Introduction to Remopla

Jan Holeček
Dejvuth Suwimonteerabuth
Stefan Schwoon
Javier Esparza

August 2006

Contents

1 Model Checking Models in Remopla

The Remopla language, an input language of the Moped model checker [?], is aimed at modeling behaviours of finite-domain programs with (possibly) unbounded recursive procedure calls. The model checker implements reachability analysis and general LTL model checking over models in Remopla.

This document gives an informal semantics of Remopla statements in terms of execution paths, i.e. sequences of model configurations, because execution paths and runs of models are of concern to the model checker. A model in Remopla can contain both finite and infinite runs. The way the model checker understands them depends on the task it is carrying out.

The finiteness of runs does not really matter concerning reachability analysis. A path leading to any reachable model configuration is always finite and vice versa. One just has to be careful not to terminate execution paths by mistake.

On the other hand, the semantics of LTL is defined over infinite runs only. By default, the model checker ignores finite runs during verification of LTL properties for reasons discussed below. However, the model checker also offers a mode in which finite runs are turned into infinite ones by appending an infinite loop to them.

With LTL model checking in mind, models in Remopla are usually designed to have infinite runs. Even models of finite computation processes usually contain explicit infinite loops to which control flow moves after the computation.

Finite runs are usually introduced into a model by Remopla statements the semantics of which include termination of particular execution paths in the sense that the statements do not define the next configuration for a given one. The termination is either explicit or implicit. The explicit termination is used to cut off unwanted execution paths based on a user-defined condition. The implicit termination cuts off execution paths on which a "run-time error" occurs. The errors namely include division by zero or value out of range.

Expectedly, implicitly terminated paths start with invalid input data so it is perfectly safe to terminate them and ignore them for the model checking. However, it is quite easy to cut off all execution paths by mistake which leads to meaningless results of model checking. That is why the model checker offers the second mode in which finite runs are not ignored. We pay extra attention to the rules for implicit termination of execution paths throughout this document.

For an example of implicit termination, consider sample code 1. The local variable `n` of module `main` is declared to be a 4-bit unsigned integer so its value can range from 0 to 15. All execution paths of the model (which differ only in the initial value of the variable `n`) start with a call to the module `main` and try to carry out the assignment at line 5. However, the assignment cannot proceed because the value is out of range of the variable and the model checker aborts the execution paths. Because there is no execution path left then, the analysis never gets to line 7.

```
1   init main;
2
3   module void main () {
4       int n(4);
5       n = 16;
6       # the next line is not reachable
7       not_reachable:  goto not_reachable;
8   }
```

Sample Code 1: Termination of Execution Paths (`termination.rem`)

2 Notation

This document gives a terse description of the syntax and semantics of the language by explaining its grammar in Backus-Naur form. The description is accompanied with sample codes that demonstrate the language concepts. These sample codes also included in the Moped distribution.

The notation we will use for the grammar is as follows. *Terminals* are set in typewriter typeface. *Non-terminals* are set in roman typeface and are enclosed in $\langle\,\rangle$. The notation also includes standard operators of BNF as listed below.

x^*	for zero ore more repetitions of x
x^+	for one or more repetitions of x
x`,`*	for zero or more comma-separated instances of x
x`,`$^+$	for one or more comma-separated instances of x
$x \mid y$	a choice of x or y
[x]	optional (zero or one instance of) x

3 Language Overview

The Remopla language is similar to other modern programming languages. A model in Remopla consists of definitions of constants, declarations of data types and variables, module definitions and statements.

Constants, types, variables, modules, labels of statements and elements of enumerated data types are named by *identifiers*. An identifier is any non-empty string of alphanumeric characters and underscores starting with a letter. The meta-rules use a regular expression syntax that represent lexical elements of the language: identifiers and positive integers.

⟨identifier⟩ $\rightarrow$ `[a-zA-Z][a-zA-Z0-9_]`*
⟨literal number⟩ $\rightarrow$ `[0-9]`$^+$

Some identifiers are reserved as keywords. Here is the list of all keywords in Remopla.

Keywords for data types:
    `bool`, `int`, `struct`, `enum`, `void`
Keywords for special values and constants:
    `undef`, `false`, `true`, `DEFAULT_INT_BITS`
Keywords for statements:
    `skip`, `if`, `fi`, `do`, `od`, `else`, `break`, `goto`, `return`
Other keywords:
    `A`, `E`, `define`, `init`, `module`

The rules governing name spaces of identifiers are rather complex. To stay on the safe side, one should not use a single identifier to denote two different entities. For instance, using an identifier for a constant and anything else would produce a syntax error. Using an identifier as a module name and as a label would work just fine but the semantics would differ from what one might have expected.

