Carnegie Mellon University
School of Computer Science
Master of Software Engineering

Analysis of Software Artifacts
Final Project

Abstract Rule Language for the ArchE system

May 5, 2005

Team Phoenix
Jinhee Lee
Luis D. Maya
Chris Metcalf
Prasanth Ramanand
Soumya Simanta
Min Wang

TABLE OF CONTENTS

TABLE OF FIGURES

Glossary

Acronym or Term	**Description**
Abstract Rule	A high-level abstraction of a Jess rule. It is written in Abstract Rule Language. ArchE-II will generate the Jess rules from these abstract rules.
Abstract Rule Language	*A*bstract *R*ule *L*anguage defines the constructs used in defining architectural rules and facts in an abstract fashion.
Fact	A collection of architectural knowledge nuggets [Jess]
Jess	Java Expert System Shell. Jess is a rule engine and scripting environment for the Java Platform [Jess]
Knowledge provider	A user having architectural knowledge who manipulates abstract rules, RFs, and designs in the ArchE system in order to contribute expert knowledge.
Quality Attribute	A characteristic of a system that is essentially extra-functional in nature and is one of the key architectural drivers for any software system. Examples of quality attributes are *performance*, *availability*, *modifiability* and *usability*.
Quality Attribute Model	A realized instance of a reasoning framework. The QA model consists of the parameters linked to their sources.
Reasoning Framework	A body of knowledge that provides analytic means to reason about a specific quality attribute. It contains all the rules and facts in the ArchE-II system.
Rule	A rule represents a set of actions that should be performed in a specific situation.
Scenario	Expresses a specific extra-functional requirement of the system. Prioritized QA scenarios are the key architectural drivers for a software system. A scenario contains six parts (stimulus, source of stimulus, environment, artifact, response, response measure).

1 Introduction

1.1 Purpose of the document

This document describes the specification and implementation approach of Rule Language for the ArchE project.

1.2 ArchE project overview

ArchE (Architecture Expert Design Assistant) is a tool developed by the Software Engineering Institute (SEI) to help software architects create architectural models of their software systems. The ArchE tool is being developed for the Robert Bosch Corporation and the SEI.

ArchE is an expert system that takes software requirements from users and architectural knowledge from knowledge providers (architectural domain experts) and generates an architecture for the software system. Currently, the architectural knowledge in ArchE is codified as rules that are written in the Jess programming language.

1.3 Business drivers and scope

ArchE is an expert system. An expert system is a software system that makes decisions using an inference engine and a knowledge base containing domain-specific rules. The inference engine processes the rules and, based on information from its environment or specified by human input, produces a decision-based output, much like a human expert's reasoning process. Currently, the knowledge is written in the Jess programming language. The Jess language is a powerful way to express architectural rules, but it has a steep learning curve. This has become a bottleneck to the enhancement of ArchE capability because knowledge providers must be skilled in the Jess language before they can contribute their architectural knowledge in rule form. ArchE's capability and knowledge can be only increased if a wider community of software architecture experts creates rules to contribute their experience into ArchE's knowledge base. [PHOENIX1]

Here is a sample Jess rule from the performance reasoning framework. It initializes the priority to the tasks that have not been assigned a priority.

```
(defrule StartingPriority {
   ?t <- (RMAModel::Task (scenario ?sn))
   ?st<- (RMAModel:SubTask (task ?t)(priority ?n &: (eq ?n 0))
   =>
   (modify ?st (priority 1))
}
```

As the above example clearly shows, Jess is much less intuitive than popular programming languages such as Java or C++. It is difficult to read and it has a high learning curve. Most knowledge providers do not know how to program in Jess and do not have the time or desire to learn it. There are many issues with Jess as a language for knowledge providers. Here are the major ones listed by our clients:

- **Poor readability** - Jess uses prefix expression, which is hard to read and write, and is counterintuitive to programmers with experience in languages such as C or Java. For example, in order to assign value or compare two properties in Jess, the knowledge provider must use two variables with the same name instead of using the properties directly; this is different from conventional programming languages because most popular languages use infix notation.

- **Chaotic ordering of rules** - By default, Jess rules are executed nondeterministically and there is no way to impose an order upon their execution. To specify an *order* on the execution of rules, Jess uses two mechanisms: *focus* and *triggers*. Currently is the responsibility of the knowledge provider to explicitly write introduce triggers and focus into their Jess code to control the order of execution of various rules. This is introduces a significant amount of overhead when modifying the order of an existing set of rules. The user has to be very careful and must review each rule one by one.

- **Complicated question rules** - In ArchE, it is common to reach a point in the execution where more information is required from the user. ArchE can ask the user questions in order to obtain this missing information. In order to trigger this behavior in Jess, the knowledge provider must break up their logical rule into several smaller rules to ask the questions and validating their answers. This is complicated and inefficient.

- **Complicated persistence management** - Writing a rule that handles persistence (that is, saving the information to a file) is also a frequent but complicated task in the current Jess environment. Whenever the knowledge providers want to save a piece of information to a file, they must write a separate rule to do it.

In order to allow more knowledge providers to input their knowledge into ArchE without knowing Jess, our clients want us to create a new rule language. The new rule language should be easier to learn, should abstract away the programming overhead that Jess introduces, and should more intuitive to read and write. It should use the same vocabulary the architectural experts use, i.e., responsibilities, tactics, and quality attributes, among others.

In this paper, we will cover the first two problems. We will continue work in summer to solve the remaining ones.

2 The rule language

For simplicity we will call this new language the “Abstract Rule Language”, or ARL, from now on.

