EINDHOVEN UNIVERSITY OF TECHNOLOGY

SLCO: The Simple Language of Communicating Objects

MANUAL

VERSION 2.0

Authors
Melroy VAN NIJNATTEN and
Anton WIJS

November 8, 2022

1 Introduction

The Simple Language of Communicating Objects (SLCO) framework has been created to simplify the development of complex, parallel and multi-component software through a model-driven approach [1]. The primary building blocks of the language are classes, and the concurrent state machines and statements contained therein. The SLCO framework is built around the Domain-Specific Language (DSL) SLCO, which can be used to specify the behavior of a concurrent software system by means of a model. This model can be converted to other artifacts through model-to-model transformations, using the framework.

The focus of this manual is on the DSL, as opposed to the framework. In particular, we focus on those aspects of the language that are supported by the state space exploration tool GPUEXPLORE (version 3.0), which is one of the tools of the framework. Adding support for the few aspects of the language that are not yet supported by GPUEXPLORE is planned for the near future.

In the SLCO framework, the DSL has been defined for PYTHON, using the TEXTX [2] meta-language. A model-to-code transformation from SLCO to CUDA code for GPUEXPLORE has been defined with the JINJA2[1] template language. This manual is structured as follows. First, an introduction to TEXTX's grammar will be given in Section A. In the next section, the SLCO 2.0 language is introduced. After that, Section 4 addresses the semantics of SLCO. Finally, Section 3 contains several SLCO models through which the language is demonstrated. In Appendix A, the TEXTX language is explained.

2 The SLCO 2.0 Language – Syntax

The SLCO 2.0 language has been designed to model systems that consist of a collection of concurrent processes at a convenient level of abstraction [1]. The definition of an SLCO model consists of a class definition, together with an instantiation of this class as an object. The class defines a finite number of state machines that run concurrently, together with a collection of member variables associated with the class. State machines consist of a finite collection of states, the guarded transitions between those states, and a set of local variables. The transitions express the desired behavior of the program through atomic statements that need to be executed upon firing.

The language allows for the state machines in an object to interact via the variables of that object. State machines are not allowed to access each other's local variables.

The remainder of this section focusses on giving an in-depth description of all of the components provided by the SLCO 2.0 language that are supported by GPUEXPLORE version 3.0.

2.1 Model

```
SLCOModel:
  'model' name=NID '{'
    ('classes' classes=Class)
    ('objects' objects=Object)
  '}'
;

NID: !Keyword ID;
```

Listing 1: The SLCO model declaration syntax defined through the TEXTX grammar language.

[1]The documentation of JINJA2 is given at http://jinja.pocoo.org

The syntax definition of the root component of the SLCO model is given in Listing 1. A model declaration starts with the `model` keyword that is succeeded by the model's name, followed by curly brackets that contain a class and an object in that order, preceded by their respective keyword. The aforementioned components within the brackets will be elaborated upon further in the next sections.

The model name is defined to be a `NID`, which is a value of built-in type `ID` that is not equivalent to any of the keywords, with the exclusion of keywords being achieved through a negative lookahead. Hence, the name can be any common identifier consisting of letters, digits and underscores, under the condition that the chosen identifier does not match one of the keywords.

2.2 Classes

```
Class:
  name=NID '{'
    ('variables' variables*=Variable)?
    ('state machines' statemachines*=StateMachine)?
  '}'
;
```

Listing 2: The class declaration syntax defined through the TEXTX grammar language.

An SLCO class is at its core a grouping of concurrently running state machines that have access to the same set of member variables. Henceforth, variables at the class level will be referenced to as class variables.

The syntax of a class is given in Listing 2. Note that there is no keyword for defining a class. The class's name is immediately followed by curly brackets, which contains a finite number of variables and concurrent state machines in the given order, with each member being optional. Note that all members needs to be preceded by their associated keyword. Variable and state machine declarations are described in Sections 2.3 and 2.4, respectively. The name of a class and the variable names used within the scope of that class are all required to be unique.

2.3 Variables

```
Variable:
  (type=Type?) name=NID
  (':=' (
    defvalue=INT | defvalue=BOOL | ('['(defvalues+=INT[','] | defvalues+=BOOL[','])']')
  ))?
;

Type:
  (base='Integer' | base='Boolean' | base='Byte') ('[' size=INT ']')?
;
```

Listing 3: The variable declaration syntax defined through the TEXTX grammar language.

The syntax of a variable is given in Listing 3. A variable is defined to be of a certain type followed by the variable's name, with the supported types being `Integer`, `Boolean`, unsigned `Byte` and arrays of the aforementioned types. The variable can be turned into an array by stating the length of the array between brackets after declaring the base type. The initial value of a variable can be declared through an assignment and needs to correspond with the given type and array length if applicable. Initial values of an array variable need to be comma separated and enclosed by square brackets. Assigning an initial value to a variable is optional–appropriate default values are assigned automatically to variables without a given initial value. Moreover, the initial values passed to an object instantiation take precedence over the initial values assigned during the variable declarations.

2.4 State Machines

```
StateMachine:
  name=NID '{'
    ('variables' variables*=Variable)?
    'initial' initialstate=State
    ('states' states*=State)?
    ('transitions' transitions*=Transition)?
  '}'
;

State: name=NID;
```

Listing 4: The state machine declaration syntax defined through the TEXTX grammar language.

State machines are non-deterministic finite state machines that use guarded and prioritized transitions between states, i.e., each state can have zero, one or more transitions, each to an arbitrary target state. A transition is considered *active* if its source state is equal to the current state of the state machine. As a state may have more than one transition, multiple transitions can be simultaneously active. State machines communicate with each other through the class variables provided by the parent object.

