RSL* extensions for RSL

Anne E. Haxthausen
DTU Compute, 4 March 2024

LEXICAL MATTERS:

Additional keywords: (must not be used as identifiers)
array, of, transition_system, init_constraint, transition_relation, transition_rules, where, ltl_assertion
Additional symbols:
{. , .} , ==> , | -, [=], [>], G , F
To be decided: should G, F be treated as symbols or pre-defined identifiers.
We could choose [>] to have lower/higher/same precedence than [=] and same (right) associativity.

SYNTAX EXTENSIONS:

Syntax conventions: are as in the document on the RSL subset, except that ascii equivalents are used for symbols.

1. Extensions allowing declarations of generic values and variables

```
value_def            ::= ... |  generic_value_def
generic_value_def   ::=  id  [ single_typing-list ] : type_expr

variable_def            ::= ... |  generic_variable_def
generic_variable_def   ::=  id  [ single_typing-list ] : type_expr
```

NOTE1 - pre condition for unfolder: any type_expr in single_typing-list of a generic_value_def or a generic_variable_def must be statically evaluable to a finite subset of Int or of a variant type.

NOTE2 – grammar rule integration : generic_value_def can be integrated with value_signature by changing the latter to:

```
value_signature  ::=  id  opt-formal_index : type_expr
formal_index     ::=  [ typing-list ]
```

Similarly generic_variable_def can be integrated with variable_def by changing the latter to:

```
variabel_def ::=  id  opt-formal_index : type_expr opt-initialisation
```

I would prefer not to integrate.

2. Extensions allowing access to elements of generic values and variables

```
value_expr ::= ... |  gen_access_expr
gen_access_expr       ::=  id  [ value_expr-list ]
```

NOTE1 - context conditions: the id in a gen_access_expr must have been declared as a generic value or generic variable and the length and types of the value_expr-list must match the single_typing-list in the declaration of id. Each value expr in the value_expt-list must be pure (i.e. not refer to variables) to be able to unfold.

NOTE2 - pre condition for unfolder: Each value expr in the value_expt-list must be statically evaluable to an Int value or a variant value.

NOTE3 – grammar rule integration: If extension 3 (below) is also made, gen_access_expr and array_acces_expr must be integrated to access_expr, see Section 4.

3. Extensions allowing array types and values and accesses

type_expr ::= ... | array_type_expr

array_type_expr ::= **array** type_expr **of** type_expr

value_expr ::= ... | enumerated_array_expr | array_access_expr

enumerated_array_expr ::= {. value_expr-list .}

array_access_expr ::= *array*-value_expr [value_expr]

NOTE1 - context conditions for an array_type_expr: In the extension for that made by Jacob, the index type (i.e. the first type_expr (after key word array) is allowed to be anything (needs not to be a finite integer interval starting in 0 or at all to be an integer type), and the element type (i.e. the second type_expr) can be anything, e.g. an array_type, so one can have arrays of arrays. In Signe's examples, index types are Integer intervals starting in 0. For our language or as a pre condition for the unfolder, we might restrict the allowed types. e.g. index types to be subsets of Int..

NOTE2 - context conditions for enumerated_array_expr.: Invented by Signe. No documentation for exists. Probably the element value_exps must be pure, and definitely they must have a common type.

PROBLEM with index type of enumerated_array_expr: It is not possible to infer the index type of an enumerated_array_expr alone (unle/ss index types were limited to be integer intervals starting in 0), and it is unknown what semantics of an expression like {. 8, 9, 10 .} [2] should be and whether it would be defined. In contrast to that, if we have a declaration A : array {| i : Int :- i isin {2..4} |} = {. 8, 9, 10 .}, then we could assume that the index type for the enumeration should be the index type of A and then A[2] would be defined and evaluate to 8. So an enumerated_array_expr needs a context from which its index type can be decided, and that index type must have an ordering (otherwise we do not know which element belongs to a certain index.)
Solution 1: One possibility could be to treat an enumerated_array_expr {. e1, ..., en .} as having index type **array** {| i Int :- i isin {0 .. n} |} **of** T, where T is the common type of the elements (value_exprs).
Solution 2: For {. e1, ..., en .} infer instead the type **array** any **of** T, where T is the common type of the elements (value_exprs) and allow only this, if its index type can be inferred from its context. With the current grammar and context rules, an array can only appear as the *array*-value_expr in an array_access_expr, or as the lhs or rhs of an equation or as argument to a function taking an array as a parameter. The first should be disallowed, the last allowed, and the middle allowed when the index type in known on one of the sides. E.g. expressions like {. 8, 9, 10 .} [2] or {. 8, 9, 10 .} = {. 7+1, 9, 10 .} should be disallowed. (This can easily be checked by requiring certain array types not to contain any) . The use of any complicates the whole type inference system (as then we need rules for how to match types containing type any). In this case any should only be matched with types having an ordering.
Solution 1 is the simplest, but also the less general one.

NOTE3 - context conditions for an array_access_expr (a[v]), the first value_expr (a) must have an array type (array ty1 of t) and the second value_expr (v) must have index type (ty1) as type.

NOTE4: This grammar for array_access_expr allows for indexing arrays of arrays, like in *a[i][j]*.

