THE NUMERICAL TRANSITION SYSTEMS LIBRARY
Version 1.0

CONTRIBUTORS:

RADU IOSIF (VERIMAG/CNRS)
FILIP KONECNY (VERIMAG/FIT VUTBR)
MARIUS BOZGA (VERIMAG/CNRS)

Contents

Chapter 1

The Numerical Transition Language

Numerical Transition Systems (NTS, also referred to as Counter Systems, Counter Automata or Counter Machines) are simple models of computation involving infinite (or very large) data domains, such as integers or real numbers. Despite their apparent simplicity, NTS can, in principle, model any real-life computer system, ranging from hardware circuits to programs. As a consequence, an important number of tools have emerged, addressing verification problems, such as reachability or termination, and deploying various techniques (widening, predicate abstraction, acceleration, etc.).

The *Numerical Transition Language* is a common language for describing numerical transition systems. In addition to the basic scalar types, we consider one-dimensional array types, defined over scalars. The semantics of array manipulation interprets arrays as functions, and considers them to be first-class citizens.

Since even the most simple programs are usually structured into subcomponents, we consider systems that are described as compositions of subsystems. There are two types of compositions: *hierarchical* (a subsystem invokes another subsystem in the same way a procedure invokes another procedure in a program) and *parallel* (two or more subsystems run in parallel and communicate via the global variables).

The design of the Numerical Transition Language (NTL) is inspired by the input language of several existing tools for the analysis of numerical transition systems, such as: ARMC, FAST, FLATA and INTERPROC. Although a single tool is unlikely to deal with models using all the features of NTL (such as e.g., parallel recursive systems with arrays), most tools can treat important subfragments of the language, and are amenable to extensions. The aim of NTL is not that of replacing existing languages, but rather that of providing means of interoperability between tools developed by different groups, and based on different principles.

1.0.1 Two Warm-up Examples

The following example is an NTS implementing the Syracuse function:

```
nts syracuse;

syracuse {
  in x :  int;  // x contains the input value at the initial state
  out y :  int; // y contains the output value at the final state
  initial si;   // initial state
  final sf;     // final state
  si -> sf { exists k : (int . 2*k=x and y'=k) }
  // if x is even return x/2
  si -> sf { exists k : (int . 2*k+1=x and y'=3*x+1) }
  // if x is odd return 3*x+1
}

main {
  n :  int; // a local variable
  initial s0;
  final s3;
  s0 -> s1 { n'>0 } // n is randomly assigned a strictly positive value
  s1 -> s2 { n'=syracuse(n) } // apply syracuse to n
  s2 -> s1 { n>1 and havoc() } // repeat while n > 1
  s2 -> s3 { n=1 and havoc() } // finish when n=1
}
```

Experimental evidence shows that this system terminates for a large set of initial values of n. However, no general termination proof has been given so far.

The following example is an NTS implementing McCarthy's 91 function:

```
nts mccarthy;

mc91 {
  in x :  int;
  out y :  int;
  t :  int;
  initial si;
  final sf;
  // if x > 100 return x - 10
  si -> sf { x>100 and y'=x-10 and havoc(y) }
  // else return mc91(mc91(x + 11))
```

```
  si -> s1 { x<100 and havoc() }
  // t is needed to compose the recursive calls
  s1 -> s2 { t'=mc91(x+11) }
  s2 -> sf { y'=mc91(t) }
}

main {
  i,j :  int;
  initial si;
  error se; // error state
  si -> s1 { j'=mc91(i) }
  // error if not j=i-10 for i>101
  s1 -> se { i>101 and j!=i-10 and havoc() }
  // error if not i=91 for i<=101
  s1 -> se { i<=101 and j!=91 and havoc() }
}
```

Although a proof of correctness for McCarthy91 exists, this function is still considered to be a challenge for automated program verification. Notice also the different properties checked: termination for the Syracuse example, and safety (unreachability of error states) for the McCarthy91 example.

1.1 Lexical Structure

An NTL specification is a sequence of tokens. The tokens are defined by the following syntax:

$$
\begin{array}{rcl}
\langle \mathit{type} \rangle & ::= & \{\mathbf{int}, \mathbf{real}, \mathbf{bool}\} \\
\langle \mathit{numeral} \rangle & ::= & \{\mathbf{0}\} \mid \{\mathbf{1}..\mathbf{9}\}\{\mathbf{0}..\mathbf{9}\}^* \\
\langle \mathit{decimal} \rangle & ::= & \langle \mathit{numeral} \rangle \ \{\mathbf{.}\}\{\mathbf{0}..\mathbf{9}\}^+ \\
\langle \mathit{boolean} \rangle & ::= & \{\mathbf{true}, \mathbf{false}\} \\
\langle \mathit{idn} \rangle & ::= & \{\mathbf{a}..\mathbf{z}, \mathbf{A}..\mathbf{Z}\}\{\mathbf{a}..\mathbf{z}, \mathbf{A}..\mathbf{Z}, \mathbf{0}..\mathbf{9}, \_\}^* \\
\langle \mathit{idp} \rangle & ::= & \langle \mathit{idn} \rangle \ \{\mathbf{'}\} \\
\langle \mathit{id} \rangle & ::= & \langle \mathit{idn} \rangle \mid \langle \mathit{idp} \rangle
\end{array}
$$

Note the distinction between *unprimed* identifiers $\langle \mathit{idn} \rangle$, used to denote current values of variables, and *primed* identifiers $\langle \mathit{idp} \rangle$, used to denote their values after one transition step.

1.2 Types and Declarations

The basic (also refer ed to as scalar) types of NTL are: boolean (**bool**), integer (**int**) and real (**real**). Variables are declared in blocks of the same type, e.g.:

```
x1,x2,x3 :  int
```

Declaration blocks can be chained as in, e.g.:

```
x1,x2,x3 :  int, x4,x5 :  real
```

Since we consider systems that are compositions of subsystems, the classical notions of *global* and *local* variable declarations apply. Global variables are declared outside the body of a subsystem, and are visible everywhere in the system. Local variables are declared inside the body of particular subsystem, and are visible inside that body only. For instance:

```
g1 :  int, g2 :  real; // global declarations

main {
 l1 :  int, l2 :  real; // local declarations
 ...
}
```

The names of local variables must not conflict with global variable names (in other words, variable shadowing is not supported).

A variable can be declared to be a *parameter*, meaning that its value does not change during the execution. Parameters are specified by enclosing the declaration sequence between the keyword **par** and a semicolon as in, e.g.:

```
par x1,x2,x3 :  int;
```

A subsystem may declare certain local variables as *input* and other as *output* as in, e.g.:

```
foo {
 in i1 : int, i2 : bool, i3, i4 : int;
 out o1, o2, o3 : int, o4 : real;
 ...
}
```

The order in which input and output variables are declared is important for invocation (hierarchical composition), thus the modifiers **in** and **out** specify *ordered sequences* of variables.

1.2.1 Arrays

The NTL language considers multi-dimensional arrays of any of the basic types. There are two kinds of declarations: arrays and *array references*. In the first case one must give the size as an arithmetic term of type **int**, as in e.g.:

```
x[5] : int, y[N], z[2*N+3*M+5] : real;
```

The variables occurring in a size specifier must be scalar parameters of type **int**. Size specifier may also involve array-size operators applied on input arrays of a subsystem (see Section 1.5). If the size is not specified, one declares an array reference, e.g.:

```
a[], b[] : int;
```

An array reference is used as a name for array objects. Array references are the only array variables that can be assigned to in a transition relation (the meaning of assignments to array references will be made clear in the next section).

The elements of an array $a[N]$ are indexed $0,\ldots,N-1$. If a is an array or an array reference, the expression $|a|$ denotes the size of a. For instance in the example below, $|a|=5$, and $|b|=0$, if b is not initialized, whereas $|b|=5$, immediately following the assignment of a to b.

```
a[5],b[] :  int; // |a| = 5, |b| = 0
b = a; // |a| = 5, |b| = 5
```

Multi-dimensional arrays are declared by multiple size specifiers of the form

$$a[e_1]\ldots[e_m]\underbrace{[\,]\ldots[\,]}_{n-times} : \tau$$

where $m+n \geq 1$, τ is a basic type, and $e_1,\ldots,e_m$ are well-typed index terms (defined in Section 1.3.1). Then, a is said to be an m-dimensional array of references to n-dimensional arrays over the basic type τ.

For instance, consider the following declaration:

```
c[2][3] : int, d[2][], e[][] : real;
```

Here c denotes an array of size 2 of arrays of size 3 of integers, whereas d is an array of size 2 of real array references. Notice that one cannot declare an array of references to arrays of specified sizes, e.g.:

```
f[][3] : int; // illegal declaration
```

As will be discussed next, one can assign an integer value to $c[i][j]$, for $0 \leq i < 2$ and $0 \leq j < 3$, and an array object to $d[i]$, for $0 \leq i < 2$. However, assigning to $c[i]$, for $0 \leq i < 2$ is not permitted, since $c[i]$ is an array object, not an array reference.

