TUMLRT: textual syntax for UMLRT

Reference Guide

Version 0.8

Zeligsoft (2009) Ltd.

February 16, 2016

Abstract

This document provides the reference for the grammar of TUMLRT, a textual concrete syntax for UMLRT [1, 3, 2]. It does not discuss the semantics of the language.

Disclaimer This documents presents an early version of the grammar and several aspects are subject to change.

Contents

1 Notation **3**

2 Grammar **4**
2.1 Models and packages 4
2.2 Classes 4
2.3 Capsules 5
2.4 Protocols 6
2.5 Behaviour 6
2.5.1 State machines 6
2.5.2 Opaque bahaviour 6
2.6 Actions 8
2.7 Common clauses 8

3 Runtime Services Library **10**

A Proposed Core Action Language for UMLRT **13**

Chapter 1

Notation

Table 1.1 shows the notation used in the language itself (the "object-language") and Table 1.2 shows the notation used in describing the grammar (the "meta-language").

Grammar expression	**Notation example**
Keywords	`message`
Delimiters	`{ }`
String literals	`"this is a string"`
Comments	`/* comment */`
Identifiers, names	`id`

Table 1.1: Object-language notation

Grammar expression	**Notation example**
Meta-variables	`<expression>`
Zero or one occurrence	`<expression>?`
Zero or more occurrences	`<expression>*`
One or more occurrences	`<expression>+`
Alternative expressions	`<expression1> | <expression2>`
Grouping	`( <expression> )`

Table 1.2: Meta-language notation

Chapter 2

Grammar

Tables 2.1 to 2.16 show the grammar in EBNF notation. Section 2.7 includes common rules among the grammar elements.

2.1 Models and packages

The top level elements are *models* (see Table 2.1). They consist of *class* (Section 2.2), *capsule* (Section 2.3), *protocol* (Section 2.4) and *package* definitions. Packages allow to group definitions as a unit, referable with a qualified name.

```
<model> ::=
    model <name>
    {
        <import>*
        ( <class> | <capsule> | <protocol> | <package> )*
    }

<package> ::=
    package <name>
    {
        <import>*
        ( <class> | <capsule> | <protocol> | <package> )*
    }

<import> ::= import uri <string-literal>;
```

Table 2.1: Models and packages

The URI in an import statement can be a standard platform URI as used in Eclipse.

2.2 Classes

Classes (Table 2.2) are defined by declaring their *attributes*, *operations* and optional *behaviour* (Section 2.5).

Defaults

	Attributes	Operations
Visibility	protected	public
Static	No	No
Read-only	No	N/A
Abstract	N/A	No
Query	N/A	No
Replication	1	N/A
Default-value	null	N/A
Parameters	N/A	()

```
<class> ::=
    class <name> ( <redefinition> )? ( <dependencies> )?
    {
        <description>?
        <attribute>*
        <operation>*
        <behaviour>?
        <annotation>*
    }

<attribute> ::=
    <visibility>? static? readonly? attribute <name> : <type-name> <replication>? ( = <val> )? ;

<operation> ::=
    <visibility>? static? abstract? query? operation <name> <parameters>? : <type-name> <body> ;

<parameters> ::= (<parameter> ( , <parameter> )*)

<parameter> ::= <direction> <name> : <type-name> <replication>?

<body> ::= <action>
```

Table 2.2: Classes

2.3 Capsules

Capsules (Table 2.3) are defined by a set of *ports*, *parts* and *connectors*. Additionally, capsules are classes, and therefore, they may have attributes, operations and behaviour as described in Section 2.2.

```
<capsule> ::=
    capsule <name> ( <redefinition> )? ( <dependencies> )?
    {
        <description>?
        <port>*
        <part>*
        <connector>*
        <attribute>*
        <operation>*
        <behaviour>?
        <annotation>*
    }

<port> ::=
    conjugate? <port-kind>? <registration>? port <name> : <protocol-name> <replication>? ;

<port-kind> ::= external | internal | relay | sap | spp

<registration> ::= auto | app | autolock

<part> ::= <part-kind>? part <name> : <capsule-name> <replication>? ;

<part-kind> ::= fixed | optional | plugin

<connector> ::= connect <connector-end> to <connector-end> ;

<connector-end> ::= ( <part-name> : )? <port-name>
```

Table 2.3: Capsules

Defaults

	Ports	Parts
Conjugate	No	N/A
Kind	external	fixed
Registration	N/A	auto
Replication	1	1

2.4 Protocols

Protocols (Table 2.4) are defined by a set of *messages*.

```
<protocol> ::=
    protocol <name> ( <redefinition> )? ( <dependencies> )?
    {
        <description>?
        <message>+
        <annotation>*
    }

<message> ::= <direction> message <name> <parameters> ;
```

Table 2.4: Protocols

2.5 Behaviour

Class and capsule behaviour can be specified in several ways. Currently supported are hierarchical state machines (Subsection 2.5.1) and opaque behaviours (Subsection 2.5.2). See . A future version may support activities (UML activity diagrams) and interactions (UML sequence diagrams).