The syntax of a state machine is given in Listing 4. Similarly to a class definition, the declaration of a state machine is not preceded by a keyword. A state machine declaration starts with a unique name within the scope of the parent class. The name is followed by curly brackets containing the state machine's local variables, an initial state, the additional states, and transitions between states in the described order. All members are optional except for the initial state. Note that the initial state does not need to be added to the list of additional states. Variable and transition declarations are described in Sections 2.3 and 2.5, respectively. The representation of a state is the name associated with the state in question. The variable and state names are all required to be unique within the scope of the state machine.

2.5 Transitions

```
Transition:
  (priority=INT ':')?
  ((source=[State] '->' target=[State]) | ('from' source=[State] 'to' target=[State]))
  ('{' statement=Statement (';')? '}')?
;
```

Listing 5: The transition declaration syntax defined through the TEXTX grammar language.

Transitions and the statements contained therein are used to describe the desired behavior of a program. The syntax of a transition is given in Listing 5. The transition definition starts with a priority followed by a colon, where the priority is defined to be an `Integer` value. A lower number indicates a higher priority. Defining a priority is optional–if no priority is given, the transition is automatically given the priority 0. Next, the names of the source and target states of the transition are defined. Two notations are supported to define the relation between the source and target states: one option is to connect the two states with an arrow (->), and the other uses the `from` and `to` keywords.

The optional body of the transition is enclosed by curly brackets, and contains one statement, optionally trailed by a semicolon. The available types of statements are introduced in Section 2.6. A transition can be *fired* if it is active and the statement in the transition's body is *enabled*. Firing a transition results in the system changing state to the target state of the transition, and the statement in the transition's body being executed.

Finally, the parallel execution of transitions is formalised using interleaving semantics, in which the SLCO framework's statements are atomic. No finer-grained interleaving is allowed [1]. More on the semantics of SLCO models in Section 4.

2.6 Statements

```
Statement:
  (Composite | Assignment | Expression )
;
```

Listing 6: The statement declaration syntax defined through the TEXTX grammar language.

SLCO offers several types of statements, namely (Boolean) expressions, assignments, and composites. Boolean expressions are statements that can be evaluated, assignments are used to alter variable values, and composites group an optional Boolean expression and zero or more assignments into an atomic operation. The different types of statements are discussed in more detail in Sections 2.6.1, 2.6.2 and 2.6.3, respectively.

2.6.1 (Boolean) Expressions

Expressions can be used to group one or more constant values, variables and operators into a numerical or Boolean statement of which the value can be evaluated. The supported constant values are of the types integer and Boolean. Both class and local variables, including ones of the array variety, are valid targets to be referenced within an expression, as long as they are in the scope of the state machine to which the expression belongs.

The available unary operators are the logical negation (`not`), numerical negation (`-`) and parentheses (`()`) operators. Additionally, the following binary operators are included: addition (+), subtraction (-), multiplication (∗), integer division (/), integer modulo (%), exponentiation (∗∗), equality (`==`, `!=`, `<>`), relation (<=, <, >, >=), logical conjunction (`and`, `&&`), logical disjunction (`or`, `||`) and logical exclusive disjunction (`xor`). The order of precedence for the operators is given in Table 1.

Expressions are used as a sub-statement within assignments and composites–these types of statements are discussed further in Sections 2.6.2 and 2.6.3, respectively.

A Boolean expression as a statement by itself is *enabled* iff it evaluates to `true`.

Operator	**Description**
`[]`	Array subscriptions
`()`	Parentheses
`not`, `+`, `-`	Negation, Unary Plus, Unary Minus
`**`	Exponentiation
`*`, `/`, `%`	Multiplication, Division, Modulo
`+`, `-`	Addition, Subtraction
`=` , `!=`, `<>`, `<`, `<=`, `>=`, `>`	Equality, Relational
`or`, `||`, `xor`, `and`, `&&`	(Exclusive) Disjunction, Conjunction

Table 1: The order of precedence of operators used in SLCO. An operator precedes another if it is placed higher up in the table. Operators located within the same table row have the same precedence level and hence have the same significance.

2.6.2 Assignments

```
Assignment:
  left=VariableRef ':=' right=Expression
;

VariableRef:
  var=[Variable] ('[' index=Expression ']')?
;
```

Listing 7: The assignment declaration syntax defined through the TEXTX grammar language.

Assignments can be used to alter the values associated with variables and are used as a sub-component of composites, which are described in the following section. The syntax of an assignment is given in Listing 7. The left-hand side component of an assignment is a reference to the target variable by name (including an index if the variable is an array, enclosed by square brackets), followed by the assignment operator (`:=`), and finalized by an expression as the right-hand side component describing the desired value to be assigned to the target variable. Note that both local and class variables are valid targets within an assignment, as long as those variables are within the scope of the state machine to which the assignment belongs.

An assignment is *always* enabled.

2.6.3 Composites

```
Composite:
  '[' (guard=Expression ';')? assignments*=Assignment[';'] ']'
;
```

Listing 8: The composite declaration syntax defined through the TEXTX grammar language.

In SLCO, statements are defined to be atomic entities. However, there exist use cases in which atomicity at the level of a single Boolean expression or assignment is not sufficient to attain the desired result. Take for example a transition that assigns a value `v` to an array variable `A` at index `i`, with `A` and `i` both defined at the class level, but only if `i` is within the bounds of the array. To achieve this, two statements are required–a guard expression that checks if `i` is within the array bounds, and an assignment that assigns `v` to `A[i]`. Yet, there are no guarantees that `i` remained unaltered between the evaluation of the guard and execution of the assignment. Recall that the transitions of different state machines are running in parallel using interleaving semantics. Due to this, another assignment to `i` can take place between the range check and the execution of the assignment. Thus, the desired behavior cannot be attained unless the entire operation is made atomic.