A *pure* array reference is an array declared in the form

$$a\underbrace{[\,]\ldots[\,]}_{n-times} : \tau$$

where $n > 0$ and τ is a basic type. All input and output array variables of subsystems are required to be pure references (e.g. *b*, *e*, but not *a*, *c*, *d* in the above example).

1.2.2 Formal Syntax of Declarations

The full formal syntax of NTL variable declarations is given below. The ⟨*arith-term*⟩ non-terminal is defined in Section 1.3.1. The ⟨*annotations*⟩ non-terminal is defined in Section 1.8.

⟨*basic*⟩ ::= ⟨*annotations*⟩ ⟨*idn*⟩
⟨*array*⟩ ::= ⟨*basic*⟩ **[** ⟨*arith-term*⟩ **]**
| ⟨*array*⟩ **[** ⟨*arith-term*⟩ **]**
⟨*array-ref*⟩ ::= ⟨*basic*⟩ **[ ]**
| ⟨*array*⟩ **[ ]**
| ⟨*array-ref*⟩ **[ ]**
⟨*array-pure-ref*⟩ ::= ⟨*basic*⟩ **[ ]**
| ⟨*array-pure-ref*⟩ **[ ]**

⟨*decl-lit*⟩ ::= ⟨*basic*⟩ | ⟨*array*⟩ | ⟨*array-ref*⟩
⟨*decl-lits*⟩ ::= ⟨*decl-lits*⟩ **,** ⟨*decl-lit*⟩
| ⟨*decl-lit*⟩
⟨*decl-block*⟩ ::= ⟨*decl-lits*⟩ **:** ⟨*type*⟩
⟨*decl-blocks*⟩ ::= ⟨*decl-blocks*⟩ **,** ⟨*decl-block*⟩
| ⟨*decl-block*⟩

⟨*decl-par-lit*⟩ ::= ⟨*basic*⟩ | ⟨*array*⟩
⟨*decl-par-lits*⟩ ::= ⟨*decl-par-lits*⟩ **,** ⟨*idn*⟩
| ⟨*decl-par-lit*⟩
⟨*decl-par-block*⟩ ::= ⟨*decl-par-lits*⟩ **:** ⟨*type*⟩
⟨*decl-par-blocks*⟩ ::= ⟨*decl-par-blocks*⟩ **,** ⟨*decl-par-block*⟩
| ⟨*decl-par-block*⟩

⟨*decl-io-lit*⟩ ::= ⟨*basic*⟩ | ⟨*array-pure-ref*⟩
⟨*decl-io-lits*⟩ ::= ⟨*d-io-lits*⟩ **,** ⟨*idn*⟩
| ⟨*d-io-lit*⟩
⟨*decl-io-block*⟩ ::= ⟨*decl-io-lits*⟩ **:** ⟨*type*⟩
⟨*decl-io-blocks*⟩ ::= ⟨*decl-io-blocks*⟩ **,** ⟨*decl-io-block*⟩
| ⟨*decl-io-block*⟩

⟨*decl*⟩ ::= ⟨*decl-blocks*⟩ **;** | **par** ⟨*decl-par-blocks*⟩ **;**
⟨*decl-glob*⟩ ::= ⟨*decl-glob*⟩ ⟨*decl*⟩ | ε
⟨*in*⟩ ::= **in** ⟨*decl-io-blocks*⟩ **;** | ε
⟨*out*⟩ ::= **out** ⟨*decl-io-blocks*⟩ **;** | ε
⟨*decl-loc*⟩ ::= ⟨*decl-loc*⟩ ⟨*decl*⟩ | ⟨*in*⟩ ⟨*out*⟩

Notice that array references cannot be declared as parameters, since the parameters cannot be assigned to, whereas the array references become useful only via assignments. Also, only pure array references can be specified in the input/output declaration of a subsystem, since passing arrays to subsystems has the same effect as assigning to them. For a detailed discussion on array assignment, one can refer to Section 1.3.2.

1.3 First-order Arithmetic

The NTL language relies on first-order arithmetic in order to describe the initial configurations and the transition relation of systems. This section describes the syntax adopted for writing first-order arithmetic formulae.

1.3.1 Literals and Terms

A *boolean literal* is either a variable identifier or a boolean constant (**true** or **false**). A *boolean term* is an expression composed of boolean literals connected via the boolean operators **not**, **and**, **or**, **imply** and **equiv**. Precedence of the connectives is as follows: **not** > **and** > **or** > **imply** > **equiv**. These operators may be shorthanded by C/C++-like operators `!`, `&&`, `||`, `->`, `<->`. The formal syntax of boolean terms is given below:

⟨*bop*⟩ ::= {**and**, **or**, **imply**, **equiv**, `&&`, `||`, `->`, `<->`}
⟨*not*⟩ ::= {**not**, `!`}
⟨*bool-lit*⟩ ::= ⟨*id*⟩ | **true** | **false**
⟨*bool-term*⟩ ::= ⟨*bool-lit*⟩ | **(** ⟨*bool-term*⟩ **)**
| ⟨*not*⟩ ⟨*bool-term*⟩
| ⟨*bool-term*⟩ ⟨*bop*⟩ ⟨*bool-term*⟩

An *arithmetic literal* is either a variable identifier, the keyword **tid** (denoting the current thread identifier), or a positive numeral. The arithmetic and array terms are defined recursively.

Given an array declaration

$$a[e_1]\dots[e_m]\underbrace{[\,]\dots[\,]}_{n-times} : \tau$$

where $m+n \geq 1$, τ is a basic type, and $e_1,\dots,e_m$ are arithmetic terms called *index terms*, the following are valid array terms and their corresponding types (i_k are index terms in the following)

	array term	type
1)	$\lvert a\rvert$	**int**
2)	$a[i_1]\dots[i_{m+n}]$	τ
3)	$a[i_1]\dots[i_k]$	$(n-k)$-dimensional array over τ $(0 \leq k < m+n)$
4)	$a'[i_1]\dots[i_{m+n}]$	τ
5)	$a'[i_1]\dots[i_m]$	reference to n-dimensional array over τ

An *arithmetic term* is an expression consisting of arithmetic literals and array terms of kind 1) and 2), connected via the arithmetic operators `+`, `-`, `*`, `/` and `%` (remainder) and brackets. The arithmetic operators have standard precedence i.e., `*`, `/` and `%` have precedence over `+` and `-`. The `%` operator is restricted to **int** type. The semantics of `/` and `%` for the **int** type is given below:

$$\begin{array}{rll} x\%y = z & \textbf{iff} & 0 \leq z < |y| \text{ and } \exists k\,.\, k*y = x-z \\ x/y = z & \textbf{iff} & (y*z)+(x\%y) = x \end{array}$$

For instance, given a declaration

```
x, y, a[N] : int;
```

the following is a valid arithmetic term:

```
2*x + (- 3*y) + (- 7) + |a| + a[0]
```

The use of array terms of kind $3)-5)$ will become apparent in Section 1.3.2. The formal syntax of arithmetic terms is given below:

$$
\begin{array}{rcl}
\langle \textit{arith-lit} \rangle & ::= & \langle \textit{id} \rangle \mid \textbf{tid} \mid \langle \textit{numeral} \rangle \mid \langle \textit{decimal} \rangle \\
\langle \textit{array-read} \rangle & ::= & \langle \textit{idn} \rangle \ \textbf{[} \ \langle \textit{arith-lit} \rangle \ \textbf{]} \mid \langle \textit{array-read} \rangle \ \textbf{[} \ \langle \textit{arith-lit} \rangle \ \textbf{]} \\
\langle \textit{array-term} \rangle & ::= & \langle \textit{array-read} \rangle \mid |\langle \textit{idn} \rangle| \\
\langle \textit{aop} \rangle & ::= & \{+, -, *, /, \%\} \\
\langle \textit{sign} \rangle & ::= & \{-\} \mid \varepsilon \\
\langle \textit{arith-term} \rangle & ::= & \langle \textit{sign} \rangle \ \langle \textit{arith-lit} \rangle \mid \langle \textit{sign} \rangle \ \langle \textit{array-term} \rangle \\
 & \mid & \textbf{(} \ \langle \textit{arith-term} \rangle \ \textbf{)} \\
 & \mid & \langle \textit{arith-term} \rangle \ \langle \textit{aop} \rangle \ \langle \textit{arith-term} \rangle \\
\langle \textit{arith-list} \rangle & ::= & \langle \textit{arith-term} \rangle \mid \langle \textit{arith-list} \rangle \ \textbf{,} \ \langle \textit{arith-term} \rangle \\
\langle \textit{multi} \rangle & ::= & \textbf{[} \ \langle \textit{arith-term} \rangle \ \textbf{]} \ \langle \textit{multi} \rangle \mid \textbf{[} \ \langle \textit{arith-list} \rangle \ \textbf{]} \mid \varepsilon \\
\langle \textit{array-write} \rangle & ::= & \langle \textit{idp} \rangle \ \langle \textit{multi} \rangle
\end{array}
$$

Typing Rules

The typing rules for arithmetic terms are simple. A primed literal x' always has the same type as the corresponding unprimed literal x. All literals occurring in one term must have the same basic type, which, in turn, is the type of the term. Furthermore, an arithmetic term can be only of type **int** or **real**. In particular, the **tid** literal is implicitly of type **int**. For instance, the term 2*x + (- 3*y) + (- 7) is well typed if and only if x and y are variables of the same type, which must be one of **int** or **real**.