2.5.1 State machines

State machines in UMLRT are a simplified form of hierarchical UML state machines. The main differences with UML state machines are:

- UMLRT state machines do not have orthogonal regions, so all states are OR-states, and there are are no fork and join pseudo-states,
- Transitions cannot "cross" state boundaries: the source and target of a transition must be in the same region or boundary. Transition chains can be formed by explicitly using entry and exit points,
- There is no shallow history, only deep history,
- A transition that ends on the boundary of a composite state is deemed to implicitly end in the composite state's deep history point, rather than its initial point,
- If the state machine belongs to a capsule, all transition triggers must have at least one port
- State machines have run-to-completion semantics: an event is fully processed by the state machine, starting in a (stable) state, following a transition chain (of one or more transitions), and ending in a (stable) state, before the next event is processed. The execution of such step is atomic: it cannot be interrupted, and it cannot end in a pseudo-state.

The textual syntax for state machines in TUMLRT is shown in Table 2.6.

2.5.2 Opaque bahaviour

Opaque behaviours provide a way to specify the behaviour of a capsule in terms of expressions outside the scope of the TUMLRT language. Such expressions are to be handled by external tools, code generators, etc.

An opaque behaviour can provide an *initialization* expression, a *handling* expression, and *extras* (see Table 2.7). The initialization expression is intended to specify the behaviour when the capsule is initialized. The handling expression is intended to specify the behaviour of the capsule when reacting to an external message on any port. The extras are intended to provide additional definitions required by the initialization and handling expressions.

```
<behaviour> ::= <state-machine> | <opaque-behaviour>
```

Table 2.5: Behaviours

```
<state-machine> ::=
    statemachine
    {
        ( initial <name> ; )?
        ( junction points { <name> ( , <name> )* } )?
        ( choice points { <name> ( , <name> )* } )?
        <state>*
        <transition>*
    }

<state> ::= <simple-state> | <composite-state>

<simple-state> ::=
     state <name> ;
|    state <name>
     {
         ( entry action <action> ; )?
         ( exit action <action> ; )?
         ( entry points { <name> ( , <name> )* } )?
         ( exit points { <name> ( , <name> )* } )?
     }

<composite-state> ::=
    composite state <name>
    {
        ( entry action <action> ; )?
        ( exit action <action> ; )?
        ( entry points { <name> ( , <name> )* } )?
        ( exit points { <name> ( , <name> )* } )?
        ( initial <name> ; )?
        ( history <name> ; )?
        ( junction points { <name> ( , <name> )* } )?
        ( choice points { <name> ( , <name> )* } )?
        <state>*
        <transition>*
    }

<transition> ::=
    transition <name>? from <name> to <name>
        ( triggers <triggers> )?
        ( guard <action> )?
        ( action <action> )? ;

<triggers> ::= <trigger> | { <trigger> ( , <trigger )* }

<trigger> ::=
     on <ports> : <message-name>
|    on <message-name> from <ports>

<ports> ::= <port-name> | { <port-name> ( , <port-name> )* }
```

Table 2.6: State Machines

```
<opaque-behaviour> ::=
    behaviour
    {
        ( init <string-literal> )?
        ( handle <string-literal> )?
        ( extras <string-literal> )?
    }
```

Table 2.7: Opaque behaviours

```
<action> ::= <string-literal>
```

Table 2.8: Actions

2.6 Actions

In its current version, only opaque actions are supported. The specification of these is given by a string literal (Table 2.8). Future versions may explicitly support some action language. Appendix A proposes a possible action language.

2.7 Common clauses

This section describes syntactic rules that are used in several constructs.

- Literals (Table 2.9) include constant values.
- Names and identifiers (Table 2.10) are used to identify different elements in the language. Qualified names allow to disambiguate conflicting names in a namespace.
- Descriptions (Table 2.11) provide a mechanism for persistent documentation, stored in the abstract syntax tree, unlike comments.
- Annotations (Table 2.12) are used to annotate different elements with information that can be used by different language processors. A typical use is to provide UML-like stereotypes. It is up to third-party tools to interpret these annotations.
- Redefinitions and dependencies (Table 2.13) are clauses that can be used in the definition of several elements, such as classes, capsules and protocols, to declare a given element definition as a redefinition of another, as well as to declare dependencies explicitly, if needed.
- Visibility (Table 2.14) declares the visibility of a given element.
- Replication (Table 2.15) declares the upper-bound multiplicity of a model element (attribute, parameter, port or part).
- Direction (Table 2.16) declares whether a parameter or message is input, output, or both.

```
<bool> ::= true | false

<nat> ::= 0 | 1 | 2 | ...

<string-literal> ::= <quote> <char>* <quote>

<val> ::= <bool> | <nat> | <string-literal>
```