The concept of composite statements has been introduced to allow for a collection of statements to be grouped into one overarching atomic entity. A composite statement consists of an optional Boolean expression that is followed by zero or more assignments. The syntax of a composite statement is given in Listing 8. The body of the composite is to be enclosed by square brackets and contains all of the statements that are part of the structure, each of them separated by a semicolon. A trailing semicolon needs to be included if only a guard expression is given. A composite statement `[x := u; ...]` can be interpreted as starting with the Boolean expression `true`, *i.e.*, `[true; x := u; ...]`.

When a composite statement is fired, the sub-statements inside it are executed in the specified sequence.

A composite statement is *enabled* iff its first sub-statement is enabled.

2.7 Objects

```
Object:
  name=NID ':' type=[Class] '(' assignments*=Initialisation[','] ')'
;

Initialisation:
  left=[Variable] ':=' (right=INT | right=BOOL | ('[' (rights+=INT[','] | rights+=BOOL[','
    ]) ']'))
;
```

Listing 9: The object declaration syntax defined through the TEXTX grammar language.

The syntax of an object is given in Listing 9 and consists of two parts: the definition of an object, and the definition of variable instantiations for variables included within the target class of the object.

First, we discuss the definition of an object. An object definition consists of a unique name for the object itself and the name of the class that is being instantiated. Additionally, the initial values of the variables used within the class can be defined between the brackets following the class name in a comma separated list. The inclusion of initial values for variables is optional–variables that retain their default value can be excluded from the list. On top of that, initial values given during the variable declaration itself are overwritten by initial values given in the object initialization.

The syntax for initializing variables within the object definition is defined as follows. An initialization is depicted as an assignment, with the left hand-side referring to the target variable by name, and the right-hand side depicting the desired target value for the variable in question. The assigned value needs to be a singular `Integer`, `Boolean`, or, in case the variable refers to an array, a comma separated list of the aforementioned types of the appropriate length, enclosed by square brackets.

3 Concrete Examples

The SLCO 2.0 language syntax is demonstrated in this section via a pair of examples. The first model is discussed in Section 3.1 and depicts the behavior of an elevator. Secondly, a model is introduced in Section 3.2 that depicts the puzzle game *Toads and Frogs*.

3.1 Elevator

In this section, an `Elevator` model is introduced, which depicts an elevator system servicing four floors. The model, given in Listing 10, originally stems from the BEEM benchmark suite [4], and has been translated to SLCO. The model is named `elevator2.1.slco`, and variants of it are included in the artifact of GPUEXPLORE. The model consists of the state machines `cabin`, `environment` and `controller`, with the four class variables `req`, `t`, `p` and `v` (line 4). In the model, the array variable `req` of length `n` is used to track which floors have requested the cabin, `t` is the floor number the controller directs the cabin to, `p` is the floor the cabin is currently at, and lastly, `v` is a byte that tracks if the cabin is currently executing the controller's movement instruction. The state machines contained within the model have been visualized in Figure 1. Each state machine fulfills a specific task:

- The `cabin` state machine (lines 6-14) has the states `idle`, `move` and `open`, with the model itself depicting the movement of the elevator cabin. The cabin starts in an `idle` state, and remains there until the elevator is instructed to move. In the `move` state, the cabin moves to the correct floor by moving up or down one step at a time.

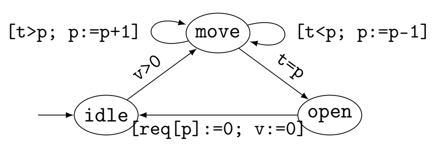
```
 1  model Elevator {
 2    classes
 3      GlobalClass {
 4        variables Byte[4] req Integer t Integer p Byte v
 5        state machines
 6          cabin {
 7            initial idle states mov open
 8            transitions
 9              from idle to mov { v>0 }
10              from mov to open { t=p }
11              from mov to mov { [t<p; p:=p-1] }
12              from mov to mov { [t>p; p:=p+1] }
13              from open to idle { [req[p]:=0; v:=0] }
14          }
15          environment {
16            initial read states
17            transitions
18              from read to read { [req[0]=0; req[0]:=1] }
19              from read to read { [req[1]=0; req[1]:=1] }
20              from read to read { [req[2]=0; req[2]:=1] }
21              from read to read { [req[3]=0; req[3]:=1] }
22          }
23          controller {
24            variables Byte ldir
25            initial wait states work done
26            transitions
27              from wait to work { [v=0; t:=t+(2*ldir)-1] }
28              from work to wait { [t<0 or t=4; ldir:=1-ldir] }
29              from work to done { t>=0 and t<4 and req[t]=1 }
30              from work to work { [t>=0 and t<4 and req[t]=0; t:=t+(2*ldir)-1] }
31              from done to wait { [v:=1] }
32          }
33      }
34    objects
35      globalObject: GlobalClass()
36  }
```

Listing 10: A concrete model declared with the SLCO 2.0 language syntax depicting an elevator.

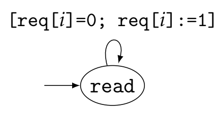
Once the right floor is reached, the cabin enters the `open` state, after which it proceeds to reset both `v` and `req[p]`, followed by a transition back to the `idle`. Observe that the `cabin` only has deterministic behavior in state `mov`–only one transition can be active at any time due to the chosen guard statements.

