependency". Here, *classes* is one of the parts or sections created as a result of the partitioning of *m.classes*. The relationship *BidirectionalDependency* is used to partition the set of classes. The statement indicates that the set of all classes in the model *m*, will be partitioned in such a way that all individual classes that fall into each partition will be related to each other by the relationship *BidirectionalDependency*. *classes* can be any of the partitions. Note that *BidirectionalDependency* accepts two parameters, both of type *Class*, which is the same type as the constituents of the partitions.

The relationship *BidirectionalDependency* between two classes is defined to exist if the two are either directly or indirectly dependent on each other. Indirect dependency is specified using the **transitive_closure** construct. *c1.associations.role2.class* is an expression that gives all classes that are involved in associations with *c1*. **transitive_closure**(*c1.associations.role2.class*) gives all transitively (indirectly) dependent classes too. Thus, *BidirectionalDependency* holds amongst two classes c1 and c2, if and only if, *c1* is either directly or indirectly

dependent on *c2*, and *c2* is either directly or indirectly dependent on *c1*.

Thus, the two constructs *Layer* and *BidirectionalDependency* enable us to partition the set of all classes, into a set of layers.

We then define another relationship, *LayerDependency* that will help us to determine if a layer is dependent on another. A layer *l1* is dependent on another layer *l2*, if and only if, *l1* contains at least one class which is either directly or indirectly dependent on a class present in *l2*. We again use the **transitive_closure** construct to specify this.

Finally, in *LayeredSystem*, we group all the layers that were obtained in a set. We then order this set based on the relative dependencies between the layers. Clearly there will be many instances of the pattern *Layer*. To group all such instances into a set, we use the keyword **all**. Thus, the statement "layers = **all** Layer(m)", finds all possible instances of *Layer(m)* and makes them elements of the set *layers*. We then order the elements of the set *layers* based on the relative dependencies between them. "**order** layers **on** layers.position **suchthat** …" does exactly that. The **order** construct is defined on sets, and a number is allotted to each element in the set, based on some conditions that determine how the numbering is done. In this example, the condition states that a layer *l1* gets a number less than another layer *l2*, only if *l1* is not dependent on *l2*. The **order** construct serves two purposes. One, it states that an ordering is done, explicitly. Two, since the possible numbers assigned to each element can potentially be infinite, the special construct allows the solver to constrain the domain, and will assign a numbering that starts from 1 only. In the **action** section, the code written in C++, simply generates a listing of each layer, and its position in the hierarchy.

The LEADS parser, after semantic checking, will generate classes (in C++) for each pattern. The following is an example of the prototypes of the classes that are generated.

```
#include "UML.h" // to provide the types Class, Model…
                 // (from the type library)

bool BidirectionalDependency(Class c1, Class c2);

class Layer
{
 public:
   set<Class> classes;
   int number;

   static set<Layer> & Match(Model m);
   void Action(Model m);
};

bool LayerDependency(Layer l1, Layer l2);

class LayeredSystem
{
 public:
   set<Layer> layers;
   static set<LayeredSystem> & Match(Model m);
   void Action(Model m);
};
```

Ultimately, LayeredSystem::Match(m) will find the pattern in the input specification and execute the Action, for each instance of the pattern that is identified. Match() also returns the set of all instances of the pattern found.

3. The Language

The important requirements that have influenced the design of LEADS are:

- Searching for patterns of information in the specifications should be easier, organized and modular.

  As discussed earlier, an important aspect of exploiting information, is to first identify the information of interest. An intuitive means of organizing this search, which would not require the tool writers to recurse in and out of the complex object hierarchies, is required. LEADS is based on the **Pattern-Action** approach, which separates the search from the action. Defining a **pattern**, which is essentially a set of **constraints** that hold on the **participants** of the pattern, specifies a search for information. A pattern can be accompanied by some **actions** that are to be taken when an instance of the pattern is identified. The problem of searching for information and subsequent actions to be taken can be decomposed into many phases (as patterns-actions and relationships) and each one tackled at a time. This approach makes the searching more modular.
- The language should offer constructs that capture the commonly encountered operations while exploiting information in architectural and design specifications.