In particular, index terms $i_1, \ldots, i_k$ of an array term $a[i_1] \ldots [i_k]$ must be of type **int**, and moreover, no primed variables are allowed to occur inside $i_1, \ldots, i_k$. For instance, $A[3*n+2*|b|+5]$ is a valid array literal if and only if n is a variable of type **int**, whereas $A[3*n'+2*|b|+5]$ is not, since the indexing term contains a primed variable.

1.3.2 Atomic Propositions and Formulae

An atomic proposition is either a boolean term, or a relation of the form $t_1 \sim t_2$, where t_1 and t_2 are arithmetic terms of the same type, and $\sim \in \{=, !=, <=, <, >=, >\}$ is a binary relation symbol. The left- and the right-hand side of $\sim$ must be of the same type.

To define array assignments, consider again the following array declaration

$$a[e_1] \ldots [e_m] \underbrace{[\,] \ldots [\,]}_{n-times} : \tau;$$

in other words, a is m-dimensional array of references to n-dimensional arrays over type τ where $m+n \geq 1$. Assignments to a are restricted to the following two forms:

$$a'[i_1] \ldots [i_m] = t \text{ (if } n > 0)$$
$$a'[i_1] \ldots [i_{m+n}] = u$$

where i_k are well-typed index-terms, t is a valid array term exposing an n-dimensional array (reference) over type τ, and u is an expression of type τ. Moreover, no primed symbol can appear in t and u. For example, consider the following declaration:

a[], b[], c[5], d[2][], e[][] : **int**;

Here $a' = c$, $b' = a$, $d'[1] = b$, $d'[0][1] = 5$, $e' = d$ are valid atomic propositions. However, assignments $c' = a$, and $d' = e$ are incorrect in the context of the above declarations.

Additionally, we allow atomic formulae for multiple assignments of the form

$$a'[i_1]\dots[i_{m-1}][j_1,\dots,j_k] = [t_1,\dots,t_k] \text{ (if } n > 0)$$
$$a'[i_1]\dots[i_{m+n-1}][j_1,\dots,j_k] = [u_1,\dots,u_k]$$

which denotes an assignment to positions

$$a'[i_1]\dots[i_{m-1}][j_1]$$
$$\vdots$$
$$a'[i_1]\dots[i_{m-1}][j_k]$$

or, respectively, to positions

$$a'[i_1]\dots[i_{m+n-1}][j_1]$$
$$\vdots$$
$$\text{and } a'[i_1]\dots[i_{m+n-1}][j_k]$$

Multiple assignments are interpreted as a succession of basic assignments performed from left to right, i.e. first assigning to $a'[i_1]\dots[i_{m-1}][j_1]$ and last to $a'[i_1]\dots[i_{m-1}][j_k]$. More details can be found in Section 2.2.

Note that unprimed array terms of the form $a[i_1]\dots[i_{m+n}]$ can be used freely in arithmetic terms, as discussed in Section 1.3.1, for instance:

```
a[i+1] = a[i]
a[n] <= d[2*n-1][0] + |a| + 1
b[2*n] >= e[n][k]
```

However, the atomic term `d2[0] = a` is not a valid one, since `d1[0]` and `a` are not valid arithmetic terms.

The formal syntax of atomic propositions is given below (semantics of *havoc* is explained at the end of this section):

$$
\begin{array}{rcl}
\langle \textit{rop} \rangle & ::= & \{=, !=, <=, <, >=, >\} \\
\langle \textit{mop} \rangle & ::= & \langle \textit{rop} \rangle \mid \langle \textit{bop} \rangle \\
\langle \textit{idn-list} \rangle & ::= & \langle \textit{idn} \rangle \mid \langle \textit{idn-list} \rangle \ \textbf{,} \ \langle \textit{idn} \rangle \\
\langle \textit{idn-list-e} \rangle & ::= & \langle \textit{idn-list} \rangle \mid \varepsilon \\
\langle \textit{havoc} \rangle & ::= & \textbf{havoc} \ \textbf{(} \ \langle \textit{idn-list-e} \rangle \ \textbf{)} \\
\langle \textit{atom} \rangle & ::= & \langle \textit{bool-term} \rangle \\
 & \mid & \langle \textit{arith-term} \rangle \ \langle \textit{rop} \rangle \ \langle \textit{arith-term} \rangle \\
 & \mid & \langle \textit{array-write} \rangle \ = \ \textbf{[} \ \langle \textit{arith-list} \rangle \ \textbf{]} \\
 & \mid & \langle \textit{havoc} \rangle
\end{array}
$$

The syntax of formulae is given below:

$$
\begin{array}{rcl}
\langle \textit{quantifier} \rangle & ::= & \{\textbf{forall}, \textbf{exists}\} \\
\langle \textit{q-type} \rangle & ::= & \langle \textit{type} \rangle \mid \langle \textit{type} \rangle \ \textbf{[} \ \langle \textit{arith-term} \rangle \ \textbf{,} \ \langle \textit{arith-term} \rangle \ \textbf{]} \\
\langle \textit{formula} \rangle & ::= & \langle \textit{atom} \rangle \mid \textbf{(} \ \langle \textit{formula} \rangle \ \textbf{)} \\
 & \mid & \langle \textit{formula} \rangle \ \langle \textit{bop} \rangle \ \langle \textit{formula} \rangle \mid \textbf{not} \ \langle \textit{formula} \rangle \\
 & \mid & \langle \textit{quantifier} \rangle \ \langle \textit{idn-list} \rangle \ \textbf{:} \ \langle \textit{q-type} \rangle \ \textbf{.} \ \langle \textit{formula} \rangle
\end{array}
$$

Notice that quantified variables must be typed by a basic (scalar) type. Consequently, array variables cannot occur in the scope of a quantifier. Also, the quantified variables are supposed to be unprimed. The use of intervals within type specifications of quantified variables is meant as syntactic sugar, namely:

- **forall** i : τ $[t_1, t_2]$. $\phi(i)$ stands for **forall** i : τ . (t_1 <= i **and** i <= t_2 **imply** $\phi(i)$)
- **exists** i : τ $[t_1, t_2]$. $\phi(i)$ stands for **exists** i : τ . (t_1 <= i **and** i <= t_2 **and** $\phi(i)$)

For example, the following are syntactically valid formulae:

```
N : int, x,y,z,a[N] : int;
forall i :  int[0,N-2] . (a[i+1] > a[i])
exists n :  int . (y * n <= x and x < y * (n + 1) and z'=n)
```

The first formula above expresses the fact that a is a sorted integer array without duplicate values. The second one encodes the integer division relation $z' = \lfloor \frac{x}{y} \rfloor$.

Furthermore, as a syntactical sugar, one can use the $=$ operator instead of the **equiv** operator. As an example, given a declaration $b_1, b_2 : \textit{bool}$, the following are valid and equivalent assignements:

- b_2' **equiv** b_1 **or** b_2
- $b_2' = (b_1$ **or** $b_2)$

Also note that one can write b_2' as a shorthand for $b_2' =$ **true**.

About Implicit Copies

The intuition behind the use of primed identifiers is that an occurrence of a primed variable identifier in an atomic proposition has the effect of an assignment to that variable (i.e., it produces a possible change of its value in the next step). In order to improve readability of the transition rules, we define

$$havoc(X) \equiv \bigwedge_{x \notin X} x' = x$$

where X are variables in the current scope. For instance, given a scope with variables x, y, z, w, a formula

```
(z<=w and x'=5 and havoc(x)) or (y'<=3 and havoc(y))
```

is a shorthand for

```
(z<=w and x'=5 and y'=y and z'=z and w'=w) or
      (y'<=3 and x'=x and z'=z and w'=w)
```

Note that y' in the first disjunct and x' in the second disjunct are left random.

About Assignments to Array References

Since arrays are considered first-class values (i.e., finite sequences of scalar data values), an assignment to an array variable would change both its size and its content. However, changing the size of an array variable, such as c[5] : **int** in the example above, would result in an array value that would be inconsistent with the declaration. Hence we allow only assignments to array references , whom sizes are not declared, e.g. a[], b[] : **int**.

1.4 Basic Systems

A basic NTS is a component of a global system. A basic NTS is identified by a unique name that is visible in the entire system. The body of a basic NTS specifies the local variables (including input and output variable declarations), the *initial*, *final* and *error* states, and a list of transitions. Typically, final states are used to check termination, whereas error states are used to state safety properties. Formal definitions of these properties are given in the next chapter.