- The `environment` state machine (lines 15-22) only has a single state `read`, and has the task of modeling the act of calling an elevator to a certain position. Internally, state `read` has a separate transition for each floor $i \in [0, 4)$ back to itself, in which the associated value of `req[`i`]` is set to one if this was not already done. The `environment` state machine randomly picks a floor to call the cabin to, and is hence behaviorally completely non-deterministic.

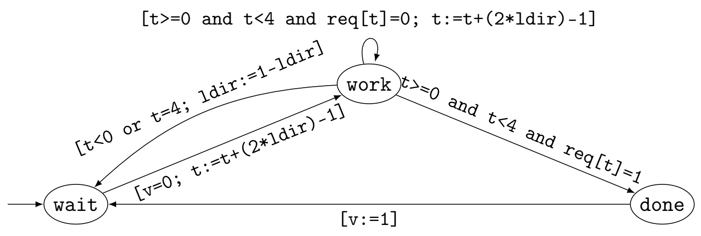
- The `controller` state machine (lines 23-32) has the states `wait`, `work` and `done`, and instructs the elevator cabin's movements. The controller starts in the `wait` state and remains there until `v` becomes zero, after which it moves to the `work` state. In the `work` state, the controller increments or decrements the value of `t` until a floor is encountered that has a cabin request. The direction of the search is dictated by the variable `ldir`, which is a local variable (line 24) that is zero when searching up, and one when searching down. The search direction is swapped whenever the cabin reaches the top or bottom floor. The state machine proceeds to the `done` state after finding an open cabin request, and subsequently moves back to the `wait` state after enabling v. Observe that, similarly to state `mov` in state machine `cabin`, the `controller` exclusively displays deterministic behavior in state `work` due to the chosen guard statements.



(a) State Machine `cabin`.



(b) State Machine `environment`, with a separate transition for all $i \in [0, 4)$.


[t>=0 and t<4 and req[t]=0; t:=t+(2*ldir)-1]
[t<0 or t=4; ldir:=1-ldir]
work
[v=0; t:=t+(2*ldir)-1]
t>=0 and t<4 and req[t]=1
wait
done
[v:=1]

(c) State Machine `controller`.

Figure 1: Visual depiction of the `Elevator` SLCO model consisting of the three state machines `cabin`, `environment` and `controller` with the class variables `req`, `t`, `p` and `v`. Additionally, the `controller` state machine has a local variable `ldir`.

3.2 Toads and Frogs

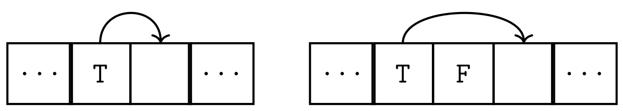
The second example depicts a single player variant of the puzzle game named *Toads and Frogs* [3]. A brief introduction to the game and associated rules is provided in Section 3.2.1, followed by a simulation of the game in the shape of an SLCO model in Section 3.2.2.

3.2.1 Game Description

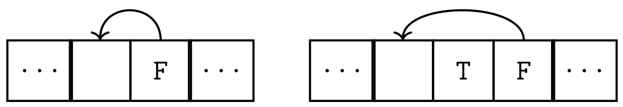
T	T	T	T		F	F	F	F

Figure 2: The initial state of the *Toads and Frogs* playing board $(n = 4)$. The board has an empty cell at the center flanked to the left by four toads (T) and to the right by four frogs (F).

The game is played on a playing board represented by an array of $n \cdot 2 + 1$ cells. There are n toads (depicted by the value T), n frogs (depicted by the value F), and one empty cell on the board. The initial state of the board is depicted in Figure 2: all toads and frogs are at the left and right side of the board, respectively, with the empty cell being at the center position $n + 1$.



(a) Legal moves for toads (`T`).



(b) Legal moves for frogs (`F`).

Figure 3: The legal moves for toads and frogs in the *Toads and Frogs* puzzle game.

The movement of the toads and frogs is restricted to the moves presented in Figure 3. Toads are forced to hop to the right, while frogs are allowed only to hop to the left. A toad may hop to an empty cell that is located immediately to its right. Additionally, a toad may hop over a frog that is directly to its right if the target frog's neighbor is an empty cell–however, it is not allowed for a toad to hop over another toad. Analogously, a frog may hop to an open spot that is located directly to its left. Similarly, a frog can hop over a toad that is directly to its left if the target toad's neighbor is an empty cell. Any moves not mentioned above are deemed illegal.

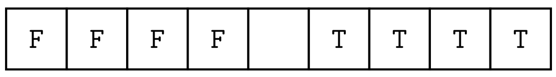


Figure 4: The state that needs to be reached for a game of *Toads and Frogs* to be won $(n = 4)$.

The end goal of the game is to mirror the playing field, i.e., the game is won if all toads are on the right side, and all frogs on the left side of the board, as depicted in Figure 4. Conversely, a game is considered lost if both toads and frogs have no legal moves remaining. In the single player variant of the game, the turns of the toads and frogs do not have to alternate–instead, the toads and frogs need to essentially work together to achieve a successful outcome.

