Expressions in REMES

Marin Orlić
University of Zagreb, Faculty of Electrical Engineering and Computing
marin.orlic@fer.hr

December 2012

Abstract

REMES is a graphical modeling language used to describe time- and resource-wise behavior in component-based embedded systems. Internal behavior of a component is modeled with REMES modes connected with edges. Both modes and edges can be annotated with expressions – invariants, guards and actions. Mode invariants are expressions which specify a continuous, timed, action of a system. Edge guards specify logical or timed conditions that must be fulfilled for corresponding edge to be enabled. Edge action expressions are executed when an edge is taken, and typically update variables. This document describes the syntax of invariant, guard and action expressions.

1 Introduction

REMES is a behavioral modeling framework tailored to model functional, timing and resource-usage embedded behavior. REMES supports hybrid systems (systems with discrete and continuous behavior) and abstract resources consumed in discrete or continuous fashion. With a generic resource model, designers can model typical computing resources such as CPU, memory, ports, or energy. For more details on REMES please consult [1, 2].

REMES models are created in a graphical editor implemented within the Eclipse environment [3,4]. We assume that you are familiar with diagram editors created with Eclipse [5] and GMF [6].

This document describes the structure of expressions used to describe invariants, guards and actions in REMES models. REMES expression syntax is influenced with the syntax for expressions used throughout UPPAAL family of tools [7, 8]. Expression syntax and semantics are similar to those of a general-purpose programming language – where necessary, REMES specifics will be noted.

Figure 1 illustrates elements of REMES models: a composite mode with compartments labeled *Constants*, *Variables* and *Resources* used to declare respective entities. Constants and variables can be defined and modified in the fields of the Properties view. A shorthand to create a variable with type exists in the form of `name : type` to declare a variable with a name and type, or the following to declare an array variable with a name, type and array size `name[size] : type`. Similar shorthand can be used to declare constants in the form `name=value`, but

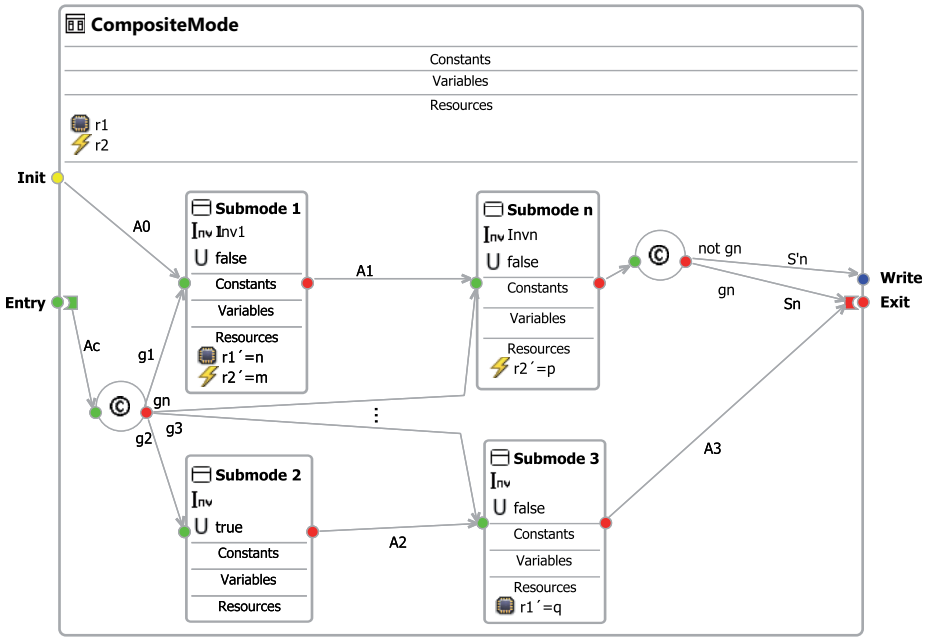

Figure 1: A REMES Composite Mode.

the type needs to be entered additionally. The shorthand expressions should be entered in the name field of the editor's Properties view.

Resource shorthand declares a resource with its consumption rate, `res'=rate` declares a resource named `res` that is consumed with a rate of 50 resource units per time unit: `res'=50`. Resource properties need to be adjusted in Properties view. Resource expressions will appear next to resource icons, similar to `r1` and `r2` in Figure 1.

2 Expressions

2.1 Invariants and guards

The composite mode in Figure 1 consists of four submodes – *Submode 1* and *Submode n* have their invariants set. Invariant placeholders *Inv1* and *Invn* are predicates over clocks (logical expressions with clocks) such as `clk > 5 && clk <= 10`. Edges connecting modes can have guards (labeled with *g1*, *g2*, *g3*, *gn*), which follow the structure of invariants. There are limitations to operations allowed in an invariant.