2.1 Example

Here is the example of a rule written in ARL. It does the same thing as the Jess rule in section 1.3.

```
rule startingPriority {
        description = “assign an initial priority for each task”;
        noPriorities = GETALL Task WHERE priority = none;
        foreach (Task task in noPriorities) {
                task.priority = 1;
        }
}
```

A programmer with basic knowledge in any object-oriented language like C++, Java, or C# will immediately understand the rule. Compared to the Jess, this rule is much easier to read and understand. It uses a syntax similar to that of Java and SQL. We will specify the syntax of the abstract rule language in this section.

An in-depth description of the language in Backus-Naur form is available in Appendix A.

2.2 Definitions and declarations

Types are defined very much like C structures.

```
1      type Person {
2         String fistName;
3         String lastName;
4         boolean famous;
5         List<Person> parentOf;
6      }
```

Types are strictly *invariant*, meaning that the use of polymorphism to create inherited or extended classes is not allowed. In technical terms this means covariance and contravariance are not permitted.

Line 6 shows how collections are declared within a type. Lists are typed: they can only have one type of element within them, and that type must be declared. In our example, the list would contain the sons and daughters of the Person. When using collections, all “unit“ types must be defined prior to declaring “List” types. The reasons behind this distinction will be described more below.

To define, assign, and reference the types we use Java-like notation. Here is an example:

```
7     Person architect = new Person(“Santiago”, “Calatrava”, true);
8     architect.firstName = “Frank Lloyd”;
9     architect.lastName = “Wright”;
10    architect.famous = true;
```

In line 7 we demonstrate a type constructor. Types must always be initialized when declared. Types have a default constructor that takes arguments in the order they were defined in the type. In our example, `“Santiago”` maps to `firstName`, `“Calatrava”` maps to `lastName` and so forth. The lists do not need to be initialized - this is why they must be declared last in a Type.

2.3 Rules

Rules are the most interesting part of our language. The basic elements of a rule are:

- A rule name
- A description of the rule
- Queries to select data of interest from the environment
- A conditional that evaluates a predicate and, if true, executes the actions
- And a set of actions, which, if executed by the conditional, can modify data in the environment or interact with the outside world.

Here is an example of a rule that prints the names of all of the people that own blue jackets.

```
1     ruleset example {
2        rule blueJackets {
3            description = “Name the owners of blue jackets”;
4            owners = GETALL jackets WHERE color = blue;
5
6            foreach (x in owners) {
7                execute println(x.owner.name);
8            }
9        }
10    }
```

Line 1 packages the rule in a named set of rules. Line 2 starts the definition of the rule by specifying its name. Line 4 queries the environment to retrieve all the owners of blue jackets. Line 6 loops on the elements that were retrieved in the query. Line 8 prints the name of the owner of the jacket. The keyword *execute* must be specified before calling any method. Note that if nobody owns a blue jacket nothing will be printed.

3 Mapping from the abstract rule language to Jess

Our abstract rule language is able to express the subset of the functionality of Jess necessary to efficiently express architectural reasoning. However, for the ARL rules to be used with ArchE, they must first be converted automatically into Jess rules.

The ARL works in a traditional functional language where statements are written following infix order. Jess is a LISP-like language written in pre-order. For example, in the abstract rule language the sum of *a* and *b* is expressed: `a+b`, in Jess the same is expressed `(+ a b)`. The pre-order notation allows for complex expressions to be written in a single statement.

The following tables show the translation from abstract rule language notation to Jess notation.

3.1 Type definition

Abstract rule language	Jess

type ResponsibilityNode {
int id;
String name;
double cost;
double cumulativeProb;
}
	(**deftemplate** Responsibility_Node
	(**slot** id (**type INTEGER**))
	(**slot** name (**type STRING**))
	(**slot** cost (**type DOUBLE**))
	(**slot** cumulativeprob(**type DOUBLE**))
	)

type Task {
String name;
List<Task> subtasks;
}
	(**deftemplate** Task
	(**slot** name (**type STRING**))
	(**multislot** subtasks)
	)

The abstract rule language specifies the following types: `int`, `double`, `String` and `Class`. The definition of a type for each member is required.

Jess supports similar types (`INTEGER`, `FLOAT`, `STRING`, and `OBJECT`) but does not require the user to specify the type, and also does not support type information within collections (like the `multislot` type). Collections are inherently typeless.

3.2 Rule definition

In Jess rules are dynamically organized in modules. When the engine is running the focus will be in one module at a time and only the rules in that module will be able to execute. The user is allowed to add rule to a module while the engine is running; the user can even create modules while the engine is running. This flexibility is powerful and allows for very unstructured programming.
On ArchE such power can be a difficulty because the rules are defined before execution time and we prefer to concentrate on checking the soundness of the rules. In the ARL we

will allow the user to organize the rules statically in sets. Rule sets have a description, an optional order of execution and rules. Here is an example:

Abstract rule language	Jess

ruleset aRulesetName {
}
	(**defmodule** ARL_ARULESETNAME)
	(**pop-focus**)

ruleset aRulesetName {
rule aRuleName {
   description = “a description”;
}
}
	(**defmodule** ARL_ARULESETNAME)
	(**defrule** aRuleName “a description”)
	(**pop-focus**)

The rule is the most important unit in Jess. A Jess rule is composed of a *condition* and an *action*. The action executes only if the condition is true or the declared variables are not null. Variables can have more than one value assigned to them. If a variable used on the *action* part of the rule has more than one value, the action will iterate for each one of them, this can be referred to as *implicit iteration*.

In the ARL we have made significant changes to the Jess rule concept. The user writes a set of queries about the information that the system has, and then he or she explicitly uses a condition or does iterations over it. This is simpler to understand, and the syntax is familiar to SQL in the query part, and to Java or C# on the condition and iteration part.