Remopla is a *case sensitive* language. The case matters for both keywords and identifiers.

Comments in Remopla are line-oriented. Everything in a line starting from the character `#` or the sequence of characters `//` is ignored up to the end of the line by the parser as a comment.

4 Model Structure

Every model in Remopla has to follow the structure given in the production rule of ⟨model⟩. A simple example of the structure is given in sample code 2.

⟨model⟩ $\rightarrow$ ⟨constant definition⟩*
    ⟨global declaration⟩*
    ⟨initial configuration⟩
    ⟨module or statement⟩$^+$
⟨global declaration⟩ $\rightarrow$ ⟨enum declaration⟩ | ⟨struct declaration⟩
    | ⟨variable declaration⟩
    | ⟨module declaration⟩
⟨module or statement⟩ $\rightarrow$ ⟨statement⟩ | ⟨module definition⟩

Constant definitions can only appear at the very beginning of a Remopla code. They are not allowed anywhere else. See section 5 for details regarding constant definitions.

Global declarations stand for type declarations, module declarations and declarations of global variables. Again, these declarations cannot be placed anywhere else. Type declarations and declarations of variables are covered in section 6. See section 10 for more information on module declarations.

Initial configuration can be thought of as the last global declaration—it is mandatory for every model and it separates the section of global declarations from the model body. The declaration specifies the initial configuration of the model. The details are given in section 7.

Model body consists of statements and module definitions implementing the behaviour of the model. The statements are usually (but not necessarily) organized in modules. Stand-alone statements are commonly used for infinite labeled loops that explicitly make finite execution paths infinite. Line 12 of sample code 2 demonstrates such a loop.

```
 1   # constant definitions
 2   define DEFAULT_INT_BITS 4
 3
 4   # global declarations
 5   module void main();
 6
 7   # initial configuration
 8   init main;
 9
10   # model body
11
12   ok: goto ok;
13
14   module void main () {
15       goto ok;
16   }
```

Sample Code 2: The Simplest Model (`simplest.rem`)

5 Constant Definitions

A constant in Remopla is a symbolic name for a non-negative integer. The primary use of constants is to parametrize models in the sense that a Remopla code can implement many instances of a problem depending on the values of constants. All constants must be defined at the very beginning of the model. The syntax of the constant definition is governed by the following rule.

⟨constant definition⟩ → `define` ⟨constant⟩ ⟨cnst expr⟩
⟨constant⟩ → ⟨identifier⟩

The second argument of the definition must be a constant integer expression (see section 8.2), i.e. an integer expression that combines numbers and previously defined constants only. A constant definition is lexically terminated either by the next constant definition or by any global declaration including the declaration of the initial configuration.

5.1 Internal Constants

Internal constants parametrize the model checking process. Regarding syntax, internal constants are keywords which can be used in the ⟨constant definition⟩ rule in place of ⟨constant⟩. They cannot be used anywhere else.

There is only one internal constant at the moment but more of them may be introduced in future versions. Currently, the only internal constant is `DEFAULT_INT_BITS` and it does exactly what the name suggests: it sets the default bit size of integers (see section 6.3). For instance, the default bit size of integers in sample code 2 would be four. The default bit size of variables and return values of modules can be overridden in declarations.

6 Data Types and Variable Declarations

This section concerns declarations of data types and variables. Formal parameters and return values of modules are treated in section 10.

The Remopla language supports boolean data type, unsigned integers of a limited range, enumerated data types and structured data types. A variable of any of these types can be declared as a simple variable, one-dimensional array or two-dimensional array. The syntax of regular variable declarations is given by the production rule of ⟨variable declaration⟩. Variables of enumerated and structured data types can also be declared within the type declarations (see subsections 6.4 and 6.5).