  Exploiting architectural and design specifications involve certain operations that are done repeatedly. Checks for reachability, connectedness; operations that partition collections into smaller groups based on some desired properties; ordering collections based on some properties occur quite frequently. Hence, LEADS offers special primitives (see section 3.5) like **partition**, **order**, and some statistical functions that capture the typical needs of a tool that exploits information in architectural and design specifications.
- The language should be decoupled from the specification environment (which provides the data).

  This has been a very important design issue in LEADS. To maintain applicability over a wide range of specification languages and specification environments, the data types that can be used in the patterns should not be constrained by the specification environments. So, LEADS is designed in such a way that it can potentially be used with any specification environment. Types specific to a specification/ specification environment are defined in the **type library**. The type library is essentially a means of defining a new type of data, and a means of obtaining the data (from the specification environment).

3.1 Patterns

Often, while searching for information in a specification, the unit of information one is looking for, is not just a basic entity but a group of entities that may interact with each other in a desired way. Such groups of entities could all be of the same type, or of different types. We refer to these entities as **participants** of the pattern. The relationships amongst the participants are specified as **constraints** that need to be true for the participants to be part of the pattern. In any given specification, there could be several instances of a pattern. This essentially means that different combinations of participants could satisfy the constraints. For every such instance of the pattern, an **action** (which is a piece of C++ code) is executed. Patterns also take parameters to support communication across patterns. This essentially reflects as a parameter that is passed to the match function of the pattern when the C++ code is generated.

A pattern is also a type. This means that a pattern can have instances of patterns as participants, or use an instance in its constraints, or take an instance of a pattern as parameter (to its match function).

3.2 Relationships

The **relationship** construct is used to specify adequacy criteria for a relationship to exist amongst some elements. A relationship is specified as a set of constraints that hold amongst its parameters. A relationship always evaluates to true or false. Relationships allow for reuse of commonly occurring constraints amongst entities. Unlike a pattern, a relationship does not have participants, and cannot be instantiated. Relationships are also very useful when using the special primitives like *partition*.

3.3 Constraints

Constraints are used in the specification of both patterns and relationships. In general constraints are specified using a simple declarative language based on first order logic. Most of the common primitives like expressions; conditional operators like *implies*, *iff* (if and only if); set operations like *union*, *intersection*, *negation*, *difference*; testing membership in sets; operations over collections like *forall*, *exists*; are provided. The special primitives that make the constraint specification more specialized towards architectural and design specifications are discussed in a subsequent section.

3.4 Basic Types

LEADS offers the following basic types; *Bool*, *Integer*, *Real*, *String*, *Character*, *Date*. The vector types offered are *Collection*, *Set*, *Sequence* and *Bag*. Set, Sequence and Bag are essentially specializations of Collection with specific properties for different contexts. Instances of Set, Sequence and bag can be converted to each other type using type casting.

3.5 Special Primitives

LEADS offers some special primitives that make the specification of constraints easier, shorter and clearer.

range – Most of the variables used in LEADS are free variables. Potentially, there can be infinite values that these free variables can hold. To find instances of the pattern more efficiently, the *range* keyword is used to constrain the domain of the free variables. The range of a free variable has to be specified before it is used in any expression. Range constrains a free variable to contain values that belong to the collection specified in the expression.

all – A pattern can have many instances. When it is required to put all the instances of a pattern into a set, the *all* keyword is used. Thus “layers = **all** Layer(t)”, puts all possible instances of *Layer(t)* into the set *layers*.

transitive_closure – Transitive closure is used to obtain instances that are related to a particular instance through transitivity (indirectly). In general, any expression that maps an instance to another instance of the same type can be given as a parameter to this function. The function will eventually evaluate to a set that will comprise of the largest set that is reachable either directly or indirectly through the expression given as parameter.

Consider the clause **transitive_closure**(c**.**parents). Say, *parents* is an attribute in type *Class*, which aggregates all the parent classes of a particular class. **transitive_closure**(c**.**parents) represents a set of *Class* which contains the parents of *c*, the parents of the parents of *c* and so on.

partition - This clause essentially defines a partition on a collection based on certain conditions. The condition based on which the partitioning is done could be a relationship that exists amongst the members of each partition, or based on a value of an expression involving each member in the partition.