Operations with clocks. Clocks are declared as variables of type clock, but behave differently than variables of other types. Operations with clocks are somewhat limited:

1. clocks can be compared to constants or numeric variables,
2. clocks can not be compared to each other

Operator group	Operators by priority
Unary	`+`, `-`, `!` / `not`
Arithmetic	`*`, `/`, `%`, `+`, `-`
Comparison	`<`, `<=`, `>=`, `>`, `==`, `!=`
Logical	`&&` / `and`, `||` / `or`
Conditional	`? :`
Assignment	`:=`, `+=`, `-=`, `*=`, `/=`, `%=`
Rate (derivation)	`'`

Table 1: Operator precedence

These limitations are not checked, but analysis of models violating is limited: violation of 1 prohibits conversion of models to UPPAAL for analysis, and violation of 2 will affect the simulation results.

Clocks can be compared with constants to specify timing behavior. However, the result of this comparison is a time interval, and not a boolean value and cannot be used in context which requires a boolean value. For example, expression `(clk > 2) ? clk < 3 : clk > 1` is not allowed.

2.2 Action statements

Action statements annotate the edges in a model, labeled with *A0* .. *A3*, *Ac*, *Sn* and *S'n* in Figure 1. Actions statements are sequences of statements separated by semicolons. Each statement is a variable or resource update or a clock reset: `res += 10; clk := 0`.

2.3 Expression syntax

Three kinds of expressions exist in REMES: invariants/guards, actions and resource updates. Actions accept a sequence of assignment expressions, interpreted as variable updates or clock resets. Invariants and guards accept only a single, logical, expression. For more details, consult the full grammar in Appendix A.1 – this is described with `<action_expression>`, `<invariant_expression>` and `resource_expression`.

2.4 Operators

Unary operators. Unary operators `+` and `-` are the simplest as they do not change the argument type.

Logical operators. Logical operator `&&`, `||`, `!`, `and`, `or`, `not` require that both the left and right-hand sides are of type boolean. The inferred type is also boolean.

Arithmetic operators. Arithmetic operators are `+`, `-`, `*`, `/`, and `%`. Table 2 lists the type inference rules for arithmetic operators.

Equality and comparison operators. Equality and comparison operators are handled differently. Equality operators are `==` and `!=`. Comparison operators are `<`, `<=`, `>=`, and `>`. Table 3 and Table 4 lists the type inference rules for equality and comparison operators, respectively.

Assignment and assignment-arithmetic operators. Table 5 lists the type inference rules for the assignment operator `:=`. Table 6 lists the type inference rules for assignment-arithmetic operators `+=`, `-=`, `*=`, `/=`, and `%=`.

References

[1] C. Seceleanu, A. Vulgarakis, and P. Pettersson, "Remes: A resource model for embedded systems," in *Proc. of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. IEEE Computer Society, June 2009.

[2] A. Vulgarakis and C. Seceleanu, "Embedded Systems Resources: Views on Modeling and Analysis," *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, Workshop on Component-Based Design of Resource-Constrained Systems*, pp. 1321–1328, July 2008.

[3] D. Ivanov, M. Orlic, C. Seceleanu, and A. Vulgarakis, "REMES Tool-chain - A Set of Integrated Tools for Behavioral Modeling and Analysis of Embedded Systems," *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, September 2010.

[4] M. Orlic, A. Vulgarakis, and M. Zagar, "Towards Simulative Environment for Early Development of Component-Based Embedded Systems," *Proceedings of the 15th International Workshop on Component-Oriented Programming*, June 2010.

[5] The Eclipse Foundation, "Eclipse Platform," http://www.eclipse.org/ (Last Accessed: 2012-02-01).

[6] ——, "Eclipse Graphical Modeling Project – GMF Tooling," http://www.eclipse.org/modeling/gmp/ (Last Accessed: 2012-02-01).

[7] "UPPAAL," www.uppaal.com, (Last Accessed: 2012-02-27).

[8] "UPPAAL CORA," http://people.cs.aau.dk/~adavid/cora/, (Last Accessed: 2012-03-27).

A Appendices

A.1 Remes expressions grammar

⟨*action_expression*⟩ ::= ⟨*assignment_expression*⟩ (‘`,`’ ⟨*assignment_expression*⟩)*
  | ⟨*empty*⟩

⟨*invariant_expression*⟩ ::= ⟨*ternary_expression*⟩
  | ⟨*empty*⟩

⟨*resource_expression*⟩ ::= ⟨*identifier*⟩ ‘`’` ‘`=`’ ⟨*value_expression*⟩
  | ⟨*empty*⟩

⟨*assignment_expression*⟩ ::= ⟨*variable*⟩ ⟨*assignOp*⟩ ⟨*ternary_expression*⟩

⟨*ternary_expression*⟩ ::= ⟨*or_expression*⟩ (‘`?`’ ⟨*or_expression*⟩ ‘`:`’ ⟨*or_expression*⟩)?