⟨variable declaration⟩ → `bool` ⟨variable spec⟩`,`$^+$ `;`
  | `int` ⟨int variable spec⟩`,`$^+$ `;`
  | `enum` ⟨type name⟩ ⟨variable spec⟩`,`$^+$ `;`
  | `struct` ⟨type name⟩ ⟨variable spec⟩`,`$^+$ `;`
⟨variable spec⟩ → ⟨var name⟩ *[* `[`⟨dimen⟩`]` *]* *[* `[`⟨dimen⟩`]` *]*
⟨int variable spec⟩ → ⟨variable spec⟩ *[* `(`⟨cnst expr⟩`)` *]*
⟨dimen⟩ → ⟨cnst expr⟩ | ⟨cnst expr⟩`,`⟨cnst expr⟩
⟨type name⟩ → ⟨identifier⟩
⟨var name⟩ → ⟨identifier⟩

The ⟨type name⟩ is an identifier which refers to a previously declared enumerated or structured data type.

The only mandatory part of a variable specification is the identifier ⟨var name⟩ which names the variable. The name of an array variable is followed by bracketed specification of array dimensions. Both simple and array integer variables can also be followed by parenthesized bit size of the variable and elements of the array, respectively, which overrides the default bit size from `DEFAULT_INT_BITS`.

Access to variables is denoted by ⟨variable⟩ in the grammar rules. The usage of a variable depends on its declaration and is summarized by the following production rule. The first part of the rule gives syntax for common variables, the second part gives syntax for inner variables of structures. The optional ⟨array idx⟩ gives syntax for referencing elements of one- and two-dimensional arrays.

⟨variable⟩ → ⟨var name⟩ *[*⟨array idx⟩*]*
  | ⟨var name⟩ *[*⟨array idx⟩*]* `.` ⟨var name⟩ *[*⟨array idx⟩*]*
⟨array idx⟩ → `[`⟨int expr⟩`]` *[*`[`⟨int expr⟩`]`*]*
⟨var name⟩ → ⟨identifier⟩

6.1 Arrays

One-dimensional and two-dimensional array variables are declared by adding one and two dimension specifications to the variable identifier, respectively. A dimension specification has two forms.

In the first form, the dimension specification consists of a single constant integer expression. Elements of the array in the respective dimension are indexed from 0 to $n - 1$ where n is the value of the expression.

In the second form, the dimension specification consists of two comma-separated constant integer expressions where the first expression must be lower than or equal to the second one. Elements of the array in the respective dimension are indexed from p to q where p and q are the values of the first and the second expression, respectively.

Note that arrays are strictly limited to the dimension of two. It is not possible to acquire arrays of more dimensions by constructing arrays of arrays.

6.2 Booleans

The value of a boolean variable can be either `true` or `false`. Boolean variables and the two keywords form the atomic boolean expressions from which compound ones can be built up (see section 8.3). For sample declarations and usage, consider the following lines.

```
bool cnd;
bool b1[0,3], b2[4];
# sample usage
cnd = cnd || (b1[0] && b2[3]);
```

Arrays `b1` and `b2` have the same range of indices. They both have four boolean elements indexed from 0 to 3 in a single dimension.

6.3 Integers

Integer variables in Remopla can store unsigned integer values of a limited range. The values are stored as bit arrays where the least significant bit is the rightmost one. Every integer variable must be given its bit size, i.e. the length of the bit array storing its value. Assuming that the bit size of a variable is n, the value of the variable can range from 0 to $2^n - 1$.

Internal constant `DEFAULT_INT_BITS` (see section 5.1) can be set to specify the default bit size of integer variables. The default value can be overridden by optional parenthesized constant integer expressions specifying the bit sizes for individual variables in variable declarations. Note, that it is an error if neither the default bit size is given nor is the bit size specified explicitly for each integer variable.

For instance, consider the next few lines of Remopla code. Variable `area` has bit size 8, both of variables `width` and `height` have bit size 4. Variable `m1` is a 5×2 array with elements indexed from 0 to 4 in one dimension and from 2 to 3 in the other. The elements are 4-bit integers. Variable `m2` is a one-dimensional array with ten elements indexed from 0 to 9. The elements are 3-bit integers.

```
define DEFAULT_INT_BITS 4
int area(8), width, height;
int m1[0,4][2,3], m2[10](3);
# sample usage
area = width*height;
m1[1][2] = 2*m2[9];
```

6.4 Enumerations

In general, an enumeration is a finite sequence of identifiers. The syntax of an enumerated data type declaration is described by the following rules.

⟨enum declaration⟩ → `enum` *[*⟨type name⟩*]* `{`
⟨enum element⟩`,`$^+$
`}` ⟨variable spec⟩`,`* `;`
⟨enum element⟩ → ⟨identifier⟩
⟨type name⟩ → ⟨identifier⟩
⟨variable spec⟩ → ⟨var name⟩ *[* `[`⟨dimen⟩`]` *]* *[* `[`⟨dimen⟩`]` *]*
⟨var name⟩ → ⟨identifier⟩

The optional element ⟨type name⟩ is an identifier denoting the declared type. Naming the new type makes it possible to declare local module variables, module parameters and modules with return values of that type.