“classes = **partition of** m**.**classes **based on relationship** BiDirectionalDependency”, (discussed in the example), partitions the set *m.classes* such that members of each partition are related to each other by the relationship *BiDirectionalDependency*. *classes* can be any of the possible partitions.

“classes = **partition of** m**.**classes **based on valueof** m**.**classes**.**ClassKind”, says that the set *m.classes* is partitioned, based on the value the expression *m.classes.ClassKind* evaluates to. Thus, this will result in partitioning of *m.classes* such that each category of class (like NormalClass, ParameterizedClass, InstantiatedClass, etc.) is placed in separate groups.

order - The order clause is used to impose an ordering on the members of a set based on some conditions. Ordering does not necessarily mean a total ordering. Thus, it is not as simple as arranging a set as a sequence. To allow partial orders too, the order clause is defined such that a number is assigned to each member in the set that will denote its position in the order. Thus the data object to which the number is assigned is also specified in the order clause. The other reason for advocating the use of the order clause instead of using constraints directly is that, since the ordering is done on a numeric data object, there will be

infinitely many assignments that will satisfy the constraints. Through this special clause, a simple ordering that starts from 1 is assigned always. Consider the following example.

"**order** layers **on** layers**.**number **suchthat** (l1**,** l2 **:** Layer**;** **range of** l1 = layers **and range of** l2 = layers **and** (l1**.**number < l2**.**number **iff count**(l1**.**classes) < **count**(l2**.**classes))"

Here, *order* will assign numbers in *layers.number* such that the *layer* with the highest number of classes gets the highest number and the layer with the least number of classes gets the number 1.

Statistical primitives - In order to find and specify metrics easily, certain statistical functions are defined over collections, like **count**(), **max**(), **min**(), **avg**(), **median**(), **mode**().

3.6 The type library

An important design goal of LEADS is to maintain applicability over several specification languages and specification environments. So, the data types offered by a specification language cannot be part of the design of LEADS itself. A means by which a data type, present in a foreign environment, is declared and accessed needs to be provided. The type library serves this purpose.

The information provided by a specification environment is typically organized as an information structure, which is navigable from the top to the leaves. In general, a node can compose of an attribute (of some basic type), an instance of some aggregate type, collections of attributes or entities, or some functions. Many specification environments provide accessibility interfaces to the information structure (sometimes called the tool API).

In the type library, one essentially defines the information structure provided by the specification environment. Its not just enough if the information structure is declared. A means of obtaining this information every time something is accessed in the information structure needs to be provided too. This is important since LEADS has no knowledge of the presence of the specification environment, the kind of protocol it requires for communication, etc. So, for every element in the information structure, a ***get*** function has to be provided. The *get* function could just be a single line of code translating requests to the tool API, or could be parsing a specification itself.

LEADS requires to iterate through the collections while matching patterns. LEADS uses predefined collection types set, bag and sequence for this. So, if the tool API provides or uses a different implementation of collections, the collections have to be converted to the ones defined in LEADS in the *get* functions.

A type library needs to be written just once for every specification environment. A LEADS specification can use more than one type library at once, thus also allowing for exchange of information between the environments. With the OMG now standardizing the representation for UML through the XMI[4], a type library that understands/parses an XMI document will ensure LEADS is usable with many specification environments that support XMI. As part of the implementation of LEADS, type libraries that allow communication with Rational Rose[5], XMI aware tools, TriSL Architectural Development Environment[6] and ACMEStudio [7] tool will be provided.