3.2.2 Model Representation

The SLCO model `ToadsAndFrogs` depicting the *Toads and Frogs* puzzle game for $n = 4$ is given in Listing 11. The model has been designed in such a manner that the program is continuous, i.e., once the game is over, the board will be reset such that the game can be restarted. The model consists of the state machines `toad`, `frog` and `control`, with the four class variables `y`, `tmin`, `tmax` and `a` (lines 5-6). The variable `y` contains the index of the empty cell on the game board. The variables `tmin` and `fmax` track the indices of the leftmost and rightmost toad and frog, respectively, such that the win condition can be evaluated efficiently. Lastly, the array `a` is a representation of the playing board, and encodes the empty cell as `0`, a frog as `1`, and a toad as `2`. The initial values assigned to the variables conform to the situation sketched in Figure 2. In the model, each state machine fulfills a specific role within the game:

- The state machine `toad` (lines 8-15) performs the movement actions that can be taken by a toad, and only contains the state `q`, which is marked as the initial state. Additionally, the state machine has four transitions. The first two transitions move a toad to a neighboring empty cell to the left, with the latter two executing a hop to the left over a frog into an empty cell. Each type of movement is represented by two transitions, with the difference being in the target subjects–the first occurrence

```
1  model ToadsAndFrogs {
2    classes
3      GlobalClass {
4        variables
5          Integer y:=4 Integer tmin:=0 Integer fmax:=8
6          Integer[9] a:=[1,1,1,1,0,2,2,2,2]
7        state machines
8          toad {
9            initial q
10           transitions
11             q -> q {[y>0 and tmin!=y-1 and a[y-1]=1; a[y]:=1; y:=y-1; a[y]:=0]}
12             q -> q {[y>0 and tmin=y-1 and a[y-1]=1; a[y]:=1; tmin:=y; y:=y-1; a[y]:=0]}
13             q -> q {[y>1 and tmin!=y-2 and a[y-2]=1 and a[y-1]=2; a[y]:=1; y:=y-2; a[y]:=0]}
14             q -> q {[y>1 and tmin=y-2 and a[y-2]=1 and a[y-1]=2; a[y]:=1; tmin:=y; y:=y-2; a[y]:=0]}

15         }
16         frog {
17           initial q
18           transitions
19             q -> q {[y<8 and fmax!=y+1 and a[y+1]=2; a[y]:=2; y:=y+1; a[y]:=0]}
20             q -> q {[y<8 and fmax=y+1 and a[y+1]=2; a[y]:=2; fmax:=y; y:=y+1; a[y]:=0]}
21             q -> q {[y<7 and fmax!=y+2 and a[y+1]=1 and a[y+2]=2; a[y]:=2; y:=y+2; a[y]:=0]}
22             q -> q {[y<7 and fmax=y+2 and a[y+1]=1 and a[y+2]=2; a[y]:=2; fmax:=y; y:=y+2; a[y]:=0]}

23         }
24         control {
25           initial running states done success failure reset
26           transitions
27             running -> done {y=0 and a[y+1]=1 and a[y+2]=1}
28             running -> done {y=1 and a[y-1]=2 and a[y+1]=1 and a[y+2]=1}
29             running -> done {y=7 and a[y-2]=2 and a[y-1]=2 and a[y+1]=1}
30             running -> done {y=8 and a[y-2]=2 and a[y-1]=2}
31             running -> done {y>1 and y<7 and a[y-2]=2 and a[y-1]=2 and a[y+1]=1 and a[y+2]=1}
32             done -> success {tmin>y and fmax<y}
33             done -> failure {not (tmin>y and fmax<y)}
34             success -> reset
35             failure -> reset
36             reset -> running {[
37               y:=4; tmin:=0; fmax:=8; a[4]:=0;
38               a[0]:=1; a[1]:=1; a[2]:=1; a[3]:=1;
39               a[5]:=2; a[6]:=2; a[7]:=2; a[8]:=2
40             ]}
41         }
42     }
43   objects
44     globalObject : GlobalClass()
45 }
```

Listing 11: A concrete model declared with the SLCO 2.0 language syntax depicting the single-player variant of the puzzle game named *Toads and Frogs*.

(lines 11, 13) targets toads that are not the leftmost specimen, and vice versa for the last occurrence (lines 12, 14). The latter two transitions maintain the variable `tmin` in addition to executing the move itself, which ensures that `tmin` always points to the index of the leftmost toad.

- The state machine `frog` (lines 16-23) is analogous to the state machine `toad`, with the difference being that it focuses on the movements performed by the frogs instead. The first two transitions move a frog to a neighboring empty cell to the right, with the latter two executing a hop to the right into an empty cell over a toad. On top of that, the second and fourth transitions (lines 20, 22) update the variable `fmax` such that it is always ensured that its value points to the index of the rightmost frog.
- The state machine `control` (lines 24-41) focuses on managing the state of the game. The state machine has the initial state `running` and is supplemented by the states `done`, `success`, `failure` and `reset`. In the `running` state, it is checked whether the game has reached a point where neither the toad or the frog is able to execute any moves. In terms of logic, this implies that the two slots to the left of the empty

cell are frogs and the two slots to the right are toads (lines 27-31). If the condition holds, the state machine proceeds to the `done` state, in which it is verified if the game has been completed successfully or ended in a failure. A game is successful if all toads are to the right and all frogs are to the left of the empty cell. Given that `tmin` and `fmax` represent the leftmost and rightmost toad and frog, respectively, it must hence hold that `tmin > y and fmax < y` for the state machine to transition to the `success` state (line 32). Otherwise, a transition will be made to state `failure` (line 33). States `success` and `failure` make an empty transition to the state `reset` (lines 34-35). Lastly, the sole transition in the `reset` state will reset all the variables to their original value, such that the game can start over, signified by the transition returning to state `running` (lines 36-40).

4 The SLCO 2.0 Language – Semantics

In this section, we define and discuss the formal semantics of SLCO. In order to do so, we first mathematically define SLCO models and their components.