⟨*or_expression*⟩ ::= ⟨*and_expression*⟩ (⟨*orOp*⟩ ⟨*and_expression*⟩)* ;

⟨*and_expression*⟩ ::= ⟨*compare_expression*⟩ (⟨*andOp*⟩ ⟨*compare_expression*⟩)*

⟨*compare_expression*⟩ ::= ⟨*add_expression*⟩ (⟨*compareOp*⟩ ⟨*add_expression*⟩)*

⟨*add_expression*⟩ ::= ⟨*multiply_expression*⟩ (⟨*addOp*⟩ ⟨*multiply_expression*⟩)*

⟨*multiply_expression*⟩ ::= ⟨*sign_expression*⟩ (⟨*mulOp*⟩ ⟨*sign_expression*⟩)*

⟨*sign_expression*⟩ ::= (‘`+`’ | ‘`-`’) ⟨*sign_expression*⟩ | ⟨*unary_expression*⟩

⟨*value_expression*⟩ ::= (‘`+`’ | ‘`-`’) ⟨*sign_expression*⟩ | ⟨*constant*⟩

⟨*unary_expression*⟩ ::= (⟨*notOp*⟩)* ⟨*primary_expression*⟩

⟨*primary_expression*⟩ ::= ⟨*variable*⟩ | ⟨*literal*⟩ | ⟨*constant*⟩ | ⟨*par_expression*⟩

⟨*variable*⟩ ::= ⟨*identifier*⟩ (‘`[`’ ⟨*ternary_expression*⟩ ‘`]`’)?

⟨*identifier*⟩ ::= (‘`a`’..‘`z`’ | ‘`A`’..‘`Z`’ | ‘`_`’) (‘`a`’..‘`z`’ | ‘`A`’..‘`Z`’ | ‘`0`’..‘`9`’ | ‘`_`’)*

⟨*literal*⟩ ::= ‘`true`’ | ‘`false`’

⟨*constant*⟩ ::= ⟨*natural*⟩ | ⟨*float*⟩

⟨*par_expression*⟩ ::= ‘`(`’ ⟨*invariant_expression*⟩ ‘`)`’

⟨*natural*⟩ ::= (‘`1`’..‘`9`’)(‘`0`’..‘`9`’)* | ‘`0`’

⟨*float*⟩ :== (‘`1`’..‘`9`’) (‘`0`’..‘`9`’)* | ‘`0`’) ‘`.`’ (‘`0`’..‘`9`’)+

⟨*compareOp*⟩ ::= ‘`<`’ | ‘`<=`’ | ‘`==`’ | ‘`!=`’ | ‘`>`’ | ‘`>=`’

⟨*addOp*⟩ ::= ‘`+`’ | ‘`-`’

⟨*mulOp*⟩ ::= ‘`*`’ | ‘`/`’ | ‘`%`’

⟨*notOp*⟩ ::= ‘`!`’ | ‘`not`’

⟨*andOp*⟩ ::= ‘`&&`’ | ‘`and`’

⟨*orOp*⟩ ::= ‘`||`’ | ‘`or`’

⟨*assignOp*⟩ ::= ‘`:=`’ | ‘`+=`’ | ‘`-=`’ | ‘`*=`’ | ‘`/=`’ | ‘`%=`’

A.2 Type of operation results

	Right					
Left	integer	boolean	natural	clock	float	resource
integer	integer	–	integer	clock	float	–
boolean	–	–	–	–	–	–
natural	integer	–	integer	clock	float	–
clock	clock	–	clock	clock	–	–
float	float	–	float	–	float	–
resource	resource	–	resource	–	–	resource

Table 2: Type inference rules for arithmetic operators

	Right					
Left	integer	boolean	natural	clock	float	resource
integer	boolean	–	boolean	boolean	boolean	–
boolean	–	boolean	–	–	–	–
natural	boolean	–	boolean	boolean	boolean	–
clock	boolean	–	boolean	boolean	–	–
float	boolean	–	boolean	–	boolean	–
resource	–	–	–	–	–	–

Table 3: Type inference rules for equality operators

	Right					
Left	integer	boolean	natural	clock	float	resource
integer	boolean	–	boolean	boolean	boolean	–
boolean	–	–	–	–	–	–
natural	boolean	–	boolean	boolean	boolean	–
clock	boolean	–	boolean	boolean	–	–
float	boolean	–	boolean	–	boolean	–
resource	–	–	–	–	–	–

Table 4: Type inference rules for comparison operators

	Right					
Left	integer	boolean	natural	clock	float	resource
integer	integer	–	integer	–	integer	–
boolean	–	boolean	–	–	–	–
natural	natural	–	natural	–	natural	–
clock	clock	–	clock	clock	–	–
float	float	–	float	–	float	–
resource	resource	–	resource	–	–	resource

Table 5: Type inference rules for assignment operators

	Right					
Left	integer	boolean	natural	clock	float	resource
integer	integer	–	integer	–	integer	–
boolean	–	–	–	–	–	–
natural	natural	–	natural	–	natural	–
clock	clock	–	clock	clock	–	–
float	float	–	float	–	float	–
resource	resource	–	resource	–	–	resource

Table 6: Type inference rules for assignment-arithmetic operators