A transition is defined by a source state, a destination state and a transition rule. For instance, the following transition rule changes the control from state q1 to q2 and increments the value of the variable x if it is less than 100:

```
q1 -> q2 { x<=100 and x'=x+1 }
```

Optionally, transitions can be labeled by identifiers that are unique within their scope (the body of the NTS definition). These identifiers can be used to display error traces. For instance:

```
t1:  q1 -> q2 { x<=100 and x'=x+1 }
```

There are two kinds of transition rules: *arithmetic relations* and *calls* to other subsystems (basic NTS). All variables that do not appear primed in a transition relation implicitly carry their values from the source to the destination state. Consequently, an empty rule carries all values from the source to the destination state.

The formal syntax of basic NTS definition is given below. The ⟨*call*⟩ non-terminal is defined in the next section. The ⟨*states*⟩ non-terminal serves as an optional declaration of control states (see Section 1.8 for its purpose):

⟨*rule*⟩ ::= ⟨*formula*⟩ | ⟨*call*⟩ | ε
⟨*transition*⟩ ::= ⟨*idn*⟩ **'->'** ⟨*idn*⟩ ⟨*annotations*⟩ **'{'** ⟨*rule*⟩ **'}'**
| ⟨*idn*⟩ **:** ⟨*idn*⟩ **'->'** ⟨*idn*⟩ ⟨*annotations*⟩ **'{'** ⟨*rule*⟩ **'}'**
⟨*transitions*⟩ ::= ⟨*transition*⟩ ⟨*transitions*⟩ | ε

⟨*statelist*⟩ ::= ⟨*statelist*⟩ **,** ⟨*idn*⟩ | ⟨*idn*⟩
⟨*statelist-a*⟩ ::= ⟨*statelist*⟩ **,** ⟨*annotations*⟩ ⟨*idn*⟩ | ⟨*annotations*⟩ ⟨*idn*⟩
⟨*states*⟩ ::= **states** ⟨*statelist-a*⟩ **;** | ε
⟨*statesinit*⟩ ::= **initial** ⟨*statelist*⟩ **;**
⟨*statesfin*⟩ ::= **final** ⟨*statelist*⟩ **;** | ε
⟨*stateserr*⟩ ::= **error** ⟨*statelist*⟩ **;** | ε
⟨*statemarks*⟩ ::= ⟨*states*⟩ ⟨*states-init*⟩ ⟨*states-fin*⟩ ⟨*states-err*⟩

⟨*nts-body*⟩ ::= ⟨*declar-loc*⟩ ⟨*statemarks*⟩ ⟨*transitions*⟩
⟨*nts-basic*⟩ ::= ⟨*annotations*⟩ ⟨*idn*⟩ **'{'** ⟨*nts-body*⟩ **'}'**

Notice that a basic NTS must specify at least one initial state.

1.5 Hierarchical Systems

A hierarchical NTS is a collection of basic NTS, of which some are denoted as entry points of the system. The entry points can be given in a global instance declaration, which also defines the parallel threads (the instance declaration is presented in detail in the next section). For sequential systems with only one entry point, this can be specified using the name **main**.

In a hierarchical NTS basic subsystems can invoke other subsystems, via *call* transition rules. A call rule consists of a name of a callee and a list of actual parameters (arithmetic terms) followed by a list of primed return variables.

The types of the actual parameter terms must match the types of input variables of the callee. Similarly, the types of the return variables must match the types of the output variables of the callee. In particular, the input array variables (which must be declared as pure references) match with both array references and arrays of the same basic type. On the other hand, the output array variables (which must be declared as pure references too) match only with pure array references of the same basic type. Note that passing arrays as formal parameters, or returning arrays, has the same effect as assigning to array references. For the reasons described in section 1.3.2, a' may appear at most once in the list of return variables for each array variable a.

For instance, consider the following declarations:

```
foo {
  in (i1, i2 : int, r : real, a[] : int)
  out (n : int, i3, b[] : int)
  ...
}

bar {
  in (x : int, q : real)
  out (m : int, c[] : int)
  par N : int;
  d[N], y, z : int;
  ...
}
```

Then, *bar* can call *foo* as follows:

```
foo(x+3,x+y,q,d,m',y',c')
```
(1.1)

As a syntactic sugar, we allow also the following:

```
(m',y',c') = foo(x+3,x+y,q,d)
```
(1.2)

Variables that do not appear in the list of return variables remain unchanged upon return. One may also specify explicitely which variables remain unchanged upon return using *havoc*:

```
(m',y',c') = foo(x+3,x+y,q,d) and havoc(m,y,c)
```
(1.3)

```
(m',y',c') = foo(x+3,x+y,q,d) and havoc(m,y,c,d)        (1.4)
```

Note that (1.1), (1.2), and (1.3) are equivalent. However, (1.3) is not equivalent to (1.4) since variable *d* is left random in (1.4).

The formal syntax of a call rule is given below:

⟨*arglist*⟩ ::= ⟨*arith-list*⟩ | ε
⟨*ret-terms*⟩ ::= ⟨*idp*⟩ | ⟨*ret-terms*⟩ **,** ⟨*idp*⟩
⟨*retlist*⟩ ::= ⟨*ret-terms*⟩ | ε
⟨*call-base*⟩ ::= ⟨*idn*⟩ **(** ⟨*arglist*⟩ **,** ⟨*retlist*⟩ **)**
| ⟨*idp*⟩ **=** ⟨*idn*⟩ **(** ⟨*arglist*⟩ **)**
| **(** ⟨*ret-terms*⟩ **) =** ⟨*idn*⟩ **(** ⟨*arglist*⟩ **)**
⟨*call*⟩ ::= ⟨*call-base*⟩ | ⟨*call-base*⟩ **and** ⟨*havoc*⟩

1.5.1 Two Examples

The example below implements Fibonacci's recursive function:

```
nts fibonacci;

fib {
  in x :  int;
  out y :  int;
  t1, t2 :  int;
  initial si;
  final sf;
  si -> sf { x=0 and y'=0 and havoc(y) }
  si -> sf { x=1 and y'=1 and havoc(y) }
  si -> s1 { x>1 and havoc() }
  s1 -> s2 { t1'=fib(x-2) }
  s2 -> s3 { t2'=fib(x-1) }
  s3 -> sf { y'=t1+t2 and havoc(y) }
}
main {
  x, y :  int;
  initial si;
  error se;
  si -> s1 { x>=0 and havoc() }
  s2 -> s2 { y'=fib(x) }
  s2 -> se { x>=4 and y<=x and havoc() }
}
```

The example below describes an NTS that concatenates two arrays:

```
nts concatenation;

concat {
  in a1[], a2[] :  int;
  out a[] :  int;
  t[|a1|+|a2|] :  int, i :  int;
  initial si;
  final sf;
  si -> s1 { i'=0 and havoc(i) }
  s1 -> s1 { i<|a1| and t'[i]=a1[i] and i'=i+1 and havoc(t,i) }
  s1 -> s2 { i=|a1| and i'=0 and havoc(i) }
  s2 -> s2 { i<|a2| and t'[i+|a1|]=a2[i] and i'=i+1 and havoc(t,i) }
  s2 -> sf { i=|a2| and a'=t and havoc(a) }
}
```

1.6 Parallel Systems

A parallel NTS is a collection of basic NTS with a global specification of instances that run in parallel. For example

```
par N : int;
instances producer[N], consumer[2*N];
```

declares a parametric concurrent system in which there are N instances of the producer thread running in parallel with 2*N instances of the consumer thread. As a general rule, the NTS used in an **instances** declaration must not declare input nor output variables.

Each instance has access to a predefined variable **tid** of type **int**. Two different instances have different **tid** values. Moreover, the order in which the instances are specified determines the value of the **tid** variable. For instance, in the example above the value of **tid** for producer threads ranges between 0 and $N-1$, whereas the value of **tid** for consumer threads ranges between N and $2*N-1$. Since the values of **tid** are pairwise distinct, all values in the above ranges are used. These considerations are useful to the specification of parallel systems with shared array resources.

The following gives the formal syntax of the instance declaration:

$$\begin{array}{rcl} \langle \textit{instance} \rangle & ::= & \langle \textit{idn} \rangle \ \textbf{[} \ \langle \textit{arith-term} \rangle \ \textbf{]} \\ \langle \textit{inst-list} \rangle & ::= & \langle \textit{instance} \rangle \mid \langle \textit{inst-list} \rangle \ \textbf{,} \ \langle \textit{instance} \rangle \\ \langle \textit{instances} \rangle & ::= & \textbf{instances} \ \langle \textit{inst-list} \rangle \ \textbf{;} \end{array}$$

1.7 Global NTS Specification

The global specification of an NTS consists of a declaration of global variables and parameters, an initial condition (a formula describing the initial configurations), an instance declaration, and a list of basic NTS. The initial condition is written using full first-order arithmetic, e.g.:

```
par N : int;
a[N] : int;
init forall i : int[0,N-1] . a[i] = 0;
```

The formal syntax of a global NTS specification is given below. The start symbol is the ⟨*system*⟩ non-terminal.