Here are some examples:

Abstract rule language	Jess
x = **GETALL** SomeType	(SomeType ?x)

x = **GETALL** SomeType **WHERE**
  property = someValue
	recursive insertion on the type:
	(SomeType ?x (property ?y & :(eq ?y someValue)))

x = **GETALL** SomeType **WHERE**
  propertyA = s1 **AND** propertyB = s2
	(SomeType ?x (propertyA ?w & propertyB ?v &
	   (and (eq ?w s1) (eq ?v s2)))

x = **GETALL** SomeType **WHERE**
  propertyA = s1 **AND**
 (propertyB = s2 **OR** propertyB = s3)
	(SomeType ?x (propertyA ?w & propertyB ?v &
	   :(and (eq ?w s1) (or (eq ?v s2) (eq ?v s3)))

x = **GETALL** SomeType **WHERE**
  propertyA = val1;
 	
foreach (SomeType sm **in** x) {
  println(sm);
}
	(SomeType ?x (propertyA ?w & propertyB ?v & :(eq ?w s1)))
	=>
	  (printout t ?x crlf)

x = **GETALL** SomeType **WHERE**
  propertyA = val1;
 	
IsNotEmpty(x) {
new Configuration.Priority(500);
}
	(SomeType ?x (propertyA ?w & propertyB ?v & :(eq ?w s1)))
	   (exists ?w)
	=>
	   (assert Configuration::Priority 500))

4 ArchE and the Abstract Rule Language

In order to generate an architecture, ArchE performs certain steps in strict order Each step carries out a group of actions that may vary according to the purpose at hand and to the expert's opinion of what should occur at a given point. For example, if the goal is to improve performance, the architecture will be transformed into a rate monotonic analysis model. It is a matter of expert opinion to say how each architectural element should be transformed.

ArchE includes the order of the steps, the purpose and the expert opinion. In theory, the expert is interested in introducing and testing his ideas into the system. Currently, rules are developed using Jess, which makes this task cumbersome and error-prone.

The ARL should simplify these tasks as much as possible. The ARL should provide a definition of the elements (i.e., responsibilities and properties), as well as the actions and its inputs and outputs.

The current order of steps and the inputs and outputs of each one of them can be found in Appendix B.

4.1 Defining the elements: the common namespace

ArchE works with different concepts that are common to all steps and reasoning frameworks:

- *Responsibilities:* Functional elements that express the application requirements.
- *Scenarios:* Statements that express the actual meaning of a quality attributes in the context of the application.
- *Architecture elements:* The high level components of the system architecture.
- *Quality attribute model:* The way that architecture elements are interpreted to understood to link them to the scenarios.

These concepts will be defined as types within a global namespace. They will then be available for all rule designers. The concepts that are particular to a reasoning framework will be defined within its own namespace.

Below is the definition of the global namespace:

```
type Basic {
       int id;
       String name;
       String description;
}

type Property {
       String name;
       double value;
}
```

```
type Relation extends Basic {
       Object from;
       Object to;
       String toRole;
       String fromRole;
}

type Responsibility extends Basic {
       List<ArchitecturalElement> architecturalElements;
       List<Relation> relations;
       List<Property> properties;
}

type ArchitecturalElement extends Basic {
       List<Properties> properties;
       List<Responsibility> matchingResponsibilities;
       List<Relation> relations;
}

type ConcreteScenario extends Basic {
       String stimulus;
       String stimulusSource;
       String environment;
       Responsibility responsibility;
       String response;
       String responseMeasure;
}

type GeneralScenario {
       List<Property> filter; // i.e. <”stimulus”, “Human”>
}

type ReasoningFramework {
       ruleset ArchEFormulations;
       ruleset questions;
       ruleset interpretation;
       ruleset solvingDecision;
       ruleset architectureSelection;
       QAModel model;
       List<GeneralScenario> gralScenarios;
       Measure responseMeasure;
}

type Measure extends Property{}

type QAModel {
       List<QAElements> allowedElements;
       Map<String, Operation> operations;
}

type QAElement extends Basic {}

} //namespace
```

5 ARL in practice: Performance reasoning framework

5.1 Types

```
type BasicTask extends ModelElement {
       double latency;
       double deadline;
       int priority;
       double period;
       Responsibility responsibility;
}
type Task extends BasicTask {
       List<Subtask> subtasks;
}
type SubTask extends BasicTask {
}
```

5.2 Actions

```
action Complain(String) : void sei.cmu.UI.Question.Complain(String);
action RmaSolver() : sei.cmu.rma.Solver.Execute() int;
action  relaxExecutionTimes()  :  sei.cmu.rma.Tactics.RelaxExecutionTimes()
void;
action relaxArrivalTimes() : sei.cmu.rma.Tactics.RelaxArrivalTimes void ;
action changeDeadlines () : sei.cmu.rma.Tactics.ChangeDeadlines void ;
```

5.3 Rules

We were given a list of rules expressed in pseudo-code. In the remaining sections we will write their equivalent in ARL.