A type library essentially has the following structure.

```
{
        /* Any C++ code, might be some Global
        Declarations, #include statements etc. */
}

type <name> [ extends <supertype> {, <supertype>}* ]

    internal
    begin
    /* Some declarations or functions(in C++) that are not
visible to LEADS specifications. Could be used to store
internal variables specific to the Specification
Environment */
    end

    external
    begin
    /* Attribute Definitions… */
        <Attribute name> : <Type>
        {
                /* Get function (C++) should return object
            of the same type */
        }

    /* Methods */
        <Method name>(<parameters>) : <Return Type>
        {
            /* Implementation */
        }
    end
end
```

To illustrate how a type library is defined we'll use some of the classes provided by ACME's tool API[7]. ACME offers a class called *ACMEParser* which parses ACME descriptions and organizes them into a *Design*. Each *Design* consists of several *System*s. Each *System* in turn offers methods that enumerate all the *Component*s, *Connector*s, *Port*s, *Role*s and *Attachment*s. The ACME library offers a class hierarchy called *Enumeration* to contain collections of each of these entities. These classes differ from the collection types offered by LEADS. In the ACME library, every class defines methods to access the entities it aggregates. This is not suitable for LEADS which defines set membership in its constraints. We shall write a small piece of the type library that would make LEADS applicable to ACME specifications.

```
/*---------------ACME.tlb-------------------------*/
{
#include <stdio.h>
```

```
namespace ACME
{
#include "ACME.h"
}
}

type Parser
    internal
    begin
        ACME::ACMEParser *p;
    end
    external
    begin
        getDesign(fname: String) : Design
        {
            FILE *fp = fopen(fname, "r");
            ACME::ACMEInputStream i(fp);
            p = new ACME::ACMEParser(&i);
            ACME::Design *d = p->parseDesign();
            Design * des = new Design;
            des->acmedesign = d;
            return(*des);
        }
    end
end

type Design
    internal
    begin
        ACME::Design * acmedesign;
    end
    external
    begin
        numSystems : Integer
        {
            return(acmedesign->getnumSystems());
        }
        systems : Set(System)
        {
            ACME::SystemEnumeration * allsys =
                new ACME::SystemEnumeration;
            acmedesign->getSystems(allsys);
            set<System &> * allsystems =
                new set<System &>;
            while (allsys->moreItems())
            {
                System *s = new System;
                s->acmesystem = allsys->getNextItem();
                allsystems->push_back(*s);
            }
            return(*allsystems);
        }
    end
end

type System
…
```

In the type library, we are essentially putting wrappers over the existing classes provided by the tool API so that the types are understandable to LEADS. Since, ultimately the LEADS parser will be generating C++ code for the type library too, there could be places where name collisions can occur (as in the example). So, we make use of namespaces and define all the types offered by "ACME.h" in the namespace ACME.

In the type Parser, we maintain a pointer to *ACME::ACMEParser* to which the *getDesign* method is delegated. *ACME::ACMEParser::parseDesign()* returns "*ACME::Design* *" which cannot be used by LEADS. Hence in *Design*, we maintain an internal variable which holds a pointer to *ACME::Design*, to which all requests are delegated.

ACME::Design provides a method called *getSystems()*. This method returns a pointer to *SystemEnumeration* which is a collection object defined in the ACME library used to hold *System*s. In the *get* function for *systems* (an attribute in *Design*), since LEADS cannot use *SystemEnumeration* as a collection type, we store all the elements of *SystemEnumeration* in a *set<System>* and return this set.

The type library can be completed proceeding on the same lines.

4. The Overall System Architecture

We now present a control flow diagram that depicts the logical organization of various components in LEADS.

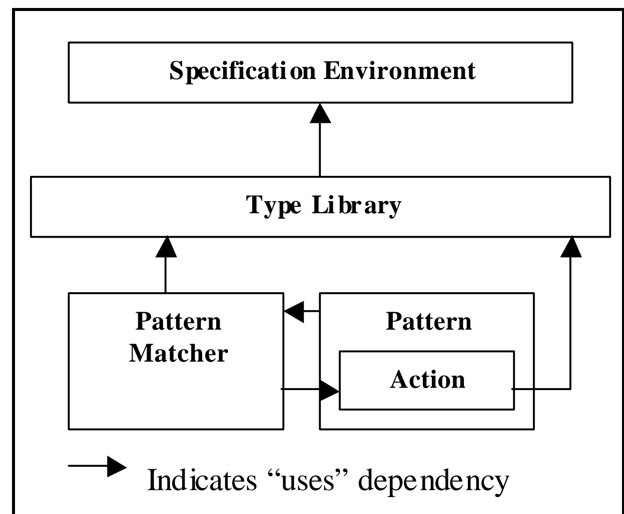

Figure 1

The *Type Library* is the interface between the LEADS environment and the specification environment. Every *Pattern*, to identify its instances uses the *Pattern Matcher*, which in turn uses the *Type Library*. For every matched instance, the *Action* is called, which could use the *Type Library*.

