3 Research

This section will briefly introduce RSL*, by introducing the concepts of *generic* specifications in depth compared to Chapter 1 and the process of unfolding and translating to RTT, the input language to RT-Tester. It will continue investigating the shortcomings in `rsltc`, and the outcome will determine if other alternatives should be investigated.

3.1 RSL*

RSL* (pronounced R-S-L-Star) is a specification language that extends RSL-SAL [4]. RSL-SAL is an extension of RSL that targets the SAL model checker and introduces the array type expression, transition system and more. RSL is the specification language, i.e. notation, used with the RAISE method. RSL* extends RSL-SAL with *generic* constants and variables – and the ability to use these when specifying the transition system, its transition rules and when writing LTL (Linear Temporal Logic) assertions. All of which are extensions of RSL-SAL.

Historically, some extensions in RSL* originate from RSL-SAL. However, to keep it simple in this thesis, RSL* will be presented as an extension to RSL, i.e. no prior knowledge of RSL-SAL or the SAL framework is assumed.

In the scope of RSL*, a *generic* specification is a specification that contains sorts and/or *generic* constants and/or *generic* variables. When a constant or variable is said to be *generic*, it is a family of constants/variables all with the same type and as many members as values in the type of which the constant/variable is *generic* over. For example, in the next section, the *generic* specification contains a *generic* variable `position` declared as `position[ t : TrainId ] : SegmentId`, then `position` is a family of variables all with the type `SegmentId` and there exists a member for each value in the type `TrainId`.

3.2 Hands-on walkthrough

This walkthrough aims to introduce the concept of *generic* specifications by creating a specification of trains driving, according to some rules, in a simplified railway network. Most new constructs in RSL* will be introduced in Section 3.3. The railway network, or just the network, is a simplified network created to illustrate the concept of *generic* specifications. Railway networks are typically more complex, including signals, switch boxes, sensors, actuators, stops and much more [6].

The goal is to create a *generic* specification of a railway network depicted in Figure 3. It is a network containing a single linear track divided into segments ranging from S_0 to S_{max} where max is an integer larger than 0. Trains are here assumed to be shorter than the length of a segment. A train can move right to the next segment if that segment is not occupied and move left if the left segment is not occupied by another train. Thus, no trains could move if there was one train on each segment.

The goal is that no train will enter a segment if that segment is occupied by another train, and no segments will contain more than one train at any time. That train can move freely to an adjacent segment if that segment is not occupied by another train. Lastly, it is assumed that when a train moves left/right to the next segment, the whole train is moved in one step, so there is no intermediate step where the train is both on the original and the next segment and only one train can move at a time, no two trains can enter the same segment at the same time.

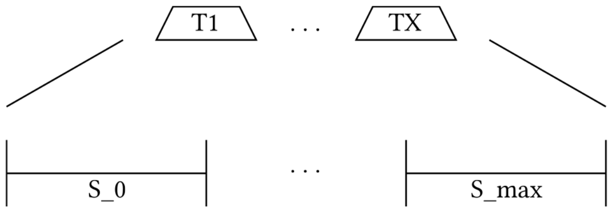

Figure 3: *Generic* simple railway network.

3.2.1 Generic specification

The network depicted in Figure 3 can be specified as `SimpleRail_generic` in Listing 1.

```
scheme SimpleRail_generic =
    class
        type
            TrainId
            SegmentId = {| n : Int :- n >= 0 /\ n < max |}
        value
            max : Int

        transition_system
            [TS]
            variable
                position [ t : TrainId ] : SegmentId,
                occupied [ s : SegmentId ] : Bool

            transition_rules
                MOVE_LEFT [=] MOVE_RIGHT

                where
                [ MOVE_RIGHT ] =
                    (([=] t : TrainId, s1 : SegmentId, s2 : SegmentId :-
                        position[t] < (max - 1) /\
                        position[t] = s1 /\
                        (s1 + 1) = s2 /\
                        ~occupied[s2] ==>
                            position'[t] = position[t] + 1,
                            occupied'[s1] = false,
                            occupied'[s2] = true)),

                [ MOVE_LEFT ] =
                    (([=] t : TrainId, s1 : SegmentId, s2 : SegmentId :-
                        position[t] > 0 /\
                        position[t] = s1 /\
                        (s1 - 1) = s2 /\
                        ~occupied[s2] ==>
                            position'[t] = position[t] - 1,
                            occupied'[s1] = false,
                            occupied'[s2] = true))
            end

    ltl_assertion
        [one_train_per_section] TS |-
          G(all t1: TrainId, t2: TrainId :-
            t1 ~= t2 => position[t1] ~= position[t2]),
```

```
44        [occupied_correct] TS |-
45          G(all t: TrainId, s: SegmentId :-
46            position[t] = s => occupied[s])
47
48      end
```

Listing 1: Simple railway network used to explain RSL* (`SimpleRail_generic.rsl`).