4.1 Objects and state machines

An SLCO model $\mathcal{N}$ corresponds with an SLCO *object*. Such an object consists of a finite number of state machines and a finite number of (object-local) variables.

Definition 1 (Model)**.** A *model* $\mathcal{N}$ is a tuple $(\mathcal{M}, \mathcal{V})$, where

- $\mathcal{M}$ is a finite set of state machines;
- $\mathcal{V}$ is a finite set of variables.

Definition 2 (State machine)**.** A *state machine* G is a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \hat{s}, \mathcal{V}_s)$, where

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of statements, with $\epsilon \in \mathcal{A}$ the empty statement.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathbb{N} \times \mathcal{S}$ is a transition relation, formalising transitions labelled with a statement between the states in $\mathcal{S}$. A transition has a priority in $\mathbb{N}$.
- $\hat{s} \in \mathcal{S}$ is the initial state.
- $\mathcal{V}_s$ is a finite set of (state machine-local) variables.

With $s \Rightarrow_i (a)\, s'$, we denote the fact that $(s, a, i, s) \in \mathcal{T}$, and in case the priority i is not relevant, we write $s \Rightarrow (a)\, s'$. The reflexive, transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.

Concerning the priority of transitions, we use the convention that 0 is the default priority, and for all priority values $p, q \in \mathbb{N}$, p is a higher priority than q iff $p < q$.

4.2 Well-formedness constraints

The definitions above are very general, and therefore allow the construction of inconsistent specifications. Therefore, next, we present a list of well-formedness constraints, that should be satisfied by an SLCO model $\mathcal{N} = (\mathcal{M}, \mathcal{V})$ in order to be consistent:

- In each state machine, all states are reachable from the initial state:
  $\forall (\mathcal{S}, \mathcal{A}, \mathcal{T}, \hat{s}, \mathcal{V}_s) \in \mathcal{M}, s \in \mathcal{S}. \hat{s} \Rightarrow^* s$
- Variables referred to in a statement of a state machine are either local to that state machine, or owned by the object containing that state machine:
  $\forall (\mathcal{S}, \mathcal{A}, \mathcal{T}, \hat{s}, \mathcal{V}_s) \in \mathcal{M}, a \in \mathcal{A}, x \in a. x \in \mathcal{V} \cup \mathcal{V}_s$

4.3 Semantics

We reason about the state of an SLCO model by means of a *situation*.

Definition 3 (Situation). Given an SLCO model $\mathcal{N} = (\mathcal{M}, \mathcal{V})$, with $\mathcal{M} = \{\mathcal{M}_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{T}_1, \hat{s}_1, \mathcal{V}_{s,1}), \ldots, \mathcal{M}_n = (\mathcal{S}_n, \mathcal{A}_n, \mathcal{T}_n, \hat{s}_n, \mathcal{V}_{s,n})\}$, *i.e.,* $\mathcal{N}$ has n state machines in total. Then, we define a *situation* as a tuple $\langle \sigma, s_1 || \ldots || s_n \rangle$, where

- σ is a total function mapping variables in $\mathcal{V} \cup \mathcal{V}_{s,1} \cup \ldots \cup \mathcal{V}_{s,n}$ to values of the appropriate types.
- $s_1 || \ldots || s_n$ indicates the *current state* of the combined state machines, where each s_i is the current state of state machine $\mathcal{M}_i$.

To evaluate statements, we use a function $\xi_\sigma : \mathcal{A} \to \mathbb{B}$ that maps a statement a to a Boolean value in the situation σ. The function reflects whether a statement is *enabled*. It is defined as follows, with $[e]_\sigma$ being the expression resulting from replacing every variable reference x in e by its corresponding value $\sigma(x)$.

$$\begin{aligned}
\xi_\sigma(\epsilon) &= \texttt{true} \\
\xi_\sigma(x := e) &= \texttt{true} \\
\xi_\sigma(e) &= [e]_\sigma \\
\xi_\sigma([e; x_1 := e_1; \ldots; x_m := e_m]) &= \xi_\sigma(e)
\end{aligned}$$

Next, we present the operational semantics of an SLCO model in the form of SOS rules. The function $\sigma[\xi(e)/x]$ is equal to σ, except for the fact that x has been updated to the value $\xi(e)$, *i.e.,* $\sigma(x) = \xi(e)$.

$$\text{silent step}\ \frac{s \Rightarrow_i (\epsilon)\, s' \land \neg(s \Rightarrow_j (a)\, s'' \land \xi_\sigma(a) \land j < i)}{\langle \sigma, s \rangle \xrightarrow{\tau} \langle \sigma, s' \rangle}$$

$$\text{assignment step}\ \frac{s \Rightarrow_i (x := e)\, s' \land \neg(s \Rightarrow_j (a)\, s'' \land \xi_\sigma(a) \land j < i)}{\langle \sigma, s \rangle \xrightarrow{x:=e} \langle \sigma[\xi(e)/x], s' \rangle}$$

$$\text{expression step}\ \frac{s \Rightarrow (e)\, s' \land \xi(e) \land \neg(s \Rightarrow_j (a)\, s'' \land \xi_\sigma(a) \land j < i)}{\langle \sigma, s \rangle \xrightarrow{e} \langle \sigma, s' \rangle}$$

$$\text{composite step}\ \frac{s \Rightarrow ([e; x_1 := e_1; \ldots; x_n := e_m])\, s' \land \xi(e) \land \neg(s \Rightarrow_j (a)\, s'' \land \xi_\sigma(a) \land j < i)}{\langle \sigma, s \rangle \xrightarrow{[e; x_1 := e_1; \ldots; x_m := e_m]} \langle \sigma[\xi(e_1)/x_1] \cdots [\xi(e_m)/x_m], s' \rangle}$$

$$\text{parallel-1}\ \frac{\langle \sigma, s \rangle \xrightarrow{a} \langle \sigma', s' \rangle}{\langle \sigma, s || t \rangle \xrightarrow{a} \langle \sigma', s' || t \rangle}$$

$$\text{parallel-2}\ \frac{\langle \sigma, s \rangle \xrightarrow{a} \langle \sigma', s' \rangle}{\langle \sigma, t || s \rangle \xrightarrow{a} \langle \sigma', t || s' \rangle}$$