⟨*nts-name*⟩ ::= ⟨*annotations*⟩ **nts** ⟨*idn*⟩ **;**
⟨*init*⟩ ::= **init** ⟨*formula*⟩ **;**
⟨*nts-list*⟩ ::= ⟨*nts-basic*⟩ ⟨*nts-list*⟩ | ε
⟨*nts*⟩ ::= ⟨*nts-name*⟩ ⟨*decl-glob*⟩ ⟨*init*⟩ ⟨*instances*⟩ ⟨*nts-list*⟩

1.7.1 Lamport's Bakery Protocol

```
nts bakery;

// global declarations
par N : int;
choose[N] : int;
num[N] : int;
CS[N] : bool;

// initial condition on global variables
init   forall i :  int[0,N-1] .  (choose[i]=0 and
       num[i]=0 and not CS[i]) and N>0;

// note that the order of entry points determines their tids:
// tid of bakery threads will be in range [0...N-1] and
// tid of the monitor thread will be N
instances bakery[N], monitor[1];

// bakery threads
bakery {
  initial s1;
  s1 -> s2 { lock(tid) }
```

```
  s2 -> s1 { unlock(tid) }
}
// monitor thread
monitor {
  initial si;
  error se;
  si -> se { exists i,j :  int[0,N-1] .  (i!=j and
             CS[i] and CS[j]) and havoc() }
}
lock {
  in i :  int;
  max :  int, j :  int;
  initial s1;
  final s5;

  // set max
  s1 -> s2 { choose'[i] = 1 and max'=0 and j'=0
             and havoc(max,choose,j) }
  s2 -> s2 { j<N and max >= num[j] and j'=j+1
             and havoc(j) }
  s2 -> s2 { j<N and max < num[j] and max'=num[j] and j'=j+1
             and havoc(max,j) }
  s2 -> s3 { j=N and choose'[i] = 0 and j'=0
             and havoc(choose,j) }

  // wait for entering the critical section
  s3 -> s3 { j<N and choose[j] != 0 and havoc() }
  s3 -> s4 { j<N and choose[j] = 0 and havoc() }
  s4 -> s4 { num[j]!=0 and (num[j]<num[i] or
             num[j]=num[i] and j<i) and havoc() }
  s4 -> s3 { not (num[j]!=0 and (num[j]<num[i] or
             num[j]=num[i] and j<i)) and j'=j+1 and havoc(j) }

  // enter the critical section
  s3 -> s5 { j=N and CS'[i]=true and havoc(CS) }
}
unlock {
  in i :  int;
  initial s1;
  final s2;
```

```
  // leave the critical section
  s1 -> s2 { CS'[i]=false and num'[i]=0 and havoc(CS,num) }
}
```

1.8 Annotations

Some syntactical elements of the NTS can be annotated. Namely, the global NTS, its subsystems, control states, transitions, and variables. In the case of global systems, subsystems, transitions, and variables, the annotation is placed before their definition (see sections 1.2.2, 1.4, and 1.7). In order to allow annotations of control states, the ⟨*statesdecl*⟩ declaration block is introduced (see Section 1.4), which contains a list of (annotated) control states. Note that such (annotated) declaration can be partial, in other words, not all the control states used later in a definition of transitions have to appear there. An annotation can be of type *int*, *real*, *bool*, *string*, or *formula*. The formal syntax for annotations follows:

⟨*string*⟩ ::= **”** ⟨*PRINTABLE-CHAR*⟩* **”**
⟨*a-type-val*⟩ ::= **int :** ⟨*sign*⟩ ⟨*numeral*⟩
| **real :** ⟨*sign*⟩ ⟨*decimal*⟩
| **bool :** ⟨*boolean*⟩
| **string :** ⟨*string*⟩
| **formula :** ⟨*formula*⟩
⟨*annotation*⟩ ::= **@** ⟨*idn*⟩ **:** ⟨*a-type-val*⟩ **;**
⟨*annotations*⟩ ::= ⟨*annotations*⟩ ⟨*annotation*⟩ | ε

The following example illustrates the use of annotations:

```
@inv:formula:  x=0 and x<=11; // annotation of global NTS
@line:int:101; @col:int:5; // annotation of global variable x
x :  int;
init x=0;
@inv:formula:  x<=11; // subsystem annotation
main {
  @line:int:101; @col:int:15; // annotation of local variable y
  y :  int;
  states // partial declaration of control states
     @line:int:5; // annotation of the control state s1
     s1
  ;
  initial s1;
```

```
  final s2;
  s1 -> s1 @line:int:105; @col:int:1; // annotation of a transition
           { x<0 and x'=x+1 }
  s1 -> s2 { x=0 and x'=x }
  s2 -> s2 { x<=10 and x'=x+1 }
}
```

Chapter 2

Numerical Transition Systems

In the following, the symbols $\mathbb{B}$, $\mathbb{N}$, $\mathbb{N}_+$, $\mathbb{Z}$ and $\mathbb{R}$ are used to denote the sets of booleans, natural, strictly positive natural, integer, rational and real numbers, respectively.

Let $\mathbb{D}$ be a *data domain*, or simply domain. Each domain has an associated first-order logical theory $\mathcal{T}_\mathbb{D} = \langle \mathbb{D}, \mathcal{X}, \{P_i\}_{i=1}^n, \{f_i\}_{i=1}^m \rangle$ consisting of:

- a (possibly infinite countable) set of variables $\mathcal{X}$
- a (possibly infinite countable) set of predicate symbols $\{P_i\}_{i=1}^n$
- a (possibly infinite countable) set of function symbols $\{f_i\}_{i=1}^m$

all ranging over the universe $\mathbb{D}$. As usual, we consider constants to be functions of zero arity.

The *alphabet* of $\mathcal{T}_\mathbb{D}$ is the set of all variables, predicate and function symbols of $\mathcal{T}_\mathbb{D}$, together with the classical first-order logic connectives: $\vee$, $\wedge$, $\neg$, $\exists$, $\forall$ and the equality sign $=$. A *term* of $\mathcal{T}_\mathbb{D}$ is either a variable $x \in \mathcal{X}$, or a function symbol $f(t_1, \ldots, t_n)$ applied to a number of terms equal to its arity. An atomic proposition is either $t_1 = t_2$, or $p(t_1, \ldots, t_n)$ for a predicate symbol of arity n and terms $t_{1,\ldots,n}$. The *language* of $\mathcal{T}_\mathbb{D}$ is the set of all syntactically valid first-order formulae build from atomic propositions, using the first-order connectives.

Each atomic proposition π in the language of $\mathcal{T}_\mathbb{D}$ has an attached *guard*, denoted $\mathcal{G}(\pi)$, which is a formula also in the language of $\mathcal{T}_\mathbb{D}$. Unless specified, we assume the guard to be true. The guard of a first-order formula is defined inductively:

- $\mathcal{G}(\varphi_1 \wedge \varphi_2) \equiv \mathcal{G}(\varphi_1 \wedge \varphi_2) \equiv \mathcal{G}(\varphi_1) \wedge \mathcal{G}(\varphi_2)$
- $\mathcal{G}(\exists x \,.\, \varphi) \equiv \mathcal{G}(\forall x \,.\, \varphi) \equiv \mathcal{G}(\neg\varphi) \equiv \mathcal{G}(\varphi)$

Given two (or more) domains $\mathbb{D}_1$ and $\mathbb{D}_2$ with associated first-order theories $\mathcal{T}_{\mathbb{D}_1} = \langle \mathbb{D}_1, \{R_i\}_{i=1}^{n_1}, \{f_i\}_{i=1}^{m_1} \rangle$ and $\mathcal{T}_{\mathbb{D}_2} = \langle \mathbb{D}_2, \{R_i\}_{i=1}^{n_2}, \{f_i\}_{i=1}^{m_2} \rangle$, the disjoint union of $\mathcal{T}_{\mathbb{D}_1}$ and $\mathcal{T}_{\mathbb{D}_2}$ is defined as:

$$\mathcal{T}_{\mathbb{D}_1 \uplus \mathbb{D}_2} = \langle \mathbb{D}_1 \uplus \mathbb{D}_2, \mathcal{X}_1 \uplus \mathcal{X}_2, \{P_i\}_{i=1}^{n_1} \uplus \{P_i\}_{i=1}^{n_2}, \{f_i\}_{i=1}^{m_1} \uplus \{f_i\}_{i=1}^{m_2} \rangle$$

An atomic proposition of $\mathcal{T}_{\mathbb{D}_1 \uplus \mathbb{D}_2}$ is either an atomic proposition of $\mathcal{T}_{\mathbb{D}_1}$ or of $\mathcal{T}_{\mathbb{D}_2}$ i.e., without mixing variables, predicates or function symbols from both theories. The language of $\mathcal{T}_{\mathbb{D}_1 \uplus \mathbb{D}_2}$ is the set of all syntactically valid first-order formulae build from the atomic propositions of $\mathcal{T}_{\mathbb{D}_1 \uplus \mathbb{D}_2}$.

Given a finite set of domains $\mathbb{D}_1, \ldots, \mathbb{D}_k$, a multi-domain *numerical transition system* (NTS) is a tuple

$$S = \langle \{\mathbb{D}_i\}_{i=1}^k, \{X_i\}_{i=1}^k, \{\mathbf{p}_i\}_{i=1}^k, \{\mathbf{x}_i^{in}\}_{i=1}^k, \{\mathbf{x}_i^{out}\}_{i=1}^k, Q, I, F, E, \Delta \rangle$$

where, for all $i = 1, \ldots, k$:

- X_i is a set of *state variables* ranging over $\mathbb{D}_i$. W.l.o.g. we require that $X_i \cap X_j = \emptyset$, for all $i \neq j$.