Declarations of global variables can be included directly into the type declaration.

The following declaration gives an example of an enumerated data type. Because the enumerated type is not given a name, one would have to exploit the implementation to declare more variables of the same data type.

```
enum {
    s_start, s_accept, s_deny
} state;
# sample usage
state = s_start;
```

The implementation of enumerated data types is based on integers. Elements of an enumerated set are numbered from zero and behave exactly the same as constants. It follows that a single identifier cannot name elements of two enumerated types. The identifier is bound to an integer at the first occurrence and the second occurrence would produce a syntax error. It also follows that enumerated types of the same number of elements are equivalent. The elements from the above example are equivalent to constant definitions as follows.

```
define s_start  0
define s_accept 1
define s_deny   2
```

However, the constant definitions does not allow to declare any equivalent to the variable `state`. The behaviour of an uninitialized common 2-bit integer variable differs from that of the variable `state`. The former can take value 3 whereas the latter cannot. See also the special `undef` value in section 9.1.

6.5 Structures

Simple variables and arrays of integers and booleans can form a single data type—a structured data type. The declaration syntax of a new structured data type is governed by the following rules. Similarly to enumerated data types, ⟨type name⟩ is optional and variable declarations can be embedded into the type declaration.

⟨struct declaration⟩ → `struct` *[*⟨type name⟩*]* `{`
                ⟨struct member⟩$^+$
        `}` ⟨variable spec⟩`,`* `;`
⟨struct member⟩ → `bool` ⟨variable spec⟩`,`$^+$ `;`
    | `int` ⟨int variable spec⟩$^+$ `;`
⟨type name⟩ → ⟨identifier⟩

⟨variable spec⟩ → ⟨var name⟩ *[* `[`⟨dimen⟩`]` *]* *[* `[`⟨dimen⟩`]` *]*
⟨int variable spec⟩ → ⟨variable spec⟩ *[* `(`⟨cnst expr⟩`)` *]*
⟨var name⟩ → ⟨identifier⟩

Note that it is not possible to use structured data type to acquire more than two-dimensional arrays. If a structured data type contains an array, it is not allowed to declare an array with elements of that type. If an array with elements of a structured data type is required, the structure must contain no arrays.

The main use of variables of structured data types is to make passing local data to and from modules easier (see section 10). Variables of structured data types cannot be used anywhere else. The inner parts of structured variables can be used as ordinary variables.

```
define DEFAULT_INT_BITS 4

struct rectangle {
    int width, height, area(8);
    bool side_property[4];
};
struct rectangle rct;
# sample usage
rct.width = 3;
rct.height = 4;
rct.side_property[3] = true;
```

The sample code above declares a new structured data type named `rectangle`. It also declares structured variable `rct` and shows the way the inner parts of the structured variable can be accessed.

7 Initial Configuration

Global declarations in a model are always terminated by a mandatory declaration of initial configuration the syntax of which is given by the following rule.

⟨initial configuration⟩ → `init` ⟨module name⟩`;` | `init` ⟨label⟩`;`
⟨module name⟩ → ⟨identifier⟩
⟨label⟩ → ⟨identifier⟩

The declaration determines a point in the model from which the model execution paths start. For a label, they simply start with the labeled statement (see section 9). For a module name, all execution paths start by a call to the module (see section 10.3).

Initial values of variables are not specified by this declaration. It effectively means that the declaration specifies a set of initial configurations, one for each combination of values that fit into the ranges of individual variables in the lexical scope of the starting point. An execution path of the model can start from any of the initial configurations.

The lexical scope of a labeled statement outside a module includes global variables only. The lexical scope of a module name includes global variables and module parameters; local variables of the module are non-initialized by the semantics of a module call (see section 10.3). Finally, the lexical scope of a labeled statement inside a module includes global variables, module parameters and local variables of the module. However, one should avoid this option.

For example, consider sample code 3 on page 13. The array is not initialized. The label *labA* is reachable because `a[0]` can be non-zero. Because it can also be zero, the label *labB* is reachable, too.