The silent step rule expresses that if a state machine in state s has an active transition with the empty statement ϵ, and no transitions from s are enabled with a higher priority, then the system can perform a transition labelled τ in which the state machine changes state accordingly. In this case, σ is not updated.

The assignment step rule expresses in a similar way that an active transition from a state s can be fired when there are no transitions enabled from s with a higher priority. In this case, the state machine changes state accordingly and the effect of the assignment is taken into account by the update to σ.

The expression step rule expresses that a transition from s can be fired if its associated expression statement is enabled, and no transitions from s are enabled with a higher priority. The state machine changes state accordingly and the function σ is not updated.

The composite step rule expresses that when a transition from s with a composite statement can be fired, the state machine changes state accordingly, and the function σ is updated by taking the assignments in the composite statement into account in the specified sequence.

Finally, the rules parallel-1 and parallel-2 express how state machines behave in a parallel context, *i.e.*, combined with other state machines. This is straightforward, as state machines cannot synchronize.

References

[1] Sander de Putter, Anton Wijs, and Dan Zhang. The SLCO framework for verified, model-driven construction of component software. In Kyungmin Bae and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, October 10-12, 2018, Proceedings*, volume 11222 of *Lecture Notes in Computer Science*, pages 288–296. Springer, 2018.

[2] Igor Dejanović, Renata Vaderna, Gordana Milosavljević, and Željko Vuković. TextX: A python tool for domain-specific languages implementation. *Knowledge-Based Systems*, 115:1 – 4, 2017.

[3] Anany Levitin and Maria Levitin. *Algorithmic puzzles*, page 53. Oxford University Press, 2011.

[4] Radek Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN 2007*, volume 4595 of *LNCS*, pages 263–267, 2007.

A The TEXTX Grammar

The syntax of SLCO's DSL, called the SLCO 2.0 language, is defined through TEXTX, which is a meta-language tool for specifying DSLs in Python. The syntax definition is clear and unambiguous, and hence, it will be referred to extensively during the introduction of the SLCO 2.0 language to provide a concise overview of the supported structures. However, note that a full understanding of the TEXTX language will not be required–a natural language description will be given of each construct, independently from the syntax definition itself. Nevertheless, a basic understanding of the TEXTX grammar is considered helpful, since it allows for the concrete syntax and the textual description thereof to be cross-referenced if further clarification if required. As such, the remainder of this section will be dedicated to giving an introduction to the TEXTX grammar.

```
<name>: <body>;
```

Listing 12: The general structure of a TEXTX rule definition.

In TEXTX[2], a language is defined through rules. The general structure of a rule is provided in Listing 12. Each rule starts with a name, which is followed by a colon that leads into the rule's body, with the rule declaration itself being concluded with a semicolon. The rule's body is defined to be an expression that contains the desired parsing behavior of the rule. The following sections will discuss the majority of basic expressions that are provided by the TEXTX grammar.

A.1 Base Types

First, several base type rules are provided by the TEXTX language, including but not limited to the `ID`, `INT` and `BOOL` rules. An `ID` matches a common identifier that consists of letters, digits and underscores and is hence generally used as the type of a name or identity, an `INT` rule matches to integer values, and a `BOOL` rule matches to the Boolean values `true` and `false`.

A.2 Matching

```
StringMatch: 'A';
RegexMatch: \[a-z]+\;
```

Listing 13: A demonstration of matching expressions in the TEXTX grammar.

The base type rules are followed by expressions that match to a given string or regular expression. The matching expressions, demonstrated in Listing 13, are the basic building blocks of the language, and allow for more complex expressions to be created. String matches are defined through quoted strings and will match a literal string in the input, and are often used to match for keywords in the target language. Regular expressions, to be enclosed by backlashes, are defined using the syntax of regular expressions as used in Python.

A.3 Sequence

```
Sequence: E1 E2 E3;
```

Listing 14: A demonstration of a sequence in the TEXTX grammar.

[2]The official documentation of the grammar can be found at `https://textx.github.io/textX/3.0/grammar/`

The sequence operator, as demonstrated in Listing 14, is an n-ary operator that is used to chain sub-expressions together. Expressions contained within a sequence will be matched in the given order. By default, whitespace characters are skipped when parsing a sequence, and hence, sub-expressions may be separated by an arbitrary number of white spaces.

A.4 Ordered Choice

```
OrderedChoice: E1 | E2 | E3;
```

Listing 15: A demonstration of an ordered choice in the TEXTX grammar.

The n-ary vertical bar operator (|), as demonstrated in Listing 15, denotes an ordered choice between expressions. During parsing, the matching of the listed expressions will be performed from left to right, which guarantees that the first successful match will be chosen. The benefit of ordered choices is that the parsing always yields the same parse tree, and hence, the output will always be unambiguous. Furthermore, it allows for certain expressions to be prioritized.

