Grasp Language Reference Manual

Version 1.0

(March 2011)

Dharini Balasubramaniam

Lakshitha de Silva

School of Computer Science

University of St Andrews

St Andrews

Fife KY16 9SX

UK

Table of Contents

1. Introduction

Grasp is a textual, general purpose Architecture Description Language (ADL) designed to express the software architectures of systems. The software architecture [1, 2] of a system defines its structure and behaviour in terms of its components and their interactions and provides a high level model that forms a useful basis for development.

The rationale for the choice of components and their interactions is an important facet of the architecture, which can be vital in guiding development and maintenance but is often overlooked in its specification. The design of Grasp was motivated by the definition of software architecture by Perry and Wolf as {elements, form, rationale} [1] and supports all 3 aspects of an architecture specification.

2. Architecture Model

Grasp has been designed to provide

- language primitives for core architectural concepts,
- composition mechanisms to allow programmers to create composite elements and architectures based on primitives, and
- generic constructs that allow customisation and reuse as required.

The conceptual architecture model of Grasp is illustrated in Fig 1 below.

3. Syntax Definition

The Grasp grammar is defined in the EBNF notation. For example, the following production for a component definition

```
component_statement : annotation* 'component'
                      name=IDENTIFIER '=' base=IDENTIFIER
                      '(' argument_list? ')' because_opt? ';'
```

allows

- zero or more annotations,
- an optional argument list,
- a mandatory name and template identifier, and
- an optional rationale

to be associated with it.

4. Types

Grasp supports the base types *integer*, *real*, *boolean* and *string* to facilitate the specification of architectural properties, rationale and check clauses. Apart from the architectural constructs, a *set* type constructor is also provided.

Expressions and Operators

The usual expressions and operators over the base and set types are supported in Grasp. For example,

```
[x] subsetof ["input", "output", "error"]
```

is a valid Boolean expression over sets of strings.

Type Checking

The Grasp compiler checks type safety where applicable. Name equivalence is used to determine type compatibility.

Scope Rules

Grasp allows both global and local declarations. Values defined within a layer or component are only available within that scope.

5. Layers

Layers are supported as explicit architectural constructs in Grasp. As layered architectures are frequently used in modern applications, this design decision eases the task of the architect by providing a primitive that will enforce the interaction integrity of layers without requiring the specification of additional constraints.

For example,

```
layer presentation over business_logic
{
      // body of presentation layer
};
layer business_logic over data_access
{
      // body of business_logic layer
};
```

the above Grasp fragment specifies that the *presentation* layer may communicate with the *business_logic* layer but it cannot directly interact with the *data_access* layer.

A layer can contain a number of other elements and their configuration and hence can be considered a composite element of an architecture.

6. Components

Components are the main units of functionality in Grasp and are typically specified as instantiations of templates to support reuse. Components can provide and require interfaces as well as take parameters. However they do not return a value.

For example,

```
component consumer = ConsumerTemplate(10);
```

the above code fragment shows a component being defined as an instantiation of a template, which takes an integer parameter.

7. Connectors

Connectors in Grasp are used to separate the concerns of interaction from core functionality. They can facilitate and provide abstractions for communication between different components.

While Grasp supports the concept of connectors, it does not enforce them. Thus, it is possible to define an architecture in Grasp where two or more components interact directly with one another without their communication being mediated by connectors.

8. Templates

Templates in Grasp are abstractions over components and connectors, which can be instantiated to produce the required elements. Templates can have all the features of components and connectors. In addition, they can also take parameters.

The code fragment below shows the definition of a template to produce consumer components:

```
template ConsumerTemplate(id) {
      requires IProducer;
}
```

Consumers produced by this template will require to be connected to an element which provides *IProducer*.

9. Properties

Properties are statements that can be made about the characteristics of architectural elements.

For example,

```
template ProducerTemplate() {
      provides IProducer;
      property maxConnections = 25;
}
```

the above Grasp code for a template can be used to specify that each producer may be connected to a maximum of 25 consumers.

Grasp also provides *check clauses* through which an element may verify the compatibility of other associated elements (parameters or components that provide required interfaces to that element).