The type declaration in lines three to five, Listing 2, specifies the train identifiers by the *sort definition* `TrainId` and segment identifiers by the *abbreviation definition* `SegmentId` having a *subtype expression* ranging the integers from 0 through max. This enables the *generic* specification to have a well-known max of segments.

```
3         type
4             TrainId
5             SegmentId = {| n : Int :- n >= 0 /\ n < max |}
```

Listing 2: Type declaration, extract from Listing 1.

The value `max` used to limit `SegmentId` is an under-specified value in line 7, Listing 3.

```
6         value
7             max : Int
```

Listing 3: Value declaration, extract from Listing 1.

Following this is the transition system, named `TS`. It consists of two *generic* variables, `position` records a train's, `t`, position on the track using `SegmentId` and `occupied` records if a given segment is occupied by a train[1]. This is in lines 11 through 13, Listing 4, `position` is said to be *generic* over the type `TrainId`.

```
11            variable
12                position [ t : TrainId ] : SegmentId,
13                occupied [ s : SegmentId ] : Bool
```

Listing 4: Transition system variable declaration, extract from Listing 1.

Finishing the transition system declaration is the transition rules declaration. Lines 14 and 15, Listing 5, is a non-deterministic choice between two named rules, `MOVE_LFT` and `MOVE_RIGHT`.

```
15            transition_rules
16                MOVE_LEFT [=] MOVE_RIGHT
```

Listing 5: Transition system transition rules declaration, extract from Listing 1.

Named rules are declared using the `where` keyword in a transition rules declaration, as seen in lines 18 to 27 in Listing 6. `MOVE_RIGHT` are a quantified expression of overall values in the `TrainId`, `SegmentId` and `SegmentId` type and expresses that for each value, there should be a rule as expressed by the inner expression, lines 21 to 27. The rules follow a guarded-command structure, having a guard and a list of effects. The effect is updating a primed version of a variable with a new value; the structure will

[1]It is assumed that a Train only is capable of occupying a single Segment.

be further introduced in Section 3.3. Named rules are not mandatory, but it is convenient to separate and reuse rules multiple times. `MORE_LEFT` is in lines 29 through 37.

```
18                  where
19                  [ MOVE_RIGHT ] =
20                      (([=] t : TrainId, s1 : SegmentId, s2 : SegmentId :-
21                          position[t] < (max - 1) /\
22                          position[t] = s1 /\
23                          (s1 + 1) = s2 /\
24                          ~occupied[s2] ==>
25                              position'[t] = position[t] + 1,
26                              occupied'[s1] = false,
27                              occupied'[s2] = true)),
```

Listing 6: Transition system transition rules declaration, extract from Listing 1.

3.2.2 Generic and concrete

The previous section introduced a *generic* specification for a simplified railway "network". The network is challenging to model check, and in the case of using RT-Tester, it is impossible because RT-Tester's model checker cannot support *generic* constants and variables.

Even though the specification cannot be analysed as is, it can be of great value. The specification, although simplified, can express an unlimited number of unique instances of a railway network - any number of segments with any number of trains – bear in mind the limitations and that a state explosion can occur given a large enough network.

This goes well with systems that are also *generic*. For example, the European Train Control System, ETCS, is partly deployed in Denmark. ETCS is a generic train management system, i.e. it must be fitted to a given line or region to function. The Danish railway network is divided into several regions, each with its instance of a unique ETCS system instance. All deployed systems must behave the same, despite having different setups.

Thus, having a *generic* system and a *generic* specification go hand in hand to enable analysis tools to analyse a new instance of a system without specifying the system from scratch.

Continuing with the *generic* simple railway network, there exists an instance containing five segments and two trains, as depicted in Figure 4.