5.3.1 Checking scenarios

Is scenario well-formed
if response-type is hard-deadline and stimulus-type is-not periodic or sporadic then complain

```
ruleset ScenarioChecking {
       rule wellFormedScenario {
              description= “is scenario well-formed”;

              scenario = GETALL ConcreteScenario
                     WHERE response_type = deadlines.hardDeadline AND
                     NOT (stimulusType = StimulusType.periodic OR
                      stimulusType = StimulusType.sporadic );
```

```
             foreach (Scenario a in scenario) {
                    execute Complain(“Scenario in element ” + a.name +
                    “ is not well formed”);
             }
       }
}
```

5.3.2 Building the QA model from the scenarios

For each hard-deadline scenario create a task

if response-type is hard-deadline and scenario is-not-allocated-to a task then
create task-i and associate stimulus-type, with task-i and associate stimulus-value with task-i and associate deadline with task-i and create subtask-i1 and allocate subtask-i1 to task-i and associate 0 execution-time with subtask-i1 and create stimulus-responsibility and allocate stimulus-responsibility to subtask-i1

```
ruleset BuildingModel {
  rule hardDeadlineForEachScenario {
     description = “for each hard-deadline scenario create a task”;

     scenarios = GETALL ConcreteScenario WHERE
     response_type == deadlines.hardDeadline AND
     scenario == none;

     foreach (sc in scenarios){
       Task task = new Task;
       task.desc = “task to meet “ + sc.name + “ hard deadline”;
       task.stimulusType = sc.stimulusType;
       task.deadline = sc.deadline;
       Task subtask1 = new Task;
       task.subtasks.add(subTask1);
       subtaks1.desc = “subtask to meet “ + sc.name + “ hard deadline”;
       subtask1.executionTime = 0;
       Responsibility resp = new Responsibility;
       resp.desc = “Stimulus for “ + sc.name + “ hard deadline”;
       resp.name = sc.name + “Stimulus”;
       subtask1.responsibility = resp;
     }
  } // rule
```

Assign a starting priority for each task

if task-i does not have a priority then associate middle-priority with task-i

```
  rule startingPriority {
       description = “assign a starting priority for each task”;
       noPriorities = GETALL task WHERE priority = none;
       foreach (task in noPriorities) {
          task.priority = Priority.middle;
       }
  }
```

Adjust priorities so that they are rate monotonic

if task-i’s deadline <= task-j’s deadline and task-i’s priority < task-j’s priority then switch priorities

```
rule adjustPrioritiesToBeMonotonic {
    description = “adjust priorities to rate monotonic”;

    Tasks = GETALL (Task JOIN Task);

    foreach (Task task1, task2 in Tasks) {

       if (task1.deadline <= task2.deadline &&
           task1.priority < task2.priority) {
           int temporal;
           temporal = task1.priority;
           task1.priority = task2.priority;
           task2.priority = temporal;
       }

    }
}//rule
```

Allow for the future possibility of subtasks having deadlines

if sub-task-ij does not have a deadline then associate priority-of-task-i with sub-task-ij

```
rule allowDeadlinesForSubtasks {
    description = “Allow subtasks to have deadlines”;

    tasks = GETALL Task;

    foreach (Task task in tasks) {

       foreach (Task subTask in task.subtasks){
           subtask.priority = int temporal;
           temporal = task1.priority;
           task1.priority = task2.priority;
           task2.priority = temporal;
       }

    }
}//rule
```

Subtasks currently do not have deadlines

if sub-task-ij has a deadline then
issue-message: “unimplemented feature”

```
rule complainAboutSubtasksWithDeadlines {
    description = “Subtasks currently do not have deadlines”;

    tasks = GETALL Task;

    foreach (Task task in tasks) {

       foreach (Task subTask in task.subtasks) {
           if (subtask.deadline != 0) {
              subtask.deadline = 0;
              execute Complain(“Unimplemented feature, subtasks
                  should not have deadlines. Changing to NO deadline”;
           }
       }

    }
}//rule
```

Computes execution time for tasks
if task-i does not have an execution time and
all of its subtasks do have execution times then
associate sum the execution times of its subtasks as its execution time

```
   rule computeExecutionTime {
       description = “Computes execution time for tasks”;

       tasks = GETALL Task as task WHERE executionTime <> 0
              AND subtasks.size > 0;
              AND #(GETALL Task as subt WHERE task.subtask.contains(subt))
                ==#(GETALL Task as subt WHERE task.subtask.contains(subt)
                    AND executionTime <> 0);

       foreach (Task task in tasks) {
          double sum = 0;
          foreach(Task subTask in task.subtasks) {
              sum = sum + subtask.executionTime;
          }
          task.executionTime = sum;
       }
   }//rule
```

Compute responsibility execution time
if responsibility-k is associated with a subtask and does not have an execution time then
associate compute-exec-time (responsibility-k) with subtask-i

```
   rule computeResponsibilityExecutionTime {
       description = “Computes responsibility execution time”;

       tasks = GETALL Task WHERE deadline != 0;

       foreach (Task task in tasks) {
          double sum = 0;
          foreach(Task subTask in task.subtasks) {
              sum = sum + subtask.deadline;
              }
          }
       }
   }//rule
```

5.3.3 Solving model

Execute RMA solver
if all tasks have execution time times then **execute** RMA-solver and associate each

latency with its respective task

```
ruleset SolvingModel {

  rule associateLatencies {
    description = "Associates latencies if all tasks have execution
times";

    /* Get all tasks with execution times */
    tasks = GETALL Task WHERE executionTime > 0;

    /* Only continue if its the same as the set of all tasks */
    equals(tasks, Task) {
      foreach(task in tasks) {
        execute RMASolver(task);
      }
    }
  }
} // ruleset
```

5.3.4 Applying tactics to model

Deadlines are violated; check priority assignment

if task latency > task deadline and rate-monotonic-violated then complain

```
ruleset ApplyingTacticsToModel {
  rule complainDeadlinesViolated {
    description = "Deadlines are violated, check priority assignment";

    /* Perform for all our tasks */
    foreach(task in Task) {
      if(task.latency > task.deadline && RateMonotonic.violated) {
        Complain("Deadlines are violated in " + task.name);
      }
    }
  }
```