8 Expressions

The Remopla language distinguishes three types of expressions: constant integer expressions, general integer expressions and boolean expressions. The syntax of all the three kinds of expressions is defined by the following rules. The following subsections treat details regarding each one of them.

```
⟨cnst expr⟩ → ⟨literal number⟩ | ⟨constant⟩
      | ⟨cnst expr⟩ ⟨integer binop⟩ ⟨cnst expr⟩
      | ( ⟨cnst expr⟩ )
⟨int expr⟩ → ⟨literal number⟩ | ⟨constant⟩ | ⟨variable⟩
      | ⟨int expr⟩ ⟨integer binop⟩ ⟨int expr⟩
      | ( ⟨int expr⟩ )
⟨bool expr⟩ → true | false | ⟨variable⟩
      | ⟨int expr⟩ ⟨integer binrel⟩ ⟨int expr⟩
      | ! ⟨bool expr⟩
      | A ⟨var name⟩ (⟨cnst expr⟩,⟨cnst expr⟩) ⟨bool expr⟩
      | E ⟨var name⟩ (⟨cnst expr⟩,⟨cnst expr⟩) ⟨bool expr⟩
      | ⟨bool expr⟩ ⟨boolean binop⟩ ⟨bool expr⟩
      | ( ⟨bool expr⟩ )
⟨integer binop⟩ → * | / | + | - | << | >> | & | | | ^
⟨integer binrel⟩ → < | <= | == | != | >= | >
⟨boolean binop⟩ → && | || | ^^ | <=>
⟨constant⟩ → ⟨identifier⟩
⟨variable⟩ → ⟨var name⟩ [⟨array idx⟩]
      | ⟨var name⟩ [⟨array idx⟩] . ⟨var name⟩ [⟨array idx⟩]
⟨array idx⟩ → [⟨int expr⟩] [[⟨int expr⟩]]
⟨var name⟩ → ⟨identifier⟩
```

The integer operators are multiplication, division, addition, subtraction, shift left, shift right, bitwise conjunction, bitwise disjunction and exclusive bitwise disjunction, respectively. The boolean operators are conjunction, disjunction, exclusive disjunction and logical equivalence, respectively.

8.1 Constant Integer Expressions

Constant integer expressions just provide syntactic sugar—a more general way how to write numbers. These expressions are evaluated at parse time Because of this, division by zero in a constant integer expression is reported as a syntax error by the parser and so is any negative value of an expression.

8.2 General Integer Expressions

The syntax of general integer expressions of Remopla is very similar to common programming languages but the semantics differs. Models in Remopla do not raise errors when an integer expression cannot be evaluated for some reason. The behaviour in these cases depends on the context of the expression.

In general, the evaluation of a general integer expression is best viewed as a relation which relates a combination of values of variables present in the expression to a result of the expression—an unlimited integer. It is the context of the expression which may pose finite bounds on the result. If the expression does not evaluate to an integer, the corresponding input values relate to nothing.

The cases in which an expression does not evaluate to an integer namely include division by zero and an out of range index of an array element at any level of nesting—the index of an array element can be another integer expression involving a reference to another array element etc.

Execution paths in which an integer expression cannot be evaluated are terminated in most cases. The only exception is an integer expressions as a part of an atomic boolean expression—the boolean expression is considered false if an integer expression it contains cannot be evaluated. See subsection 8.3 for more details.

The statements that implicitly terminate execution paths in which an integer expression cannot be evaluated include assignments, return statements returning non-boolean value and module calls with non-boolean parameters (see sections 9.1, 10.2 and 10.3).

8.3 Boolean Expressions

The primary use of boolean expressions in Remopla is to guard clauses of conditionals and loops (see section 9.5). Of course, the expressions can also be used in assignments to boolean variables (see sections 6.2 and 9.1).

The syntax of boolean expressions is similar to common programming languages. Standard binary logical connectives and the negation operator (the exclamation mark `!`) are available. Moreover, there are universal quantifier `A` and existential quantifier `E` which require more explanation.

Both quantifiers declare an identifier as a read-only integer variable and its range that can be used inside the quantified expression only. A universally quantified expression holds if and only if the inner expression holds for every value of the quantified variable. An existentially quantified expression holds if and only if the inner expression holds for at least one value of the quantified variable.