  An *interpretation* of the state variables is a mapping $\nu_i : X_i \to \mathbb{D}_i$. We denote by X_i' the set of primed variables i.e., $\{x' \mid x \in X_i\}$, and by $\nu_i' : X_i' \to \mathbb{D}_i$ an interpretation of the primed variables.

  The set of *relations* over X_i is the set of formulae in the language of $\mathcal{T}_{\mathbb{D}_i}$ with free variables in the set $X_i \cup X_i'$, and is denoted as $\mathcal{R}_{\mathbb{D}_i}$. For any relation $R \in \mathcal{R}_{\mathbb{D}_i}$ we write

  $$\models R[\nu_i / X_i, \nu_i' / X_i']$$

  if the formula in which each variable $x \in X_i$ is substituted by $\nu_i(x)$, each $x_i' \in X_i'$ is substituted by $\nu_i'(x_i')$, and each predicate and function symbol is interpreted over $\mathbb{D}_i$, is logically true.

- $\mathbf{p}_i \subseteq X_i$ is a set of *parameters* i.e., variables whose values do not change during execution

- $\mathbf{x}_i^{in} \subseteq X_i$ is a (possibly empty) ordered sequence of *input variables*

- $\mathbf{x}_i^{out} \subseteq X_i$ is a (possibly empty) ordered sequence of *output variables*

- Q is a finite set of *control states*

- $I \subseteq Q$ is a non-empty set of *initial states*

- $F \subseteq Q$ is a (possibly empty) set of *final states*

- $E \subseteq Q$ is a (possibly empty) set of *error states*
- Δ is a set of *transition rules* of the form:

$$q \xrightarrow{R} q'$$

where $q, q' \in Q$ and $R \in \mathcal{T}_{\mathbb{D}_1 \uplus \ldots \uplus \mathbb{D}_k}$

We require that, for all transition rules $q \xrightarrow{R_1} q', \ldots, q \xrightarrow{R_n} q'$ with q and q' as the source and destination states, respectively, for no two $i, j \in \{1, \ldots, n\}$ it is the case that $R_i \Longrightarrow R_j$ is valid (a transition rule does not subsume another).

Additionally, for each transition relation R occurring in Δ, the following implication is assumed to be valid:

$$R \Longrightarrow \bigwedge_{i=1}^{k} \bigwedge_{p \in \mathbf{p}_i} p = p'$$

Moreover, we assume a transition rule $q \xrightarrow{\neg \mathcal{G}(R)} q_e$ to some error state $q_e \in E$, for each transition rule $q \xrightarrow{R} q'$.

A *configuration* of an NTS S is a tuple $(q, \nu_1, \ldots, \nu_k)$, where $q \in Q$ is a control state and $\nu_i : X_i \to \mathbb{D}_i$ are interpretations of the state variables. A configuration whose control state is from E (F) is called an error (final) configuration, respectively. A configuration $(q', \nu'_1, \ldots, \nu'_k)$ is an *immediate successor* of $(q, \nu_1, \ldots, \nu_k)$, denoted as

$$(q, \nu_1, \ldots, \nu_k) \Rightarrow (q', \nu'_1, \ldots, \nu'_k)$$

if and only if S has a transition rule $q \xrightarrow{R} q'$, where $R \in \mathcal{T}_{\mathbb{D}_1 \uplus \ldots \mathbb{D}_k}$ and

$$\models R[\nu_1/X_1, \ldots, \nu_k/X_k, \nu'_1/X'_1, \ldots, \nu'_k/X'_k]$$

Given two control states $q, q' \in Q$, *a run* ρ *of* A *from* q *to* q' is a finite sequence of configurations

$$\rho \ : \ c_1 \Rightarrow c_2 \Rightarrow \ldots c_m$$

where $c_1 = (q, \nu_1, \ldots, \nu_k)$ and $c_m = (q', \nu'_1, \ldots, \nu'_k)$ for some interpretations $\nu_1, \ldots, \nu_k$ and $\nu'_1, \ldots, \nu'_k$. A run where the endpoint is not specified is assumed to be infinite.

Let $c = (q, \nu_1, \ldots, \nu_k)$ be an arbitrary configuration of an NTS S and let $c \Rightarrow c' = (q', \nu'_1, \ldots, \nu'_k)$ and $c \Rightarrow c'' = (q'', \nu''_1, \ldots, \nu''_k)$:

- S is said to be *control deterministic* iffy $q' = q''$, independently of the choice of c, c' and c''.
- S is said to be *data deterministic* iff $\nu'_i = \nu''_i$, for all $i \in \{1,\ldots,k\}$, independently of the choice of c, c' and c''.
- S is said to be *deterministic* iff it is both control and data deterministic, and *non-deterministic* otherwise.

Let $S = \langle \{\mathbb{D}_i\}_{i=1}^k, \{X_i\}_{i=1}^k, \{\mathbf{p}\}_{i=1}^k, \{\mathbf{x}_i^{in}\}_{i=1}^k, \{\mathbf{x}_i^{out}\}_{i=1}^k, Q, I, F, E, \Delta \rangle$ be an NTS, and let $\Theta(X)$ be a formula in the language of $\mathcal{T}_{\mathbb{D}_1 \uplus \ldots \uplus \mathbb{D}_k}$ denoting *initial conditions*. A configuration $(q, \nu_1, \ldots, \nu_k)$ such that $q \in I$ and $\models \Theta[\nu_1/X_1, \ldots, \nu_k/X_k]$ is called a Θ-*initial configuration*. The following define the *verification problems* we consider for NTS.

Definition 1 (SAFETY) *S is said to be* safe *with respect to Θ if and only if there is no run starting in a Θ-initial configuration, and ending in an error configuration.*

Definition 2 (TERMINATION) *S is said to* terminate *with respect to Θ if and only if each run starting in a Θ-initial configuration eventually reaches a final configuration.*

2.1 Basic NTS

A *basic NTS* (BNTS) is an NTS over purely numeric domains. We distinguish between boolean $\mathbb{B}$, integer $\mathbb{Z}$ and real $\mathbb{R}$ domains. The domain theory is in this case the first order arithmetic of addition and multiplication i.e., $\langle \mathbb{D}, \leq, 0, 1, +, \cdot \rangle$, where $\mathbb{D} \in \{\mathbb{Z}, \mathbb{R}\}$ and $\leq$, 0, 1, $+$ and $\cdot$ are interpreted as usual. For booleans we write $false$, $true$, *or* and *and* instead of $0, 1, +$ and $\cdot$, respectively.

Notice that the domain of variables plays an important role in the semantics of the transition relation. For instance, the relation $x < 8 \wedge x' + 8 = x$ is satisfiable on $\mathbb{Z}$ but unsatisfiable on $\mathbb{N}$, whereas $2 \cdot x \cdot x = y \cdot y$ is satisfiable on $\mathbb{R}$ but unsatisfiable on $\mathbb{Z}$.

2.1.1 A Classification of BNTS (skip on first reading)

As shown by Minsky[1], the class of single-domain BNTS with $\mathbb{D}_1 \in \{\mathbb{N}, \mathbb{Z}\}$, $X_1 = \{x_1, x_2\}$ and transition rules of one of the forms, for $i = 1, 2$:

[1] M. Minsky. Computation: Finite and Infinite Machines. Prentice-Hall, 1967.

$$q \xrightarrow{x_i'=x_i+1} q' \quad \text{(increment)}$$
$$q \xrightarrow{x_i'=x_i-1} q' \quad \text{(decrement)}$$
$$q \xrightarrow{x_i=0} q' \quad \text{(zero test)}$$

has equivalent (Turing-complete) verification problems as the class of BNTS using unrestricted first-order transition rules. On the other hand, using only increment, decrement and zero tests results in complex and hard to understand system descriptions (as one needs a great number of control states). Therefore it seems reasonable to identify expressive subfragments of the first order arithmetic and classify BNTS according to the syntax of their transition rules.