Deadlines are violated; ask what’s negotiable

if task latency > task deadline then query if relaxing execution times is OK and
query if relaxing arrival rates is OK and query if changing deadlines is OK

```
  rule askNegotiable {
    description = "Deadlines are violated, ask what is negotiable";

    /* For all the tasks*
    foreach(task in Task) {
      if(task.latency > task.deadline) {
        Result result = execute Query("Is it OK to relax”+
               “the execution times?", yes, no);
        task.relaxExecutionTime = result;

        result = Query("Is it OK to relax the arrival rates?", yes, no);
        task.relaxArrivalRates = result;
        result = Query("Is it OK to change the deadline?", yes, no);
        task.changeDeadline = result;
      }
    }
  }
```

Apply reduce arrival rate

if relaxing arrival rate is OK for task-i then display delta-period necessary to meet deadline

```
  rule applyReduceArrivalRate {
    description = "Reduce arrival rate if OK";

    /* For all the tasks */
    foreach(task in Task) {
      if(task.relaxArrivalRates == true) {
        Display("DeltaPeriod = " + task.deltaPeriod);
      }
    }
  }
```

Apply change deadline

if changing deadline is OK for task-i then set deadline to latency and propagate back to scenario

```
  rule applyChangeDeadline {
    description = "If changing deadline is ok, ” +
        “then set deadline to latency and propagate back to scenario";
    foreach(task in Task) {
      if(task.changeDeadline == true) {
        task.deadline = task.latency;
        task.scenario.deadline = task.latency;
      }
    }
  }
} // ruleset
```

5.3.5 Subsetting requirements

Determine which deadlines are missable

if not (relaxing execution times is OK or relaxing arrival rates is OK or changing deadlines is OK) then assign importance to each scenario

```
ruleset SubsettingRequirements {
   rule determineMissableDeadlines {
        description = "Assign importance to each scenario";
        // assuming these are states
        if (relaxExecutionTime | relaxArrivalRates) {
            foreach(scenario in Scenario) {
            execute assignImportance(scenario);
        }
    }
  }
} // ruleset
```

6 Improvements over Jess

6.1 Readability

Compared to Jess, the rules written in the ARL are much more intuitive. Because the ARL uses a syntax similar to that of Java and SQL, it is likely that architecture experts will be able to understand the rule better than if it were written in Jess. This reduces the learning curve of the ARL.

For example, when writing Jess rules, to the rule designer cannot compare two properties directly, so they must use references with same name to compare two properties. In ARL rules, they are allowed to use a logical operator to compare two properties directly.

Also, expressions in Jess rules are written prefix notation, which is difficult for many people to read. The ARL uses more familiar infix expressions.

Here is a sample rule that initializes priority for each task. Notice that the ARL rule is much more clear and readable.

ARL:

```
class Task
{
  Property String name;
  Property Integer priority;
  …;
}

rule startingPriority {
    description = “assign an initial priority for each task”;
    noPriorities = GETALL Task WHERE priority = none;
    foreach (task in noPriorities) {
           task.priority = 1;
    }
}
```

Jess:

```
(deftemplate Task
(slot name) ;
(slot priority) ;
…;
)

(defrule StartingPriority {
?t <- (RMAModel::Task (scenario ?sn))
?st<- (RMAModel:ubTask (task ?t)(priority ?n &: (eq ?n 0))
=>
    (modify ?st  (priority 1)
}
```

6.2 Strongly typed

ARL is a strongly typed language. Each property has an explicitly defined type, which allows us to perform type checking for assignment and conditional expressions to prevent illegal expressions.

In contrast, Jess is weakly typed, and properties types in Jess rules are not declared initially. In Jess, a programmer may assign or compare any type of value to any other type of properties without violating type constrictions.

Note that in the example above, the members of the ARL type `Task` were explicitly typed, while in Jess they are not typed and instead are loosely declared as `slot`.

6.3 Ordered rule execution

The order of execution of rules is non-deterministic. In Jess, the programmer uses triggers to control the order in which rules execute. Inside a Jess rule, a trigger can be specified at the beginning, which means the rule will only be executed when the trigger is present. At the end of a Jess rule, the programmer can create another trigger or remove existing triggers. These triggers are scattered throughout the rulesets, so it is very hard to predict and manage the ordering of all the rules.

In ARL, all of the triggers are defined and managed (as named events) in a central global space. It includes three maps: a map from each event to all rules invoked by this event, a map from each event to the rules that create this event, and a map from each event to all rules removed by this event.

Here is an example:

ARL:

```
Triggers
{
  RunsWhen {
  trigger1 rulel;
  trigger2 rule2, rule3;
  }
  CreatedBy{
  trigger1 rule0;
  trigger2 rule1;
  }
  RemovedBy{
  trigger1 rule1;
  trigger2 rule2;
  }
}

rule rule0{
…
}
rule rule1 {
…
}

rule rule2 {
…
}

rule rule3 {
…
}
```

And its counterpart in Jess:

Jess:

```
(defrule rule0 {
…
(assert (Trigger trigger1))
}

(defrule rule1 {
?e <- (Trigger trigger1)
…
(assert (Trigger trigger2))
(retract ?e)
}

(defrule rule2 {
?e <- (Trigger trigger2)
…
(retract ?e)

}
(defrule rule3 {
(Trigger trigger2)
}
```

Since in ARL all the events are defined and managed in “Event” block, the programmer is better able to predict and modify the ordering of all the rules as compared to Jess.