The quantified expressions can be very handy when it comes to tests on arrays. Sample code 3 demonstrates usage of quantified expressions. The two boolean expressions in the example are equivalent: they check if there is a non-zero element in the array `a`. Because the array is not initialized, both labels are reachable.

```
int a[4](4);

init main;

module void main () {
    if
    :: ! A i (0,3) a[[]i] == 0 -> goto labA;
    ::   E i (0,3) a[[]i] != 0 -> goto labA;
    :: else  -> goto labB;
    fi;
}

labA: goto labA;
labB: goto labB;
```

Sample Code 3: Quantified Boolean Expressions (`quant-bool.rem`)

A boolean expression is always evaluated either to true or false because it is so for all atomic expressions. If an atomic expression (a boolean variable or an integer comparison) involves an integer expression that cannot be evaluated or an out of range index to an array element, the atomic expression is considered false. See also section 8.2.

Note that there are no bounds on the values being compared in an integer comparison. Again, the comparison is best seen as a relation on involved variables. The comparison yields true if the current values of variables belong to the relation and false otherwise. Size of no subexpressions but array indices matter. For instance, `-1==-1` yields true whereas `-1==-2` yields false.

9 Statements

The following rules list all types of statements in Remopla. We are not going to a give formal semantics of the statements. Instead, we will give an informal description of the effects of the statements on execution paths of a model. Recall the discussion of implicit termination of execution paths in section 1 and evaluation of integer and boolean expressions in section 8.

⟨statement⟩ → *[*⟨label⟩`:`*]* ⟨real statement⟩
⟨real statement⟩ → ⟨assignments⟩
  | ⟨skip statement⟩
  | ⟨goto statement⟩
  | ⟨break statement⟩
  | ⟨if statement⟩
  | ⟨do statement⟩
  | ⟨return statement⟩
  | ⟨call statement⟩
⟨label⟩ → ⟨identifier⟩

Any statement can be labeled with an identifier. Labels must be unique across the model. Labeling a statement makes it possible to bring control flow to that statement by a go-to statement and the declaration of initial configurations (see sections 9.3 and 7, respectively) and to query reachability of the label. Labels also form a subset of atomic propositions of LTL formulae over the model at hand.

9.1 Assignments

⟨assignments⟩ → ⟨assignment⟩`,`$^+$`;`
⟨assignment⟩ → *[*⟨quantify⟩*]* ⟨variable⟩ `=` `undef`
  | ⟨variable⟩ `=` ⟨variable⟩
  | *[*⟨quantify⟩*]* ⟨variable⟩ `=` ⟨int expr⟩
  | *[*⟨quantify⟩*]* ⟨variable⟩ `=` ⟨bool expr⟩
⟨quantify⟩ → `A` ⟨var name⟩ `(`⟨cnst expr⟩`,`⟨cnst expr⟩`)`
⟨variable⟩ → ⟨var name⟩ *[*⟨array idx⟩*]*
  | ⟨var name⟩ *[*⟨array idx⟩*]*`.`⟨var name⟩ *[*⟨array idx⟩*]*
⟨array idx⟩ → `[`⟨int expr⟩`]` *[*`[`⟨int expr⟩`]`*]*
⟨var name⟩ → ⟨identifier⟩

Assignments are usually at the core of any model. A variable of any given type receive the value of another variable of the same type and the same dimensions for an array variable. Simple integer and boolean variables can be assigned the result of an integer and boolean expression, respectively.

Multiple assignments can be put in parallel. A parallel assignment consisting of any number of individual assignments works as a single relation which relates current values of variables in expressions at the right-hand sides to new values of variables at the left-hand sides.

An assignment can be universally quantified using the `A` keyword. The quantification declares a read-only integer variable and its range. The variable can be used in the assignment. Such an assignment is equivalent to a parallel assignment which contains one instance of the assignment for each value of the quantified variable from the given range. It is very handy in operations with arrays. For instance, the following code initializes an array to zeros.

```
define ARRLNG 5
int arr[ARRLNG](2);
A i (0,ARRLNG-1) arr[i] = 0;
```

9.1.1 Unsetting Variables

The special `undef` value "unsets a variable", i.e. it gives the variable a non-deterministic value within its range, effectively making it non-initialized like before the first assignment to it. The assignment effectively splits each execution path into several ones, one for every possible value of the unset variable.

There are just two values for a boolean variable, `true` and `false`. The number of possible values of a simple integer variable depends on its bit size—all values from the range are considered. For a variable of an enumerated data type, there is one path for every element of the enumerated set. Concerning arrays and structures, unsetting is applied to all array elements and inner variables of the structure, respectively.

9.1.2 Implicit Termination of Execution Paths

There are three independent conditions under which execution paths are terminated by assignments.