Table 2.9: Literals

```
<letter> ::= a | ... | z | A | ... | Z

<digit> ::= 0 | ... | 9

<id> ::= <letter>( <letter> | <digit> | _ )*

<name> ::= <id>

<qualified-name> ::= <id> ( . <id> )*

<type-name> ::= <qualified-name>

<protocol-name> ::= <qualified-name>

<capsule-name> ::= <qualified-name>

<port-name> ::= <name>

<part-name> ::= <qualified-name>
```

Table 2.10: Names and identifiers

```
description <string-literal>
```

Table 2.11: Descriptions

```
<annotation> ::=
    annotation <name> (<annotation-parameter> ( , <annotation-parameter> )*);

<annotation-parameter> ::= <name> = <string-literal>
```

Table 2.12: Annotations

```
<redefinition> ::= redefines <qualified-name>

<dependencies> ::= dependencies <qualified-name> ( , <qualified-name> )*
```

Table 2.13: Redefinitions and dependencies

```
<visibility> ::= public | private | protected
```

Table 2.14: Visibility

```
<replication> ::= [ <nat> ]
```

Table 2.15: Replication

```
<direction> ::= in | out | inout
```

Table 2.16: Direction

Chapter 3

Runtime Services Library

Table 3.1 shows the textual representation of the runtime services library available to the user. Table 3.2 shows the built-in primitive types currently supported.

```
model RTSLibrary
{
    import uri "platform:/resource/org.eclipse.papyrusrt.rts/libraries/TUMLRT-Types.umlrt"

    protocol Timing
    {
        in message timeout();
        annotation RTSLibraryElement;
    }

    protocol Log
    {
        out message log(in msg : TypesLib.String);
        annotation RTSLibraryElement;
    }
}
```

Table 3.1: Runtime library

```
model TypesLib
{
    /* UML primitives */
    typedef primitivetype Boolean;
    typedef primitivetype Integer;
    typedef primitivetype Real;
    typedef primitivetype String;
    typedef primitivetype UnlimitedNatural;

    /* C/C++ primitives */
    typedef primitivetype bool;
    typedef primitivetype char;
    typedef primitivetype double;
    typedef primitivetype float;
    typedef primitivetype int;
    typedef primitivetype long;
    typedef primitivetype "long double";
    typedef primitivetype short;
    typedef primitivetype "unsigned char";
    typedef primitivetype "unsigned int";
    typedef primitivetype "unsigned long";
    typedef primitivetype "unsigned short";
    typedef primitivetype void;

    /* Ansi C Library */
    typedef primitivetype int8_t;
    typedef primitivetype int16_t;
    typedef primitivetype int32_t;
    typedef primitivetype int64_t;
    typedef primitivetype uint8_t;
    typedef primitivetype uint16_t;
    typedef primitivetype uint32_t;
    typedef primitivetype uint64_t;
    typedef primitivetype wchar_t;
}
```

Table 3.2: Built-in types

Bibliography

[1] B. Selic. Using UML for modeling complex real-time systems. In Frank Mueller and Azer Bestavros, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474 of *LNCS*, pages 250–260. Springer, 1998.

[2] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object Oriented Modeling*. Wiley & Sons, 1994.

[3] B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Whitepaper, Rational Software Corp., 1998.

Appendix A

Proposed Core Action Language for UMLRT

This section describes the syntax for CALUR, a proposed action language for UMLRT.

```
<statement> ::=
     send <port-ref>:<message-name> <args>? ;
|    incarnate <capsule-name> at <part-ref> <args>? ;
|    destroy <part-ref> ;
|    import <capsule-name> at <part-ref> <args>? ;
|    deport <part-ref> ;
|    register <port-ref> to <port-ref> ;
|    deregister <port-ref> ;
|    var <name> : <type-name> ;
|    <var-name> := <expr> ;
|    <var-name> . <operation-name> <args>? ;
|    if <expr> then <statement> ( else <statement> )?
|    while <expr> do <statement> ;
|    { <statement>* }

<args> ::= ( <expr> ( , <expr> )* )

<port-ref> ::= <port-name> <index>?

<part-ref> ::= <part-name> <index>?

<index> ::= [ <expr> ]

<expr> ::=
     <literal>
|    <var-name>
|    <unary-op> <expr>
|    <expr> <binary-op> <expr>
|    <var-name> . <operation-name> <args>?
```

Table A.1: CALUR statements