For example, a connector may want to check that the any producer it is wired to can cope with at least 15 consumers. It can do so as follows:

```
template ConnTemplate() {
      requires IProducer p;
      provides IProducer;
      check p->maxConnections >= 15;
}
```

Grasp also allows check clauses over the compatibility of values to be accepted as parameters.

10. Linking

The architecture of a system is defined as a configuration of a number of elements, which may interact with one another. In Grasp, the interaction is specified using *link* statements. As a convention, a link is made from an element providing an interface to an element that requires the interface.

For example,

```
component consumer = ConsumerTemplate(10);
component producer = ProducerTemplate();
connector cp_conn = ConnTemplate();
link producer.IProducer to cp_conn.IProducer
link cp_conn.IProducer to consumer.IProducer
```

the above code shows the architecture of a very simple system containing a *producer* and a *consumer* interacting via a *cp_conn* connector.

11. Rationale

An important motivation for the design of Grasp is to capture architecture rationale and associate it with the relevant architectural elements.

Rationale is defined in Grasp as an and-composition of one or more reasons. For example,

```
rationale Conn_Rationale {
      reason #'Separate functionality from interaction';
      reason #'Have to cope with at least 15 consumers';
}
template ConnTemplate() because Conn_Rationale{
      requires IProducer p;
      provides IProducer;
      check p->maxConnections >= 15;
}
```

the above code fragment shows the reasons for the connector and the checks the connector has to do.

12. Annotations

Annotations are supported in Grasp to provide information for tools that work over architecture specifications. The annotation syntax is somewhat similar to that of Java. Annotations can be associated with any architectural element.

For example,

```
@Generator(location = "/usr/source/")
template ProducerTemplate() {
      provides IProducer;
      property maxConnections = 25;
}
```

the above code fragment can be used to inform the generator tool that any code generated for *ProducerTemplate* should be placed in the directory */usr/source*. Labels for the annotations are defined by the user.

13. System

An architecture specification in Grasp is typically structured as a collection of reusable rationale and template definitions followed by the instantiations of these definitions and their configuration into an element graph, which forms the architecture. The *system* construct identifies the latter part.

14. Architecture Styles

TBD

15. Standard Functions

Grasp provides a number of pre-defined functions to access and interrogate features of an architecture specification. These functions are particularly useful in constructing check clauses that represent dependencies and consistency or compatibility requirements of architectural elements.

For example,

```
template ConnTemplate() {
      requires IProducer p;
      provides IProducer;
      check p->properties() subsetof [(maxConnections, 25)];
}
```

the above code uses the *properties* standard function, which returns the set of properties of a component or connector as name - value pairs.

16. An Example

The example below is a Grasp specification of the architecture of a wireless sensor network simulator and is used to illustrate the concepts outlined above.

```
//
// A simple Grasp architecture specification of a WSN simulator
//
architecture WSNSimulator
{
      // Rationale descriptors
      rationale R1() {
            reason #'Use layered architecture style';
            reason #'Achieve clear separation of concerns';
      }
      rationale R2() {
            reason #'Use MVC design pattern';
      }
      rationale R3() {
            reason #'Simulator engine must perform event logging';
      }
```

```
// Templates
template ViewComponent() because R2 {
      provides IView;
      requires IModel;
}
template ControllerComponent() because R2 {
      requires IView;
      requires IModel;
}
template ModelComponent() because R2 {
      provides IModel;
      requires ISequencer;
      requires INetwork;
}
template SensorNetworkComponent() {
      provides INetwork;
      requires ISequencer;
      requires ILogger;
}
template SequencerComponent() {
      provides ISequencer;
      requires ILogger;
}
template LoggerComponent() {
      provides ILogger;
      check isConnected();
}

// Static model
@Visualiser(Canvas = [1000,1000])
system StaticModel
{
      layer PresentationLayer over
            SimulatorLayer, UtilityLayer because R1
      {
            component texts = ViewComponent();
            component graphics = ViewComponent();
            component controller = ControllerComponent();
            component model = ModelComponent();

            link texts.IModel to model.IModel;
            link graphics.IModel to model.IModel;
            link controller.IView to texts.IView;
            link controller.IView to graphics.IView;
            link controller.IModel to model.IModel;
      }
```

```
            layer SimulatorLayer over UtilityLayer because R1
            {
                  component network = SensorNetworkComponent();
                  component sequencer = SequencerComponent();
                  link network.ISequencer to sequencer.ISequencer;
            }

            layer UtilityLayer because R1
            {
                  component logger = LoggerComponent();
            }

            link PresentationLayer.model.INetwork to
                  SimulatorLayaer.network.INetwork;
            link PresentationLayer.model.ISequencer to
                  SimulatorLayaer.sequencer.ISequencer;
            link SimulatorLayaer.network.ILogger to
                  UtilityLayer.logger.ILogger because R3;
            link SimulatorLayaer.sequencer.ILogger to
                  UtilityLayer.logger.ILogger because R3;
      }
}
```