Assignments implicitly terminate all execution paths in which the left-hand side of the assignment contains an integer expression or an out of range index of an array element at any level of nesting. Non-existent element of an array is not a valid variable for assignment.

Assignments to integer variables implicitly terminate all execution paths in which the integer expression on the right-hand side yields a value which is out of the variable range or if the expression does not evaluate at all (see section 8.2). Neither overflow nor underflow can happen.

A parallel assignment containing multiple assignments to the same variable terminates all execution paths except for those in which the variable receives the same value in all assignments. For instance, consider the following lines of Remopla code. The first line is equivalent to the simple assignment `n=1`. The two assignments in the second line never assign the same value to `n` so all execution paths are terminated.

```
n=1, n=undef;
n=1, n=2;
```

9.2 Skip Statement

⟨skip statement⟩ → `skip` *[*`(`⟨bool expr⟩`)`*]*`;`

The skip statement is an extended version of skip statements from common programming languages whose primary use is to fulfill syntax requirements of other statements. The skip statement in Remopla can do more.

The statement takes one argument—a parenthesized boolean expression. It continues to the next statement for exactly those execution paths in which the expression holds. It aborts the other paths. The argument may be omitted in which case true is assumed—no path is terminated.

For example, the label *lbl* cannot be reached by executing the first skip statement in the following line of Remopla code.

```
skip false; lbl: skip;
```

9.3 Go-To Statement

⟨goto statement⟩ → `goto` ⟨label⟩`;`

The go-to statement moves control flow to the given label. The statement always preserves global data and the depth of the call stack. However, the effects on the context on the top of the stack, and hence local data, vary depending on the usage.

Jumps are usually used within the same lexical scope, i.e. both the go-to statement and the label are outside modules or in a single module. The jump preserves the context including any local data then.

If a jump spans across more lexical scopes, the current context (that of the go-to statement) is replaced by a new one fitting the jump destination. Local variables of the new context are non-initialized after the jump which effectively means all possibilities are examined. However, this practice is not recommended except for jumps to infinite go-to loops (see sample code 2).

9.4 Break Statement

⟨break statement⟩ → `break;`

Outside of a loop or a conditional, the break statement does nothing. Within a clause of a loop or a conditional, it terminates the innermost loop or conditional and brings control to the lexically next statement after the terminated one.

9.5 Conditionals and Loops

```
⟨if statement⟩ → if
                   ⟨guarded clause⟩+
                   [:: else -> ⟨statement⟩+]
                   fi;
⟨do statement⟩ → do
                   ⟨guarded clause⟩+
                   [:: else -> ⟨statement⟩+]
                   od;
⟨guarded clause⟩ → :: ⟨bool expr⟩ -> ⟨statement⟩+
```

Conditional and loop statements of Remopla have much in common which is why this single section concerns them both. Both conditionals and loops consist of guarded clauses of statements where guards are boolean expressions (see section 8.3).

When control arrives at the statement, all guards are evaluated first. The clause that gets control is then chosen non-deterministically among those clauses whose guards hold. This effectively means there is an execution path for each of them. If no guard holds and there is an `else`-clause, control is passed to it. If the `else`-clause is missing, the statement implicitly terminates all execution paths that do not satisfy any guard.

The `if`-statement is evaluated only once. The `do`-statement is evaluated repeatedly until `break` or `goto` occurs (see sections 9.3 and 9.4, respectively). The two statements can terminate the `if`-statement, too.

The semantics of loops and conditionals has several implications on the implementation of similar statements from common programming languages. For instance, the label *lbl* is not reachable in the following Remopla code if `a>=b`.

```
if
:: a<b  -> a=b, b=a;
fi;
lbl: skip;
```

This was probably unintended. The `else`-clause is required to fix it as in the following code.

```
if
:: a<b  -> a=b, b=a;
:: else -> break;
fi;
lbl: skip;
```

Let us present one more example to demonstrate the non-deterministic choice among satisfied guards. The following `do`-statement will loop until the execution of the last clause takes place. The value of `i` can be any of `1`, `2` and `3` then.

```
i = 1;
do
:: true -> i = 2;
:: true -> i = 3;
:: true -> break;
od;
```

10 Modules

Modules implement the concept of functions and procedures in Remopla. A module can take arguments and return a value. A module call can also be queried for reachability.

The Remopla language contains four distinct constructs concerning modules. A *module declaration* is required if a module is called lexically before its definition. A *module definition* implements the module. A *module call* is used to pass control to a module. Finally, the *return statement* returns control (and possibly a value) from a module to its caller.