Figure 4: *Concrete* simple railway network.

3.2.3 Concrete specification

Luckily, the specification for the network in Figure 4 does not have to be done from scratch as we can *concretise* the specification in Listing 1 getting Listing 7, being a concrete specification of Figure 4.

```
scheme SimpleRail =
    class
        type
            TrainId == t1 | t2,
            SegmentId = {| n : Int :- n >= 0 /\ n < max |}
        value
```

```
            max : Int
        axiom
            max = 5

        transition_system
            [TS]
            variable
                position [ t : TrainId ] : SegmentId,
                occupied [ s : SegmentId ] : Bool

            init_constraint
                position[t1] = 0 /\
                position[t2] = 3 /\
                occupied[0] = true /\
                occupied[1] = false /\
                occupied[2] = false /\
                occupied[3] = true /\
                occupied[4] = false

            transition_rules
                MOVE_LEFT [=] MOVE_RIGHT

                where
                [ MOVE_RIGHT ] =
                    (([=] t : TrainId, s1 : SegmentId, s2 : SegmentId :-
                        position[t] < (max - 1) /\
                        position[t] = s1 /\
                        (s1 + 1) = s2 /\
                        ~occupied[s2] ==>
                            position'[t] = position[t] + 1,
                            occupied'[s1] = false,
                            occupied'[s2] = true)),

                [ MOVE_LEFT ] =
                    (([=] t : TrainId, s1 : SegmentId, s2 : SegmentId :-
                        position[t] > 0 /\
                        position[t] = s1 /\
                        (s1 - 1) = s2 /\
                        ~occupied[s2] ==>
                            position'[t] = position[t] - 1,
                            occupied'[s1] = false,
                            occupied'[s2] = true))
            end

    ltl_assertion
        [one_train_per_section] TS |-
          G(all t1: TrainId, t2: TrainId :-
            t1 ~= t2 => position[t1] ~= position[t2]),
        [occupied_correct] TS |-
          G(all t: TrainId, s: SegmentId :-
            position[t] = s => occupied[s])

    end
```

Listing 7: *Concrete* simple railway network (`SimpleRail.rsl`).

Concretiseing a *generic* specification is done by “removing” under-specified types and providing an (initial) value for all *generic* constants and variables, as seen in Listing 7.

The first change is line 4, Listing 8. Here, `TrainId` is a variant definition with two choices, `t1` for one train and `t2` for the other.

```
4           TrainId == t1 | t2,
```

Listing 8: Type declaration, extract from Listing 7.

The type abbreviation `SegmentId` is untouched in the type declaration, as well as the value `max` in the value declaration. `max` is given a value, 5, in lines 8 and 9, as an axiom declaration, Listing 9.

```
8       axiom
9           max = 5
```

Listing 9: Axiom declaration, extract from Listing 7..

Moving further to the transition system, the *generic* variables are given an initialisation constraint using the `init_constraint` declaration as seen in lines 17 to 24, Listing 10. Both trains are given an initial position, and the *generic* variable `occupied` is given initial values accordingly.

```
17          init_constraint
18              position[t1] = 0 /\
19              position[t2] = 3 /\
20              occupied[0] = true /\
21              occupied[1] = false /\
22              occupied[2] = false /\
23              occupied[3] = true /\
24              occupied[4] = false
```

Listing 10: Init constraint, extract from Listing 7.

3.2.4 Unfolding and translation

Both Listing 1 and Listing 7 are RSL* specifications – one *generic* and the other *concrete* or “configured”. However, even though the *concrete* specification has converted values to be explicit and variables an init constraint, and the under-specified types have been specified, it is still in the *generic* subset of RSL*, and RT-Tester cannot use it as input to model check.

RSL*$_{\text{subset}}$ is the subset of RSL* that is translatable to RTT, the input language to RT-Tester. The subset is RSL* without *generics*, i.e. all *generic* constants and variables have been translated to a non *generic* version. In other words, everything in RSL can be expressed in RSL*$_{\text{subset}}$, and everything in RSL*$_{\text{subset}}$ can be expressed in RSL*, but not the other way around. Thus, the language set can be depicted as in Figure 5. Not all RSL* constructs can be unfolded to RSL*$_{\text{subset}}$ or translated to RTT.