A.5 Optional

```
Optional: E1 E2?;
```

Listing 16: A demonstration of optional expressions in the TEXTX grammar.

As demonstrated in Listing 16, an expression can be made optional during matching by appending it with the unary question mark operator (?). The rule `Optional` requires a match with `E1`, which is sequenced with an optional expression `E2`. In other words, the rule will match to both expression `E1` and the sequence of expressions `E1 E2`. The optional operator relatively has a high order of precedence–as shown through the example, the operator solely targets expression `E2`, and not the sequence `E1 E2` in its entirety. More complex optional behavior can be expressed by surrounding the target expression by a set of parentheses prior to adding the optional operator.

A.6 Repetitions

```
ZeroOrMore: E1*;
OneOrMore: E1+;
```

Listing 17: A demonstration of repetitions in the TEXTX grammar.

Repetition operators are provided that allow for an expression to be repeated an undefined number of times. Zero or more repetitions of an expression are specified by appending the expression with the asterisk (*) operator. Similarly, one or more repetitions are specified by appending the plus (+) operator. Both types of repetition operators are demonstrated in Listing 17. Note that, similarly to the optional operator, repetitions have a high order of precedence–more complex behavior can be expressed through the combination of parentheses and repetition operators.

```
UnorderedGroupS: ((E1 E2) E3)#;
UnorderedGroupC: (E1 E2 | E3)#;
```

Listing 18: A demonstration of unordered groupings in the TEXTX grammar.

An additional repetition type operator provided by TEXTX is the unordered grouping operator as presented in Listing 18. As shown in the listing, an unordered group is expressed by appending a sequence or ordered choice with the number sign (#). An unordered grouping operator matches any input that repeats a permutation of the sub-expressions contained within the given sequence or ordered choice. Note that nested sequence and ordered choice sub-expressions are not considered separate elements in the group–due to this, rules `UnorderedGroupS` and `UnorderedGroupC` are considered to be equivalent. Hence, the sequence `E1 E2 E3 E3 E1 E2` is a match to both rules in the given example, but `E3`, `E1 E3 E2` and `E3 E3 E1 E2` are not.

A.7 Assignments

```
PlainAssignment: a=E1;
```

Listing 19: A demonstration of a plain assignment in the TEXTX grammar.

All of the expressions and operators discussed up to this point have in common that they allow for a rule to be created that can match a certain input, but they do, thus far, not provide the means to extract information and data contained therein. The plain assignment operator (=), as demonstrated in Listing 19, allows for a target match or rule value to be saved as an attribute within the generated Python object graph. The left-hand side of the assignment contains the attribute name, with the right hand side holding a reference to another rule or match expression.

```
BooleanAssignment: a?=E1;
ZeroOrMoreAssignment: a*=E1;
OneOrMoreAssignment: a+=E1;
```

Listing 20: A demonstration of additional assignment types in the TEXTX grammar.

Note that the plain assignment is just one of four assignment types provided by the TEXTX grammar. The remaining types of assignments are presented in Listing 20. The first rule uses the Boolean assignment operator (`?=`), which sets the target attribute to `true` if the target match can be made, `false` otherwise. The second rule uses the zero or more assignment operator (`*=`), and assigns a list that contains all matches of the rule to the target attribute. If no matches can be made, the attribute will be assigned an empty list. Finally, the third rule uses the one or more assignment operator (`+=`), which behaves virtually the same as the zero or more assignment operator, but will fail if no matches can be made instead of returning an empty list.

A.8 Repetition Modifiers

```
RepetitionModifier: a*=E1[','];
```

Listing 21: A demonstration of a repetition modifier in the TEXTX grammar.

The repetition modifier expression demonstrated in Listing 21 allows for modifications to be made to expressions with repetition (`*`, `+`, `#`, `*=`, `+=`). The modifications are defined at the end of an expression between square brackets, with each entry separated by a comma. TEXTX defines two modifier types: separator modifiers and end-of-line termination modifiers (`eolterm`). The former is used to set a simple string or regular expression as the required separator between elements in the list, while the latter will terminate the repetition on an end-of-line character.

A.9 References

```
MatchRuleReference: a=E1;
LinkRuleReference: a=[E1];
```

Listing 22: A demonstration of references in the TEXTX grammar.

A reference to a rule can be included within another rule to create a nested structure of rules or refer by name to instances of rules that have been instantiated elsewhere. The two types of references are demonstrated in Listing 22, being the match rule and link rule references respectively, with the latter being defined by surrounding the rule name with square brackets. A match rule reference simply executes the target rule, while the link rule reference matches to an identifier for an existing instance of the given rule, with the reference itself being resolved automatically by TEXTX. By default, the identifier is equivalent to the `name` attribute of the target rule.

A.10 Syntactic Predicates

```
NegativeLookahead: !Keyword ID;
PositiveLookahead: Keyword &ID;
```

Listing 23: A demonstration of syntactic predicates in the TEXTX grammar.

The final basic expressions that remain to be discussed are syntactic predicates, which focus on implementing lookahead functionality for the parser. Two types of syntactic predicates are provided by the TEXTX grammar, namely the negative lookahead (`!`) and positive lookahead (`&`). Both types of lookaheads are demonstrated in Listing 23. Observe that the syntactic predicate operators are placed before their target expression. The rule `NegativeLookahead`, using a negative lookahead, matches to any `ID` that is not a match to `Keyword`, and can hence be used to ensure that the sets of available identities and keywords are disjoint. The rule `PositiveLookahead` uses a negative lookahead, and will only match an `ID` if it is preceded by a `Keyword`.