10.1 Declarations and Definitions

⟨module declaration⟩ → `module`
    ⟨return type⟩ ⟨module name⟩ `(`⟨parameter⟩`,`*`);`
⟨module definition⟩ → `module`
    ⟨return type⟩ ⟨module name⟩ `(`⟨parameter⟩`,`*`) {`
        ⟨variable declaration⟩*
        ⟨statement⟩$^+$
    `}`
⟨parameter⟩ → `bool` ⟨variable spec⟩ | `int` ⟨int variable spec⟩
    | `enum` ⟨type name⟩ ⟨variable spec⟩
    | `struct` ⟨type name⟩ ⟨variable spec⟩
⟨return type⟩ → `void`
    | `int` [`[`⟨dimen⟩`]`] [`[`⟨dimen⟩`]`] [`(`⟨cnst expr⟩`)`]
    | ⟨data type⟩ [`[`⟨dimen⟩`]`] [`[`⟨dimen⟩`]`]
⟨data type⟩ → `bool` | `enum` ⟨type name⟩ | `struct` ⟨type name⟩
⟨module name⟩ → ⟨identifier⟩
⟨type name⟩ → ⟨identifier⟩
⟨dimen⟩ → ⟨cnst expr⟩ | ⟨cnst expr⟩`,`⟨cnst expr⟩

Each module definition starts with a header that specifies the prototype of the module: name, formal parameters and return type of the module.

The name is an identifier. Formal parameters essentially form a comma-separated list of variable declarations (see section 6) where each parameter is given its type individually. The return type is very similar to the declaration of a variable, too, the variable name is just omitted. The return type can also be a special keyword `void` which means that the module has no return value.

After the header follows the module body: declarations of local variables and actual statements implementing the module. Declarations of local variables are exactly the same as declaration of global ones. Local variables of the same name as global variables are not allowed.

The prototype of each module must be known prior to a call to the module. It cannot be satisfied by the order of definitions only if the module calls form a cycle. Module declarations in the section of global declarations (see section 4) can be used to specify the prototype of modules prior to any definitions and calls. The declaration consists of the module header and each declared module must be defined with exactly the same header in the model later on.

10.2 Return Statement

⟨return statement⟩ → `return;`
  | `return` ⟨bool expr⟩`;`
  | `return` ⟨int expr⟩`;`
  | `return` ⟨variable⟩`;`

The return statement can only be used within a module and the form must match the return type of the module. It terminates execution of the module and returns control and a value to the caller of the module. It follows from the syntax that the return value must be stored in a local or a global variable if the type of the return value is more complex than a simple integer or boolean value.

A return statement returning a boolean value or no value at all does not terminate any execution paths implicitly.

Returning an integer value implicitly terminates all execution paths in which the expression cannot be evaluated. It also terminates the paths in which the value of the expression does not fit into the bounds of the return value as given by the module declaration or by the `DEFAULT_INT_BITS` internal constant.

If the return statement returns a more complex value it must be given in a variable. The variable given to the return statement can be an array element. In this case, the return statement implicitly terminates all execution paths in which the index of the element cannot be evaluated or is out of range.

10.3 Module Call

$\langle$call statement$\rangle$ $\rightarrow$ $\langle$module call$\rangle$ `;`
  | $\langle$variable$\rangle$ `=` $\langle$module call$\rangle$ `;`
$\langle$module call$\rangle$ $\rightarrow$ $\langle$module name$\rangle$ `(`$\langle$argument$\rangle$`,`*`)`
$\langle$argument$\rangle$ $\rightarrow$ $\langle$variable$\rangle$ | $\langle$int expr$\rangle$ | $\langle$bool expr$\rangle$

Module calls are statements which pass control to a module along with the actual parameters the module requires. Modules can be called as procedure and, if they return a value, in an assignment that stores the return value to a variable. Note that it is not possible to use a module call in an expression even if it returns simple integer or boolean value. Nor can it be put in parallel with other assignments and module calls.

The parameters are passed by value and must comply with the module header. Values of integer and boolean parameters can be given as integer and boolean expressions, respectively. More complex values must be passed in a variable.

Passing parameters by value is essentially equivalent to assignments. The rules for implicit termination of execution paths are hence the same (see section 9.1). Parameters of complex data types can cause termination of execution paths, similarly to the return statement (see section 10.2) if the index of some array element is invalid.

References

[1] Stefan Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, Technische Universität München, 2002.