7 Implementation of the ARL Parser

7.1 Overview

The ARL parser will be created using the Java “Compiler Compiler” package (JavaCC). JavaCC will allow us to quickly build a parser and abstract syntax tree (AST) generator from a Backus-Naur Form (BNF) representation of the structure of our language.

The abstract syntax tree can then be traversed using a specialized abstract syntax tree Visitor [GoF]. As the visitor traverses the abstract syntax tree, it will encounter places where it must perform type checking. For example, if it encounters an assignment, it can check whether the bound fact being assigned is of the same type as the value it is being assigned.

7.2 Creating the ARL Parser

To generate the ARL parser, first we must describe our abstract rule language in BNF form and create a JJTree “.jjt” file. The .jjt file contains the BNF representation of the rule along with special code to define the abstract syntax tree for our rule language.

That jjt file is then fed into the JJTree parser. JJTree uses the special AST generation code in the jjt file to create a JavaCC “.jj” file which contains the code to create the parser that will generate the abstract syntax tree.

That jj file is then parsed by JavaCC to create Java source code that, when compiled, will yield a parser to tokenize an ARL file and a tree generator that will generate an AST data structure defining the internal structure of its code. [JJT1]

The Workflow to Create the ARL Parser

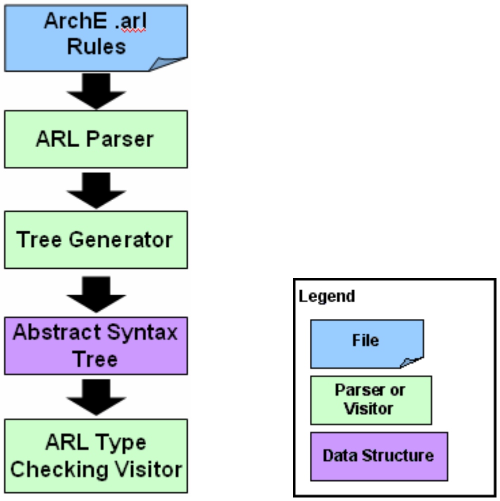

Figure 1: Creating the ARL Parser

Steps:

1. The .jjt file is parsed by JJTree
2. JJTree creates .jj source code for JavaCC containing the tree generation code.
3. The .jj file is parsed by JavaCC.
4. JavaCC creates the .java source which can be compiled to create the ARL parser.

7.3 Using the ARL Parser

The ARL parser and tree generator source code created by JavaCC is then compiled into a Java parser that can then be used to parse ARL source code files. It will also yield a tree generator that will create an abstract syntax tree using input from the parser. The ARL files are read in by the parser and then passed to the tree generator. The tree generator creates an abstract syntax tree for that file. An abstract syntax tree is a special data structure that expresses the logical structure and meaning of the source code. A "Visitor" class can then be created that will traverse the nodes of the abstract syntax tree and check properties. For example, we can perform type checking by verifying that the parameters of operations that are occurring at each node of the tree are of compatible types.

Performing Type Checking on an ARL File

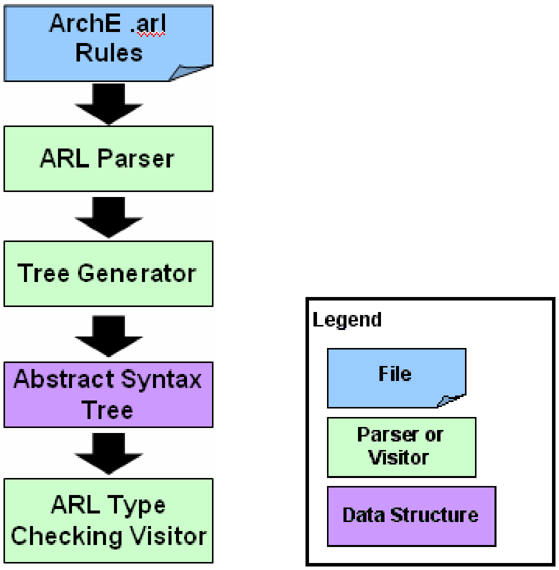

Figure 2: Type checking on the ARL

Steps:

1. The ARL Parser reads in the .arl file.
2. The parser then generates the abstract syntax tree.
3. The ARL Type-Checking Visitor then traverses the abstract syntax tree and performs type checking.

8 Conclusion –the progress so far

We are happy of the results so far and our clients have expressed their satisfaction with our work.

In conclusion, we have created a new language that has addressed the main short-comings of Jess in the ArchE system. The new language is:

- More readable and hence easier to use for users who are already familiar with high-level language such as Java and C;
- Easier to verify and maintain;
- More abstract than Jess as it abstracts away a lot of infrastructure complexity (question rules)
- Provides more explicit control to the user
- More structured than Jess.

We have discussed the syntax and semantics of our new rule language with our clients. They have asserted that our language fits their needs and are excited about its future. We discussed this using mapping between existing Jess rules and the same rules written in our abstract rule language.

Finally, we have done experiments and written prototypes using JavaCC and JJTree. These experiments and prototypes have helped us understand the overall workflow of creating a parser and abstract syntax tree from a standard grammar (BNF) representation. This has given us confidence and knowledge that will help implement the whole rule language in the summer semester.

9 Future work

The abstract rule language is not finished, there are still several tasks remaining that we will work on during the summer.

1) **Make the language more complete and abstract** - Test that our language actually covers all the Jess rules in the existing set of rule in ArchE i.e. the *modifiability* and the *performance* reasoning frameworks.

2) **Type checking implementation** - The parser and type checking should be implemented based on the approaches in section 7. It is not necessary to wait until the BNF is completely done. We can start to implement the parser with a subset of the BNF and keep refining it continuously because the architecture of the parser supports such extensibility.