17. Implementation Status

A Grasp parser, developed using the ANTLR parser generator framework [3], is currently available.

Work is ongoing on a number of tools to facilitate the use of Grasp, including those for checking properties, visualising architectures, graphical design, code generation and an IDE as an Eclipse plug-in.

References

[1] Dewayne E. Perry and Alexander L. Wolf
Foundations for the Study of Software Architecture
ACM SIGSOFT Software Engineering Notes, Vol 17-4, pp 40 – 52, 1992.

[2] Mary Shaw and David Garlan
Software Architecture: Perspective of an Emerging Discipline
Prentice Hall, 1996.

[3] Terence Parr
ANTLR Parser Generator
http://www.antlr.org/, 2011.

Appendix A: Grasp Grammar

```
// Entry point
start      : architecture_statement

// <architecture> statement
architecture_statement
        : annotation* 'architecture' name=IDENTIFIER '{' architecture_item* system_statement
        architecture_item* '}'
architecture_item
        : requirement_statement
        | quality_attribute_statement
        | template_statement
        | rationale_statement

// <requirement> statement
requirement_statement
        : annotation* 'requirement' name=IDENTIFIER ';'

// <quality_attribute> statement
quality_attribute_statement
        : annotation* 'quality_attribute' name=IDENTIFIER supports_opt?
        ('{' property_statement* '}' | ';')

// <rationale> statement
rationale_statement
        : annotation* 'rationale' name=IDENTIFIER '(' parameter_list? ')' extends_opt?
        '{' reason_statement* '}'

// <reason> statement
reason_statement
        : annotation* 'reason' (expression | supports_opt) inhibits_opt?  ';'

// <template> statement
template_statement
        : annotation* 'template' name=IDENTIFIER '(' parameter_list? ')' extends_opt? because_opt?
        '{' statement* '}'
```

```
// <system> statement
system_statement
        : annotation* 'system' name=IDENTIFIER because_opt? '{' statement* '}'

// <layer> statement
layer_statement
        : annotation* 'layer' name=IDENTIFIER layer_over? because_opt? '{' statement* '}'
layer_over
        : 'over' layers+=IDENTIFIER (',' layers+=IDENTIFIER)*

// <component> statement
component_statement
        : annotation* 'component' name=IDENTIFIER '=' base=IDENTIFIER '(' argument_list? ')'
        because_opt? ';'

// <connector> statement
connector_statement
        : annotation* 'connector' name=IDENTIFIER '=' base=IDENTIFIER '(' argument_list? ')'
        because_opt? ';'

// <provides> statement
provides_statement
        : annotation* 'provides' name=IDENTIFIER because_opt? ('{' provides_item* '}' | ';')
provides_item
        : property_statement

// <requires> statement
requires_statement
        : annotation* 'requires' name=IDENTIFIER variable_list? because_opt?
        ('{' requires_item* '}' | ';')
requires_item
        : property_statement

// <check> statement
check_statement
        : annotation* 'check' expression because_opt? ';'
```

```
// <property> statement
property_statement
        : annotation* 'property' name=IDENTIFIER ('=' expression)? because_opt? ';'

// <link> statement
link_statement
        : annotation* 'link' name=IDENTIFIER? link_source 'to' link_target because_opt?
        ('{' link_item* '}' | ';')
link_source
        : member_expression
link_target
        : member_expression
link_item
        : check_statement