5. On the expressiveness of the language

Through LEADS, we wanted to make the information searching process easier and modular. Organizing the process into patterns is the basic underlying principle. We also took a declarative approach towards specifying the patterns, since we felt this was intuitive and simple. The patterns are specified as a set of constraints that hold on its participants. The constraint language is based on First Order Logic, and does not define relations over relations,

etc. However, since our aim is to make specifying patterns easier, we have enriched the language by adding some special primitives that don't necessarily fall under First Order Logic. This decision was driven by the fact that certain operations were done repeatedly in many exploitations. To name some, checks for reachability, connectedness, which are captured by the *transitive_closure* construct; ordering entities, so as to do things like diagram layout, getting test sequences, organizing the designs, which is captured by the *order* construct; partitioning sets of entities based on some properties of interest, which is captured by the *partition* construct. These primitives also make the patterns easier to express.

Newer primitives can be introduced into the language, by defining them in the type library (as methods of types), or by modeling them as patterns.

6. On the reusability of LEADS specifications

LEADS was designed keeping in mind that there are several specification environments from different vendors often supporting the same specification language. Scripts that exploit the information content in one specification environment, cannot be reused in another specification environment, if the API provided by the specification environment changed.

In LEADS only the type library depends on the specification environment. The same LEADS specifications can be used with a different specification environment, by simply altering the type library to suit the specification environment.

7. Related Work

In LEADS we incorporate ideas that have been successful in various domains, to exploit information in specifications. In this section, we summarize the related work in various domains that have influenced the design of LEADS.

Specifications

UML has been adopted by OMG as a standard for software specification. UML provides various views and diagrams where in software can be diagrammatically specified. The UML standard is obtainable from the OMG web site [8]. Several industry strength tools have emerged that support UML. A brief listing of various commercial tools and the features they offer can be found at [9]. Many of these tools offer extensibility interfaces by which one can access the information structure and the tool's functionality (Eg: Rose Extensibility Interface from Rational Rose Help).

The academic community has been looking into Software Architecture Description Languages. ACME[7] has been proposed as a means of achieving interchange across ADLs. ACME is supported by a tool, which provides accessibility API..

The OMG has recently proposed the XMI (XML MetaData Interchange)[4], by which several tools that implement UML can exchange information. OMG recommends that all tools that support UML provide for interchange using XMI. The XMI recommendation is available at the OMG Web Site[8].

Extensibility in Specification Environments

Rational Rose[5] provides a scripting language called Rose Scripts through which one can write scripts (in Basic programming language) that can exploit the information content in the specification being edited/ written in Rational Rose. These scripts essentially access the API provided by Rose. The script is coded in a programming language that does not make the search process easy. One has to work in and out of the information structure while trying to search for information of interest. Rose Scripts are also tightly bound to the Rational Rose tool. In contrast, LEADS is specially designed to make specification of searches easy. The patterns are specified in a declarative language that is intuitive and simple to use. LEADS is also designed in such a way that it maintains applicability to a wide range of specification environments and is not tightly bound to any single specification environment.

Patterns in Specifications

The SADL[10] architectural description language is designed to facilitate Architectural Refinement, from an abstract high level architecture to a more detailed concrete architecture. In [11], Mark Moriconi et. al. talk about Refinement Patterns that essentially provide mappings from a high level, less detailed pattern to a lower level, concrete pattern. Given such refinement patterns, the SADL system parses specifications written in SADL, looks for instances of the high level patterns and replaces them with more concrete instances, as defined by the refinement pattern. In LEADS, we try to generalize the problem of Architecture Refinement, by allowing a piece of C++ code to be written whenever a pattern is identified. The mapping/refinement to the lower level architecture can be implemented in the actions. This way Architectural Refinement can be achieved across specification languages and environments. So, potentially one could look at an architectural description (in an ADL-based environment) for high level patterns and refine them into a low level (say Design) specification in a different design environment.