Definition 3 (DIFFERENCE BOUNDS) *A formula $\phi(X)$ is a* difference bounds constraint *if it is equivalent to a finite conjunction of atomic propositions of the form $x-y \leq a$ where $x,y \in X$ and $a \in \mathbb{Z}$.*

Let $\mathcal{R}_{\mathbb{D}}^{db}(X,X')$ denote the class of difference bound relations over the variables in X. A BNTS using only difference bounds transition relations is called a *difference bounds (B)NTS.*

Definition 4 (OCTAGONS) *A formula $\phi(X)$ is an* octagonal constraint *if it is equivalent to a finite conjunction of atomic propositions of the form $\pm x \pm y \leq a$, $2x \leq b$, or $-2x \leq c$, where $a,b,c \in \mathbb{Z}$ and $x,y \in X$.*

The class of octagonal relations over the variables in X is denoted by $\mathcal{R}_{\mathbb{D}}^{oct}(X,X')$. A BNTS using only octagonal transition relations is called an *octagonal (B)NTS.*

Definition 5 (POLYHEDRA) *A formula $\phi(X)$ is a* polyhedron *if it is equivalent to a finite conjunction of atomic propositions of the form $\sum_{i=1}^{n} a_i x_i + b \geq 0$ where $x_1,\ldots,x_n \in X$ and $a_1,\ldots,a_n,b \in \mathbb{Z}$.*

The class of polyhedral relations over the variables in X is denoted by $\mathcal{R}_{\mathbb{D}}^{poly}(X,X')$. A BNTS using only polyhedral transition relations is called a *polyhedral (B)NTS.*

Definition 6 (PRESBURGER) *A formula $\phi(X)$ is a* Presburger *formula if it is equivalent to a conjunction of a polyhedron with a finite conjunction of atomic propositions of the form $c \mid \sum_{i=1}^{n} a_i x_i + b$ where $x_1,\ldots,x_n \in X$, $a_1,\ldots,a_n,b,c \in \mathbb{Z}$ and $\mid$ denotes integer division. A Presburger formula is interpreted only over $\langle \mathbb{N}, \leq, 0, 1, +, \cdot \rangle$ and $\langle \mathbb{Z}, \leq, 0, 1, +, \cdot \rangle$.*

The class of Presburger relations over the variables in X is denoted by $\mathcal{R}_{\mathbb{D}}^{presb}(X,X')$. A BNTS using only Presburger transition relations is called a *Presburger (B)NTS*.

We have the following hierarchy[2], which reflects the expressive power of different subfragments of the first-order arithmetic.

$$\mathcal{R}_{\mathbb{D}}^{db} \subsetneq \mathcal{R}_{\mathbb{D}}^{oct} \subsetneq \mathcal{R}_{\mathbb{D}}^{poly} \subsetneq \mathcal{R}_{\mathbb{D}}^{presb}$$

The last inclusion concerns only the case of $\{\mathbb{N},\mathbb{Z}\}$ domains. Also, when interpreted over $\mathbb{N}$ or $\mathbb{Z}$, all classes of BNTS defined above have equivalent (Turing-complete) decision problems (safety and termination).

2.2 Array NTS

Let $\mathbb{D}$ be a base domain, in our case, one of $\mathbb{B}$, $\mathbb{Z}$ or $\mathbb{R}$. For the time being, we consider only one-dimensional arrays of purely numeric types. The *array domain of* $\mathbb{D}$ is the set

$$\mathcal{A}_{\mathbb{D}} = \{a \,:\, [0\ldots N-1] \to \mathbb{D} \mid N \in \mathbb{N}_+\}$$

The alphabet of the theory $\mathcal{T}_{\mathcal{A}_{\mathbb{D}}}$ consists of the combined alphabet of $\mathbb{N}$, $\mathbb{D}$ and the following three function symbols:

- $size \,:\, \mathcal{A}_{\mathbb{D}} \to \mathbb{N}_+$, $size(a) = N$ if $a : [0,\ldots,N-1] \to \mathbb{D}$ for some $N \in \mathbb{N}_+$
- $read \,:\, \mathcal{A}_{\mathbb{D}} \times \mathbb{N}_+ \to \mathbb{D}$, $read(a,i) = a(i)$ if $1 \le i \le size(a)$, and undefined otherwise.
- $write \,:\, \mathcal{A}_{\mathbb{D}} \times \mathbb{N}_+ \times \mathbb{D} \to \mathcal{A}_{\mathbb{D}}$, $write(a,i,v) = a[i \leftarrow v]$ if $1 \le i \le size(a)$, and undefined otherwise.

Note that, since an array is a function, the theory of arrays is not a first-order theory stricto sensu. For this reason, we do not allow (second-order) quantification over array variables for the time being.

Given a multiple array assignment

$$a'[i_1,\ldots,i_k] = [e_1,\ldots,e_k]$$

its effect is as if the basic assignments were performed left to right, formally:

$$write(write(write(\ldots(write(a,i_1,e_1))\ldots),i_{k-1},e_{k-1}),i_k,e_k)$$

An *array NTS* (ANTS) is an NTS over three kinds of domains:

[2]This hierarchy is not just purely syntactic. For instance, $x-y \le 0 \wedge x-z \le 1 \wedge 2x-y-z \le 1$ is still a difference bounds constraint, equivalent to $x-y \le 0 \wedge x-z \le 1$, although the syntax is that of a polyhedron.

- $\mathbb{N}$ is the *index domain*. The set of index variables is denoted by I.
- $\mathbb{D}_{1,\ldots,k}$ are the *scalar domains*, namely $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$. We denote the sets of scalar variables by $V_{1,\ldots,k}$.
- $\mathcal{A}_{\mathbb{D}_{1,\ldots,k}}$ are the *array domains*. We denote the sets of array variables by $A_{1,\ldots,k}$.

The transition rules of S are of the form

$$q \xrightarrow{R} q'$$

where $R \in \mathcal{T}_{\mathbb{N} \uplus \mathbb{D}_1 \uplus \ldots \mathbb{D}_k \uplus \mathcal{A}_{\mathbb{D}_1} \uplus \ldots \uplus \mathcal{A}_{\mathbb{D}_k}}$. We formally restrict the occurrences of unprimed array variables a to read ($read(a,i)$) and write ($write(a,i,t)$) functions, and primed array variables to atomic propositions of the form $a' = write(a,i,t)$. If π is an atomic proposition containing an array read or write, let $\mathcal{G}(\pi) \equiv 1 \leq i \wedge i \leq size(a)$ i.e., we explicitly represent array out-of-bounds errors.

2.3 Hierarchical NTS

A *hierarchical NTS* (HNTS) $\mathcal{S}_{\langle m \rangle}$ over the domains $\mathbb{D}_{1,\ldots,k}$ is defined using a collection $S_1, \ldots, S_n$ of basic NTS over $\mathbb{D}_{1,\ldots,k}$, with a designated *main* NTS, denoted by $m \in \{1,\ldots,n\}$. Let

$$S_i = \langle \{\mathbb{D}_j\}_{j=1}^k, \{X_{i,j}\}_{j=1}^k, \{\mathbf{p}_{i,j}\}_{j=1}^k, \{\mathbf{x}_{i,j}^{in}\}_{j=1}^k, \{\mathbf{x}_{i,j}^{out}\}_{j=1}^k, Q_i, I_i, F_i, E_i, \Delta_i \rangle$$

for all $i \in \{1,2,\ldots,n\}$, such that, for any $j, \ell \in \{1,2,\ldots,n\}$, $j \neq \ell$ implies $Q_j \cap Q_\ell = \emptyset$. We have then

$$\mathcal{S}_{\langle m \rangle} = \big\langle \{\mathbb{D}_j\}_{j=1}^k, \{\bigcup_{i=1}^n X_{i,j}\}_{j=1}^k, \{\bigcup_{i=1}^n \mathbf{p}_{i,j}\}_{j=1}^k, \{\mathbf{x}_{m,j}^{in}\}_{j=1}^k, \{\mathbf{x}_{m,j}^{out}\}_{j=1}^k, \bigcup_{j=1}^k Q_j, I_m, F_m, \bigcup_{j=1}^k E_j, \bigcup_{i=1}^n \Delta_i \big\rangle$$

Let $X_j^g = \bigcap_{i=1}^n X_{i,j}$ denote the sets of *global* (common) variables, and $X_{i,j}^l = X_{i,j} \setminus X_j^g$ denote the sets of *local* variables, for all $j \in \{1,\ldots,k\}$. We explicitly require that, for any $i \neq i'$, $X_{i,j}^l \cap X_{i',j}^l = \emptyset$ i.e., local variables of component NTS are pairwise disjoint.

The transition rules of $\mathcal{S}_{\langle m \rangle}$ are of two kinds:

- $q \xrightarrow{R} q'$ (*internal rules*)

  where $q, q' \in Q_i$, for some $i \in \{1,\ldots,n\}$, and $R \in \mathcal{T}_{\mathbb{D}_1 \uplus \ldots \uplus \mathbb{D}_k}$

- $q \xrightarrow{S_j(\mathbf{t}_1,\ldots,\mathbf{t}_k,\mathbf{y}'_1,\ldots,\mathbf{y}'_k)} q'$ (*call-return rules*[3])

  where $q, q' \in Q_i$, for some $i \in \{1,\ldots,n\}$, and $\mathbf{t}_\ell$ are ordered sequences of terms over the variables $X_{i,\ell}$, such that $\|\mathbf{t}_\ell\| = \|\mathbf{x}^{in}_{j,\ell}\|$ and $\|\mathbf{y}'_\ell\| = \|\mathbf{x}^{out}_{j,\ell}\|$ for some $j \in \{1,\ldots,n\}$ and for all $\ell \in \{1,\ldots,k\}$.