Jess transformer implementation - Jess transformer is a code generator to translate abstract rules to Jess rules. Currently our mapping between the abstract rule language and Jess is ad hoc. We want to represent this transformation between the abstract rules and the Jess rules using *formal rewriting rules*. Once we have formally defined the rewriting rules from the abstract domain to the Jess domain the implementation can be done easily.

10 Appendix A: BNF for the Abstract Rule Language

10.1 Notation

Symbol	Meaning
*	zero or more
?	zero or one
+	one or more
|	Or
()	Group
words in bold and italics	language symbol
//	comments

```
ARL ::= def*
def ::= ruleset | namespace
```

10.2 Collections of rules

```
ruleset ::= ruleset name { trigger_definition? description rule* }
description ::= description = “ ANSII_character* ”

rule ::= rule name { description query* conditional* }

// Defining rule’s left hand side (the SQL-like notation)
query ::= variable = GETALL type_name (WHERE query_condition)? ;
type_name ::= name
query_condition ::= boolean_term
boolean_term ::= (boolean_term boolean_op)? (NOT)? predicate
boolean_op ::= AND|OR
predicate ::= (qual_name comparison_op qual_name)
   | (EXISTS qual_name)
   | (UNIQUE qual_name)
comparison_predicate ::=
comparison_op ::= <|>|≤|≥|=

// Defining rule’s right hand side (the consequence)
conditional ::= conditional_type { action* }
conditional_type ::= (foreach left_parens name in variable right_parens)
       | (if left_parens boolean_conditional right_parens)
boolean_conditional ::= (isEmpty left_parens variable right_parens)
       | (variable (==|!=) variable)
```

10.3 Actions

```
action ::= conditional | function_call | definition | assignment | delete
```

10.3.1 Function calls

```
function_call ::= execute method_name parameters ;
method_name ::= name;
parameters ::= left_parens (assigned_value comma_sep_val*)? right_parens)
comma_sep_val ::= , assigned_value  // add the comma before each parameter
assigned_value ::= ((value math_operation)? value)
       | (new type_name parameters)
value ::= variable | constant_value
math_operation ::= + | - | * | /
constant_value ::= numeric_character* | name
```

10.3.2 Assignments

```
assignment ::= variable = assigned_value;
```

10.3.3 Definitions

```
definition ::= (type_name|basic_type) name = value ;
basic type ::= int | String | double | boolean
```

10.3.4 Delete

```
delete ::= delete name
```

10.4 Triggers

```
trigger_definition ::= Events { t_invoke? t_create? t_remove? }
trigger_invoke ::= RunsWhen { declarations }
trigger_create ::= CreatedBy { declarations }
trigger_remove ::= RemovedBy { declarations }
declarations::= trigger_name: rule_name? (, rule_name)* ;
trigger_name ::= name
```

10.5 Namespaces

```
namespace ::= namespace name { definitions* }
definitions ::= enum_definition | type_definition

// this is like the C++ enums or the Java 1.5 enums
enum_definition ::= enum name { name? (, name)* }

type_definition ::= type type_name (extends type_name)?
                    { description declaration* }
declaration ::= any_type name (= constant_value)? ;


// name and string definitions
qual_name ::= name (. name)*  // qualified name, DOT notation
variable ::= name
```

```
name ::= alphabetic_character alphanumeric_character*
```

10.6 Character definitions

```
ANSII_character = (alphanumeric_character |\t|...) // includes spaces
alphabetic_character = (a|b|c|...|x|y|z|A|B|C|...|X|Y|Z)
numeric_character = (0|1|2|3|4|5|6|7|8|9)
alphanumeric_character = (alphabetic_character| numeric_character)
```