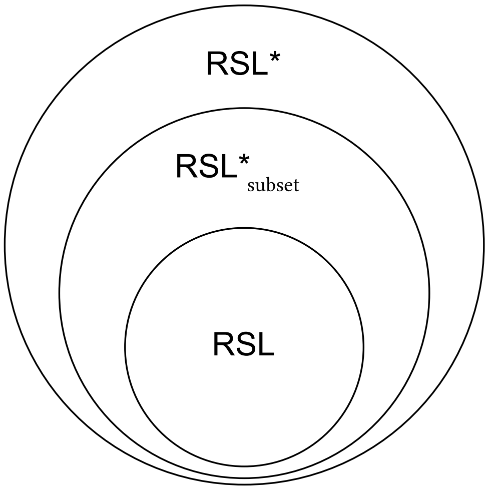

Figure 5: Depiction of RSL and RSL* language relation

Unfolding is transpiling RSL* to $\text{RSL*}_{\text{subset}}$ and is done by invoking `rsltc` with the `-unfrtt` option and an input file. For example, `$ rsltc -unfrtt SimpleRail.rsl` will unfold `SimplerRail.rsl` and output the unfolding specification to `SimpleRail_unfolded.rsl`. The unfolding process is briefly introduced in Section 3.3 and elaborated in Section 4.4.

The unfolded specification can be translated to RTT using the `-rtt` option. For example, `$ rsltc -rtt SimpleRail_unfolded.rsl` will output the RTT file to `SimpleRail_unfolded.rtt`, which can be used as input to the model checker in RT-Tester. The initial RT-Tester translation implementation in `rsltc` is done in [18] and modified by Signe Geisler during Geisler's research work.

Listing 11 is Listing 7 unfolded and the size of the unfolded specification is significantly larger and many things have been changed. The process of unfolding will be explained in further in Chapter 4. However, the changes from unfolding Listing 7 will be listed to give an idea of the unfolding process.

- References to the *generic* constant `max` has been replaced with the value from the axiom declaration; one instance is in the sub-type expression in line 6.
- The axiom declaration has been removed.
- The *generic* variable definitions are *concrete*, i.e., the square brackets' content has been appended using `_` as a delimiter. This can be seen in the variable declaration lines 14 to 20 and the init constraint declaration lines 22 to 28. For example, `position[t1]` is unfolded to `position_t1`.
- The quantified expression in the transition rules has been unfolded and significantly contributes to the size. Lines 29 to 926 are the unfolded transition rule, which has been truncated in Listing 11. The full version is in Appendix G. A lot is going on. Firstly, the references to the named transition rules have been replaced with the rule, which has been unfolded. The rules are two quantified expressions. A quantified expression is unfolded by considering the combination of all definitions represented by the typing list, replacing the reference to the typing with the instance, and combining each value expression according to the quantifier. In this example, the quantifier is `[=]`, the non-deterministic choice, and combines each unfolded value expression. Notice how many of the unfolded guarded expressions in the transition rules are "dead", i.e. they have a guard which statically can be evaluated as false. It can be determined that the effect is never executed.
- LTL assertions are also unfolded. Like transition rules, the quantified expressions are unfolded. Here, the `all` (forall/∀) quantifier is used, resulting in the unfolded being a conjunction of value expressions.

```
scheme SimpleRail_unfolded =
class
type
TrainId == t1 | t2,
SegmentId = {| n : Int :- n >= 0 /\
  n < 5 |}

value
max : Int = 5

transition_system
[TS]
variable
position_t1 : SegmentId,
position_t2 : SegmentId,
occupied_0 : Bool,
occupied_1 : Bool,
occupied_2 : Bool,
occupied_3 : Bool,
occupied_4 : Bool
init_constraint
position_t1 = 0 /\
position_t2 = 3 /\
occupied_0 = true /\
occupied_1 = false /\
occupied_2 = false /\
occupied_3 = true /\
occupied_4 = false
transition_rules
  position_t1 > 0 /\
  (position_t1 = 0 /\
  ((0 - 1) = 0 /\
  ~occupied_0))
  ==>
  position_t1' = position_t1 - 1,
  occupied_0' = false,
  occupied_0' = true
[=]
  position_t1 > 0 /\
  (position_t1 = 0 /\
  ((0 - 1) = 1 /\
  ~occupied_1))
  ==>
  position_t1' = position_t1 - 1,
  occupied_0' = false,
  occupied_1' = true
[=]
...
 [=]
  position_t2 > 0 /\
  (position_t2 = 4 /\
  ((4 - 1) = 4 /\
  ~occupied_4))
  ==>
  position_t2' = position_t2 - 1,
```

```
  occupied_4' = false,
  occupied_4' = true
[=]
  position_t1 < (max - 1) /\
  (position_t1 = 0 /\
  ((0 + 1) = 0 /\
  ~occupied_0))
  ==>
  position_t1' = position_t1 + 1,
  occupied_0' = false,
  occupied_0' = true
[=]
...
 [=]
  position_t2 < (max - 1) /\
  (position_t2 = 4 /\
  ((4 + 1) = 4 /\
  ~occupied_4))
  ==>
  position_t2' = position_t2 + 1,
  occupied_4' = false,
  occupied_4' = true
end

ltl_assertion
[one_train_per_section] TS |- G((t1 ~= t1 =>
  position_t1 ~= position_t1) /\
  ((t1 ~= t2 =>
  position_t1 ~= position_t2) /\
  ((t1 ~= t1 =>
  position_t1 ~= position_t1) /\
  (t2 ~= t2 =>
  position_t2 ~= position_t2)))),
[occupied_correct] TS |- G((position_t1 = 0 =>
  occupied_0) /\
  ((position_t1 = 1 =>
  occupied_1) /\
  ((position_t1 = 2 =>
  occupied_2) /\
  ((position_t1 = 3 =>
  occupied_3) /\
  ((position_t1 = 4 =>
  occupied_4) /\
  ((position_t2 = 0 =>
  occupied_0) /\
  ((position_t2 = 1 =>
  occupied_1) /\
  ((position_t2 = 2 =>
  occupied_2) /\
  ((position_t2 = 3 =>
  occupied_3) /\
  (position_t2 = 4 =>
  occupied_4))))))))))
end
```