Given base domains $\mathbb{D}_{1,\ldots,k}$ and the set of basic NTS $\{S_i\}_{i=1}^n$ above, the *frame domain* is defined as follows:

$$\mathcal{F}_{\mathbb{D}_{1,\ldots,k}} = \bigcup_{i=1}^{n} \left(\Delta_i \times (X^l_{i,1} \to \mathbb{D}_1) \times \ldots \times (X^l_{i,k} \to \mathbb{D}_k)\right)$$

Given a relation $R \in \mathcal{R}_{\mathbb{D}_1,\ldots,\mathbb{D}_k}$, interpretations of global variables $\gamma_j, \gamma'_j : X^g_j \to \mathbb{D}_j$, for all $j \in \{1,\ldots,k\}$, and frames $\phi = (\delta, \nu_1, \ldots, \nu_k)$, $\phi' = (\delta', \nu'_1, \ldots, \nu'_k)$, we write $\langle \Gamma, \phi, \Gamma', \phi' \rangle \models R$ for the following:

$$\models R[\gamma_1/X^g_1, \ldots, \gamma_k/X^g_k, \nu_1/X^l_1, \ldots, \nu_k/X^l_l, \gamma'_1/X^{g'}{}_1, \ldots, \gamma'_k/X^{g'}{}_k, \nu'_1/X^{l'}{}_1, \ldots, \nu'_k/X^{l'}{}_k]$$

where $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$ and $\Gamma' = \{\gamma'_1, \ldots, \gamma'_k\}$.

The *stack* domain is the set of sequences of frames

$$\mathcal{S}_{\mathbb{D}_{1,\ldots,k}} = (\mathcal{F}_{\mathbb{D}_{1,\ldots,k}})^*$$

The alphabet of the stack domain consists of three functions:

- $push : \mathcal{S}_{\mathbb{D}_{1,\ldots,k}} \times \mathcal{F}_{\mathbb{D}_{1,\ldots,k}} \to \mathcal{S}_{\mathbb{D}_{1,\ldots,k}}$ defined as $push(\sigma, \phi) = \sigma \circ \phi$, where $\circ$ denotes sequence concatenation.
- $pop : \mathcal{S}_{\mathbb{D}_{1,\ldots,k}} \to \mathcal{S}_{\mathbb{D}_{1,\ldots,k}}$ defined as $pop(\sigma \circ \phi) = \sigma$, and undefined if applied to the empty sequence.
- $top : \mathcal{S}_{\mathbb{D}_{1,\ldots,k}} \to \mathcal{F}_{\mathbb{D}_{1,\ldots,k}}$ defined as $top(\sigma \circ \phi) = \phi$, and undefined if applied to the empty sequence.

A configuration of a HNTS is a tuple $(q, \gamma_1, \ldots, \gamma_k, \sigma)$, where

- $q \in \bigcup_{i=1}^n Q_i$ is the current control state
- $\gamma_i : X^g_i \to \mathbb{D}_i$ are interpretations of global variables, for all $i \in \{1,\ldots,k\}$
- $\sigma \in \mathcal{S}_{\mathbb{D}_{1,\ldots,k}}$ is a stack

[3]For the sake of simplicity, we introduce transition rules that cannot be written using first-order logic. The semantics however can be fitted into logic defining a stack domain and push/pop functions.

The immediate successor relation is defined as follows. We have

$$(q, \gamma_1, \ldots, \gamma_k, \sigma) \Rightarrow (q', \gamma'_1, \ldots, \gamma'_k, \sigma')$$

if and only if either one of the following holds:

- $q \xrightarrow{R} q' \in \Delta$ and $\langle \Gamma, top(\sigma), \Gamma', top(\sigma') \rangle \models R$

- $\delta \equiv q \xrightarrow{S_j(\mathbf{t}_1,\ldots,\mathbf{t}_k,\mathbf{y}'_1,\ldots,\mathbf{y}'_k)} q'' \in \Delta_i$ for some $i \in \{1,\ldots,n\}$, $q' \in I_j$, $\gamma_\ell = \gamma'_\ell$, for all $\ell \in \{1,\ldots,k\}$ and $\sigma' = push(\sigma, \phi)$, where $\phi = (\delta, \nu_1, \ldots, \nu_k)$ is a stack frame such that:

$$\langle \Gamma, top(\sigma), \Gamma', \phi \rangle \models \bigwedge_{\ell=1}^{k} \bigwedge_{m=1}^{p} x'_{m,\ell} = t_{m,\ell}$$

  where $\mathbf{x}^{in}_{j,\ell} = \langle x_{1,\ell}, x_{2,\ell}, \ldots x_{p,\ell} \rangle$ and $\mathbf{t}_\ell = \langle t_{1,\ell}, t_{2,\ell}, \ldots t_{p,\ell} \rangle$ are ordered sequences, with equal number of elements.

- $\delta \equiv q'' \xrightarrow{S_j(\mathbf{t}_1,\ldots,\mathbf{t}_k,\mathbf{y}'_1,\ldots,\mathbf{y}'_k)} q' \in \Delta_i$ for some $i \in \{1,\ldots,n\}$, $q \in F_j$, $\gamma_\ell = \gamma'_\ell$, for all $\ell \in \{1,\ldots,k\}$, such that $top(\sigma) = (\delta, \nu_1, \ldots, \nu_k)$. Let $\sigma'' = pop(\sigma)$ and $top(\sigma'') = (\delta'', \nu''_1, \ldots, \nu''_k)$. Then $\phi = (\delta'', \nu'_1, \ldots, \nu'_k)$ is a frame such that:

$$\langle \Gamma, top(\sigma''), \Gamma', \phi \rangle \models \bigwedge_{\ell=1}^{k} \bigwedge_{x \in X^l_{i,\ell} \setminus \mathbf{y}_\ell} x' = x$$

  and

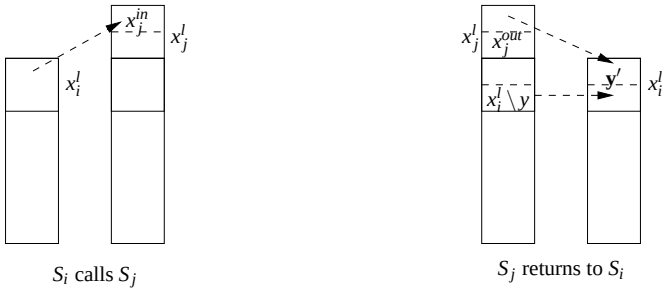
$$\langle \Gamma, top(\sigma), \Gamma', \phi \rangle \models \bigwedge_{\ell=1}^{k} \bigwedge_{m=1}^{p} x_{m,\ell} = y'_{m,\ell}$$

  where $\mathbf{x}^{out}_{j,\ell} = \langle x_{1,\ell}, x_{2,\ell}, \ldots x_{p,\ell} \rangle$ and $\mathbf{y}'_\ell = \langle y'_{1,\ell}, y'_{2,\ell}, \ldots y'_{p,\ell} \rangle$ are ordered sequences, with equal number of elements. Finally, let $\sigma' = push(pop(\sigma''), \phi)$.

2.4 Parallel NTS

A *parallel NTS* PNTS $\mathcal{S}[M]$ is defined as a collection of (possibly hierarchic) NTS $S_1, \ldots, S_n$, over the domains $\mathbb{D}_1, \ldots, \mathbb{D}_k$, together with a *multiset of instances* $M : \{1,\ldots,n\} \to \mathbb{N}$. A configuration of $\mathcal{S}[M]$ is a tuple

$$C = \langle c_{1,1} \ldots c_{1,M(1)},\ c_{2,1} \ldots c_{2,M(2)},\ \ldots,\ c_{n,1} \ldots c_{n,M(n)} \rangle$$



where $c_{i,1},\ldots,c_{i,M(i)}$ are configurations of S_i, for all $i \in \{1,\ldots,n\}$.

Given a formula Θ in the language of $\mathcal{T}_{\mathbb{D}_1 \uplus \ldots \uplus \mathbb{D}_k}$, a Θ-initial configuration of $S[M]$ is a configuration whose $c_{i,j}$ components are all Θ-initial configurations of S_i, for all $i \in \{1,\ldots,n\}$. A final configuration is a configuration whose $c_{i,j}$ components are all final configurations of S_i, for all $i \in \{1,\ldots,n\}$. An error configuration is a configuration such that at least one $c_{i,j}$ component is an error configuration of S_i, for some $i \in \{1,\ldots,n\}$.

The transition relation is defined as follows. We have $C \Rightarrow C'$ if and only if C' is same as C except for exactly one component $i \in \{1,\ldots,n\},\ j \in \{1,\ldots,M(i)\}$, for which we have $C_{i,j} \Rightarrow_i C'_{i,j}$, where $\Rightarrow_i$ is the transition relation of S_i.

In a parallel NTS, all thread instances have access to a predefined variable **tid**. Globally, **tid** is defined as a partial function $tid(i,j) = \sum_{\ell=1}^{i-1} M(\ell) + j - 1$, if $1 \leq i \leq n$ and $1 \leq j \leq M(i)$, and undefined otherwise. Intuitively, $tid(i,j)$ is the local value of the variable **tid** for the instance (i,j) of the parallel system.