11 Appendix B: ArchE's current workflow

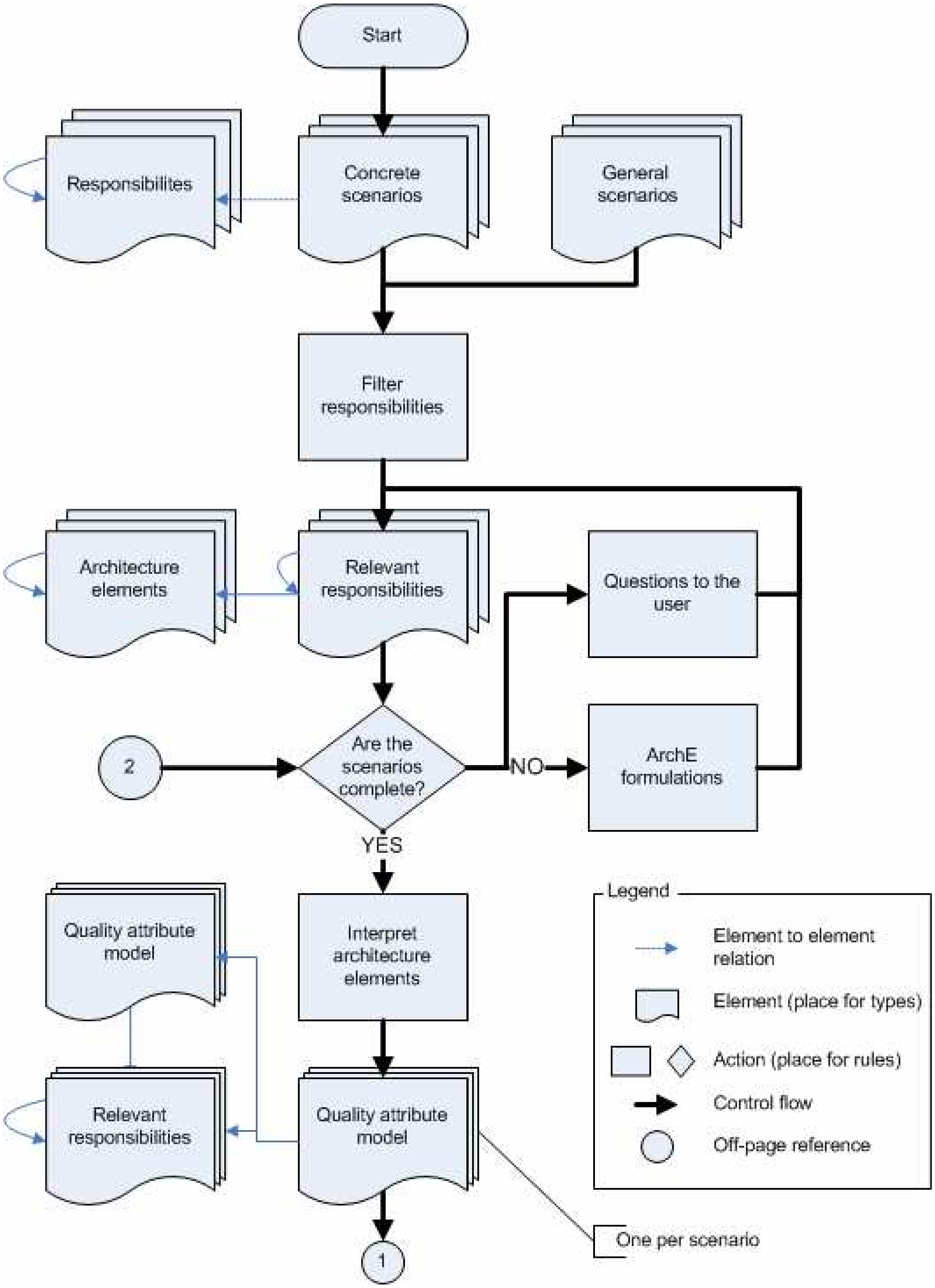

Figure 3: Architectural design process (part 1)

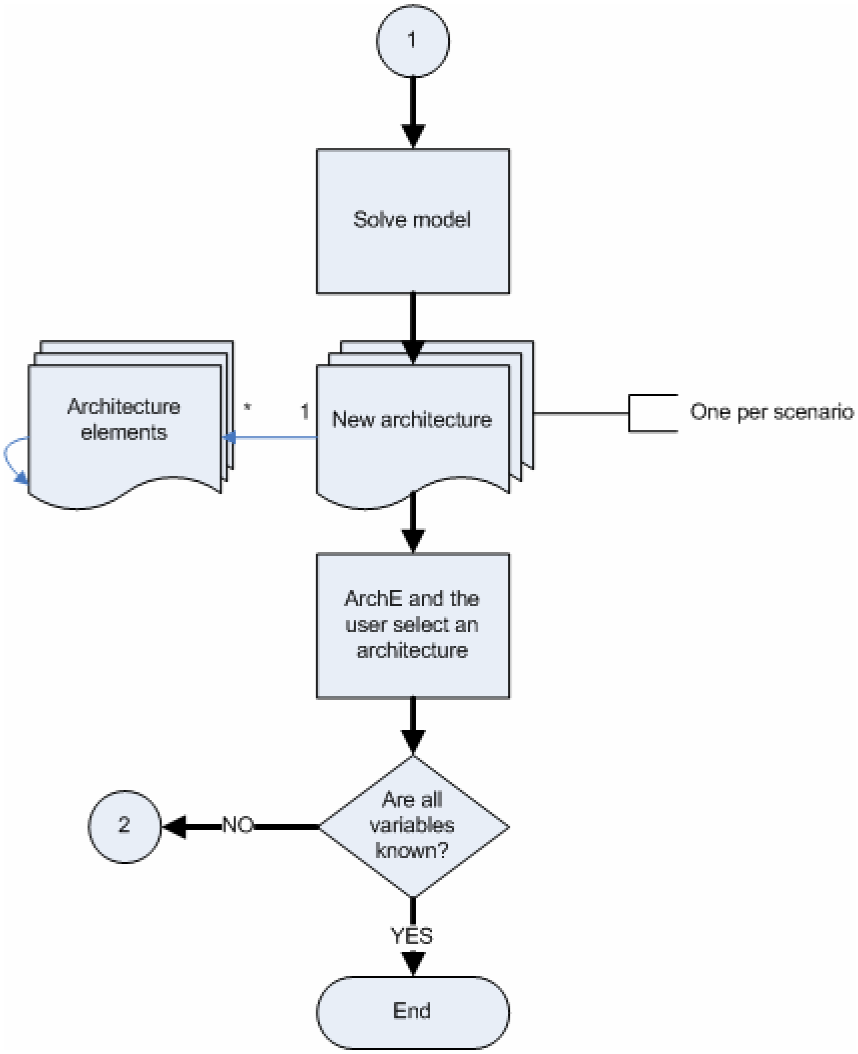

Figure 4: Architectural design process (part 2)

12 References

[GoF]	E. Gamma et. al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[Jess]	Ernest Friedman-Hill, Jess in Action (Rule Based system in Java.), Manning Publishers
[JJT1]	JavaCC [tm]: JJTree Reference Documentation, JavaCC Project, 2004 (https://javacc.dev.java.net/doc/JJTree.html)
[PHOENIX1]	Software Requirement Specification for ArchE-II system, March, 2005, Team Phoenix
[PHOENIX2]	Statement of Work, December, 2004, Team Phoenix