In [12], Rick Kazman and Marcus Burth describe IAPR (Interactive Architecture Pattern Recognition) where user defined patterns can be searched within a software architecture. IAPR is described as a reverse engineering tool that can be used to understand software architectures by viewing them from higher levels of abstraction. IAPR provides GUIs, an Architecture Modeler and a Pattern Modeler for user specification of the software architecture and patterns respectively. In LEADS, we build up on this idea of identifying patterns in architectures, allow users to specify actions to be performed when patterns are identified, and define an interface (through the type library) by which LEADS can be used in association with any specification environment.

The Pattern-Action approach

The Pattern-Action approach has been used in many useful tools. *yacc*[2] allows users to associate actions with production rules (conforming to LALR(1) grammars) that

are performed when the production rules are parsed. In XML[13], the XSL style sheets are specified as patterns and actions, and used to convert custom XML documents to HTML or any other form.

The design of LEADS has been influenced by the Z[14] software specification language. Though LEADS only offers a small subset of Z, it provides for some primitives (not in Z) that make the pattern specification process easier. Our concept of defining a pattern is very similar to schemas in Z. Schemas in Z contain some members and some invariants that hold amongst them. Schemas in Z can be instantiated, as a member of another schema.

8. Conclusions and Future Work

We present a Language for Exploiting Architectural and Design Specifications. Our contribution through LEADS is that we make tool writing faster, easier and modular. The phases of exploiting information are separated, and constructs are provided to make the information searching process easier. With the type library, we decouple from the specification environment and ensure the applicability of LEADS over a wide range of specifications and specification environments.

As we use the language to exploit the information content in specifications, more primitives that will ease the pattern searching process will be discovered. These primitives will be included in the language in future revisions.

Since, we generate C++ classes for the patterns, they can be extended easily to suit more specific requirements. A repository of such patterns can be maintained and the application of the exploitations can be integrated into the specification environments itself. This would eventually allow knowledge reuse.

9. References

[1] Rational Rose Extensibility Interface, Rational Rose Help, Rational Rose 98i from Rational Corporation, http://www.rational.com.

[2] S.C. Johnson, “YACC – Yet another compiler compiler”. Technical Report 32, Murray Hill, NJ: Computing Science Research Center, AT&T Bell Laboratories, 1975.

[3] J. Rumbaugh, I. Jacobson, G. Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1999.

[4] XML Metadata Interchange, http://www.omg.org

[5] Rational Software, http://www.rational.com

[6] R. Lakshminarayanan, “TriSL – A Software Architecture Description Language and Environment”, M.Sc.(Engg) dissertation, Department of Computer Science and Automation, Indian Institute of Science, May 1999

[7] ACME Web Site, http://www.cs.cmu.edu/~acme

[8] OMG Web Site, http://www.omg.org

[9] http://www.objectsbydesign.com/tools/umltools_byCompany.html

[10] M. Moriconi, R. A. Riemenschneider, “Introduction to SADL 1.0--A Language for Specifying Software Architecture Hierarchies”, Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, March 1997, http://www.csl.sri.com/sadl/sadl-intro.ps.gz.

[11] M. Moriconi, X. Qian, and R. A. Riemenschneider, “Correct Architecture Refinement”, IEEE Transactions on Software Engineering, vol. 21, no. 4, April 1995, pp. 356-372.

[12] R. Kazman, M. Burth, “Assessing Architectural Complexity”, 2nd Euromicro Working Conference on Software Maintenance And Reengineering (CSMR ’98), IEEE Computer Society Press, 1998.

[13] XML, http://www.w3c.org/XML

[14] J. Michael Spivey, “The Z Notation”, A Reference Manual, Second Edition, Prentice Hall, 1992