Listing 11: Unfolded `SimpleRail.rsl` with truncated transition rules (`SimpleRail_unfolded.rsl`, full version in Appendix G).

3.3 RSL* additions

This section will briefly walk through the additions in RSL* in relation to RSL. It will not introduce the full grammar supported by `rslts`, but the full grammar supported by `rslts` is defined in Section 4.2. Neither will it give a comprehensive overview of unfolding, which will be defined in Section 4.4.

3.3.1 Array

The array is an addition to RSL* that results in an addition to type and value expressions.

`array <index_type> of <value_type>` is the array type expression, an addition to type expressions. For example, `ArrayType = array IndexType of ValueType` will yield a type named `ArrayType` and be an array of `ValueType` indexed by `IndexType`.

Accessing an array is a new value expression on the form: `array_value_expr[<value_expr>]`. It will access the element of `array_value_expr` corresponding to the value indexed by `<value_expr>` and must have `IndexType` as its type. The value expression `array_value_expr[<value_expr>]` will have the type of the array, here `valueType`.

Initialising an array is done by `{. value_0, ..., value_n .}`, another addition to the value expressions. It denotes an array with elements `value_0, ..., value_n`.

```
scheme ArrayExample =
  class
    type
      ArrayType = array IndexType of ValueType,
      IndexType = {| i : Int :- i >= 0 /\ i <= 10 |},
      ValueType = Int
    value
      arrayValue : ArrayType = {. 1, 2, 3, 4, 5 .},
      getArrayValue: ArrayType >< IndexType -> ValueType
      getArrayValue(arr, i) is
        arr[i]
  end
```

Listing 12: Example showing array addition (`ArrayExample.rsl`).

Listing 12 is valid RSL*, gibberish, however, and is an example of array usage. An array's index type must follow `{| i : Int :- i >= 0 /\ i < <upper_bound> |}` as the structure for the sub-type expression, where `<upper_bound>` is any integer above 0. In this case, `i` can be an `Int` or `Nat`; the identifier can be any valid identifier.