// <supports> option
supports_opt
        : 'supports' IDENTIFIER (',' IDENTIFIER)*

// <inhibits> option
inhibits_opt
        : 'inhibits' IDENTIFIER (',' IDENTIFIER)*

// <because> option
because_opt
        : 'because' because_item (',' because_item)*
because_item
        : name=IDENTIFIER ('(' argument_list? ')')?

// <extends> option
extends_opt
        : 'extends' extendee=IDENTIFIER

// <@> annotation
annotation
        : '@'clss=IDENTIFIER? '(' annotation_node (',' annotation_node)* ')'
```

```
annotation_node
        : name=IDENTIFIER '=' expression

// Expressions
expression
        : expr=inner_expression
inner_expression
        : subsetof_expression
subsetof_expression
        : logicalOr_expression (SUBSETOF^ logicalOr_expression)*
logicalOr_expression
        : logicalAnd_expression (DIS^ logicalAnd_expression)*
logicalAnd_expression
        : equality_expresion (CON^ equality_expresion)*
equality_expresion
        : relational_expresion ((EQL | NEQ)^ relational_expresion)*
relational_expresion
        : acceptence_expression ((GTN | GTE | LTN | LTE)^ acceptence_expression)*
acceptence_expression
        : augmentation_expression (ACCEPTS^ augmentation_expression)*
augmentation_expression
        : additive_expression ((AUG | NAG)^ additive_expression)*
additive_expression
        : multiplicative_expression ((ADD | SUB)^ multiplicative_expression)*
multiplicative_expression
        : unary_expresion ((MUL | DIV | MOD)^ unary_expresion)*
unary_expresion
        : (CMP | NOT)^ unary_expresion
        | '+' expr=unary_expresion
        | '-' expr=unary_expresion
        | primary_expression
primary_expression
        : '('! inner_expression ')'!
        | member_expression
        | literal
member_expression
```

```
        : member_part ((DOT | IND)^ member_part)*
member_part
        : name=IDENTIFIER '(' argument_list? ')'
        | name=IDENTIFIER
literal
        : atom=INTEGER_LITERAL
        | atom=REAL_LITERAL
        | atom=BOOLEAN_LITERAL
        | atom=STRING_LITERAL
        | atom=DECLARATIVE_LITERAL
        | set_literal
string_literal
        : '\'' atom=SINGLE_QUOTE_TEXT '\''
set_literal
        : '[' set=set_element_list? ']'
set_element_list
        : set_element (','! set_element)*
set_element
        : literal
        | '(' pair=set_pair ')'
        | name=IDENTIFIER
set_pair
        : set_keyword ','! IDENTIFIER
        | IDENTIFIER ','! literal
set_keyword
        : 'provides'
        | 'requires'

// Formal parameter list
parameter_list
        : parms+=IDENTIFIER (',' parms+=IDENTIFIER)*

// Variable list
variable_list
        : vars+=IDENTIFIER (',' vars+=IDENTIFIER)*
```

```
// Argument list
argument_list
        : args+=expression (',' args+=expression)*

// Statement
statement
        : layer_statement
        | component_statement
        | connector_statement
        | requires_statement
        | provides_statement
        | link_statement
        | check_statement
        | property_statement

// Literals
INTEGER_LITERAL
        : DECIMAL_DIGIT+
        | '0' ('x' | 'X') HEX_DIGIT+

REAL_LITERAL
        : DECIMAL_DIGIT* '.' DECIMAL_DIGIT+

BOOLEAN_LITERAL
        : 'true'
        | 'false'

STRING_LITERAL
        : '\'' cs=SINGLE_QUOTE_TEXT '\''
        | '\"' cs=DOUBLE_QUOTE_TEXT '\"'

DECLARATIVE_LITERAL
        : '#\'' cs=SINGLE_QUOTE_TEXT '\''
        | '#\"' cs=DOUBLE_QUOTE_TEXT '\"'
```