NOTE5 – alternative, more restricted syntax for array_access_expr: The grammar for array_access_expr allows the *array*-value_expr to be an enumerated_array_expr. This possibility might not be needed and it has the problems described above. A possibility is to prevent such array_access_expr using the following restricted grammar:

array_access_expr ::= id [value_expr] | array_access_expr [value_expr]

NOTE6 - grammar rule integration: If extension 2 (above) is also made, gen_acces_expr and array_access_expr must be integrated to access_expr, see Section 4.

4. Syntax integration of gen_access_expr and array_access_expr to access_expr

Syntax integration alternative 1 (based on the restricted array_access_expr as defined in NOTE 5):

access_expr   ::= id index | access_expr index
index   ::= [value_expr-list]

NOTE1: The above grammar allows only for access expressions of the form id[v1, .]...[vm].

NOTE2 - context conditions for access_expr:
For case *id[vlist1]*: Either (1) the id has been declared as a generic value or generic variable *id[tlist1] : T* and vlist1 and tlist1 match each other (have same length and elementwise matching types), or (2) the id has been declared as a non-generic array value/variable *id : array T1 of T* and *vlist1* has length 1 and type T1. The whole access_expr will then get type *T.*
For case *access_expr index:*   The access_expr must have an array type *T = array T2 of T3* and the index must only contain one value_expr and that must have type *T2*. The whole access_expr will then get type *T3.*

NOTE3 - alternative grammar rule without left-recursion for access_expr allowing for the same, (makes the static semantics cumbersom to write):
access_expr   ::= id index-string
index-string ::= index | index index-string

NOTE4 - alternative syntax integration 2, now also integrating with the name (i.e. id) alternative of a value_expr: So access_expr now integrates name (which is id), gen_access_expr and array_access_expr
access_expr   ::= id | access_expr index
This extension allows for some more syntactically correct strings, that must then be ruled out by the type checker: a value_expr can now be an access-expr, which is the id of a generic value/variable (before it had to be the id of a non-generic value/variable).

NOTE5 - alternative syntax integration 3 (based on array_access_expr as defined in start of section 3)
access_expr   ::= value_expr index
or access_expr   ::= *generic_value_or_variable*-id index | *array*-value_expr index | *array*-access_expr index

These two are the most general solutions. The latter is ambigous as *array*-value_expr can be an id.
For the former access_expr rule to work, value_expr which is a name (i.e. an id) should be allowed to be the id of a generic value/variable (before it had to be the id of a non-generic value/variable).

5. Extensions allowing declarations of transistion systems and ltl_properties

decl   ::= ... | transition_system | ltl_decl

transition_system   ::=
transition_system [id]
opt-variabel_decl
opt-init_constraint_decl
transition_relation
end

5.1 variable_decl was defined for RSL (subset) and extended in Section 1.
5.2 initialisation constraints

init_constraint_decl ::= **init_constraint** *logical*-value_expr

NOTE For the unfolder a more restrictive syntax may be required only allowing certain forms of value_expr. The same holds for axiom_def.

5.3 transition relation

transition_relation ::= transition_rules | transition_expr

transition_rules ::= **transition_rules** transition_rule opt-named_rules_decl

transition_rule ::=
guarded_command |
quantified_rule |
rule_choice |
bracketed_rule |
rule_name

guarded_command ::= opt-axiom_naming *logical*-value_expr ==> effect-list

effect ::= primed_name = value_expr |
primed_access_expr = value_expr |
(all single_typing-list :- effect)

primed_name ::= *variable-id '*
primed_access_expr ::= primed_name index | primed_access_expr index

NOTE: Recommended to forbid in RSL* the declaration of variables having a prime inside their ids (although allowed in RSL) - to a void confusion with primed variable names, *id'*, used to denote the post version of a variable having name, *id*.

QUESTION – what should be required for an index in a primed_access_expr: are they allowed to contain variables and primed variables? It seems as rsltc allows both, but they have not been used in any examples.

quantified_rule := ([=] single_typing-list :- transition_rule)

rule_choice := transition_rule choice_op transition_rule

choice_op ::= [=] | [>]

bracketed_rule ::= (transition_rule)

rule_name ::= id

named_rules_decl ::= **where** named_rule-list

named_rule ::= [id] = transition_rule

transition_expr ::= **transition_relation** *logical*-value_expr --- NEW like a post_expr, can wait

NOTE: For the latter rule to be useful, value_expr must be extended and the type checker should only allow these extensions inside a transition_expr or maybe inside an index in an effect, cf QUESTION above:

value_expr ::= ... | primed_name | primed_access_expr -- only needed, if transition_expr is included

5.4 ltl assertions

ltl_decl ::= **ltl_assertion** ltl_assertion-list **end**

ltl_assertion ::= [id] id |- ltl_formula

ltl_formula ::= *logical*-value_expr

NOTE: This extension was implemented in rsltc as follows: ltl_formula is evaluated in a value env extended (by overwriting) to contain G and F of type Bool -> Bool. So G and F were treated as special predefined functions.

NOTE alternative: One might instead introduce G and F as keywords and specialize the syntax for ltl_formulas.