3.3.2 *Generic* constants

Value declaration has been extended with the option to create *generic* value definitions, such as `someValue [ i : IndexType ] : Nat` and accessing these as accessing an array. *Generic* value can be over a list of types. They can be used to create *generic* constants. The type used must be specified and finite to be unfoldable and when unfolded all instances are explicit value definitions.

```
scheme GenericValueDeclarationExample =
  class
    type
      IndexType1 == t1 | t2 | t3,
```

```
      IndexType2 = {| i : Int :- i >= 0 /\ i < max |},
      ValueType
    value
      max : Int,
      genericValue1 [ i : IndexType1 ] : ValueType,
      genericValue2 [ i1 : IndexType1, i2 : IndexType2 ] : ValueType
    axiom
      genericValue1[t1] = ...,
      genericValue2[t1, 0] = ...,
  end
```

Listing 13: Example showing *generic* value declaration when creating generic constants (`GenericValueDeclarationExample.rsl`).

Listing 13 is an example of *generic* value defintions when creating *generic* constants. The same format is used when creating *generic* variable definitions, but contradictory to unfolding *generic* value definitions, the *generic* variable definitions remain implicit and their initialisation clause is in the init-constraints declaration.

3.3.3 Transition Systems

A transition system consists of three declarations: (1) variables, (2) init constraints and (3) transition rules.

(1) Variables

The variable declaration format is the same as in RSL and has been extended to support *generic* variable definitions. The same format from *generic* constants is used when creating *generic* variable definitions, but contradictory to unfolding *generic* value definitions, the *generic* variable definitions remain implicit and their initialisation clause is in the init-constraints declaration.

```
        variable
            position [ t : TrainId ] : SegmentId,
            occupied [ s : SegmentId ] : Bool
```

Listing 14: Transition system variable declaration, extract from Listing 1.

(2) Init constraints

Init constraint declarations are used to give an initial value to each variable. This is a list of infix expressions or quantified expressions. The infix expressions must be with a name or *generic* name value expression on the lhs, and any value expression on the rhs, which must have the same type as the lhs name has been declared with. The quantified expression must use `all` (∀) as a quantifier, and the quantified value expression must be an infix expression following the previously listed requirements. As seen in Listing 15.

```
        init_constraint
            position[t1] = 0 /\
            position[t2] = 3 /\
```

Listing 15: Init constraint, extract from Listing 7.

(3) Transition rules

The transition rules declaration consists of two parts: (1) transition rules and (2) a named transition rules section; naming rules make it easier to divide transition rules into parts and help get an overview when creating a specification.

(1) Transition rules can be combined using the non-deterministic `[=]` and prioritised choice `[<]` operators. A single transition rule is a guarded value expression based on the guarded command language structure, as seen in lines 32 to 37 in Listing 16. Both operators can also be used in a quantified expression. The guarded command uses ==> as an infix operator, and the lhs is a boolean expression. The rhs is a prime update expression, i.e. an infix expression with the = (equal) operator where the lhs must be a primed access expression and the rhs a value expression of the same type as the accessed value.

(2) The named transition rules section indicated with the keyword `WHERE` is a list of named rules, with a name and a transition rule, as seen in Listing 7.

```
32                              position[t] < (max - 1) /\
33                              position[t] = s1 /\
34                              (s1 + 1) = s2 /\
35                              ~occupied[s2] ==>
36                                  position'[t] = position[t] + 1,
37                                  occupied'[s1] = false,
```

Listing 16: Guarded value expression, extract from Listing 7.

3.3.4 LTL assertions

The LTL assertions declaration is aiding analysis tools in understanding the transition system and is a list of named LTL assertions each bound to a transitions system and uses temporal operators. Temporal model operators are also an addition to value expressions that can be used in LTL assertions, such as **G**lobally ($\square$) and **F**inally ($\diamond$).

Consider Listing 17 as the format for LTL assertions and Listing 18 is LTL assertions from lines 51 to 57 in Listing 7.

```
ltl_assertion
    [ <name> ] <transition_system_name> |- <value_expr>,
    [ <name> ] <transition_system_name> |- <value_expr>
```

Listing 17: LTL assertion format.

```
51    ltl_assertion
52        [one_train_per_section] TS |-
53          G(all t1: TrainId, t2: TrainId :-
54            t1 ~= t2 => position[t1] ~= position[t2]),
55        [occupied_correct] TS |-
56          G(all t: TrainId, s: SegmentId :-
57            position[t] = s => occupied[s])
```

Listing 18: LTL assertions, extract from Listing 7.