CARL
VON
OSSIETZKY
universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

MoDiShCa
Model Checking Discrete Shape Calculus

September 19, 2005

Jan-David Quesel

Carl von Ossietzky Universität Oldenburg
Fakultät II
Department für Informatik
Abteilung Entwicklung korrekter Systeme

Gutachter: Prof. Dr. Ernst-Rüdiger Olderog
Dipl. Inf. Andreas Schäfer

Abstract

This thesis describes how a decidable fragment of the Shape Calculus, an extended Duration Calculus, can be translated into WS1S. WS1S is the monadic second order logic with one successor. We generalise a translation for the two dimensional case to an arbitrary number of dimensions and beyond an implementation of it will be presented: MoDiShCa. Also we will illustrate how this tool can be used to verify complex systems by presenting a case study.

Contents

1 Introduction

This document describes how model checking of a decidable fragment of the spatio-temporal logic Shape Calculus (SC) [Sch05] is possible using the Model Checker [CGP99] MONA [EKM98]. This calculus is based on the Duration Calculus (DC) [ZHR91]. As in the Duration Calculus, it is not possible to specify spatial constraints, Shape Calculus adds such a possibility. DCVALID [Pan00] is a model checker for the DC using MONA as back-end. It is able to check specifications of discrete DC. It has some syntactical enhancements to make it easier to use.
The first chapter describes the SC and its features. To be able to model check the SC using MONA, it is necessary to translate SC formulas into weak S1S [Tho97] formulas (WS1S). The second chapter specifies an inductive translation of SC with an arbitrary number of finite spatial dimensions and one infinite temporal dimension into weak S1S. We use this fragment because it is decidable. The third chapter describes how the implementation of this translation schema is achieved and the input format of the tool. To illustrate how this translation is done a few examples will be given. A case study will illustrate how MoDiShCa can be used to verify greater systems.

2 Basics

2.1 Shape Calculus

The Shape Calculus is an interval logic able to describe system behaviors in time and space. As Shape Calculus itself is undecidable [Sch05], we only consider a decidable fragment in this thesis. The fragment makes the following restrictions: Instead of defining the interpretation function over $\mathbb{R}$, we use $\mathbb{N}$. Also we assume that the spatial dimensions are finite and that the cardinality of each dimension d is given by $card(d) \geq 1$. The cardinality of a dimension is the scale of that direction. As dimension names we use $d_0, \ldots, d_n$. We only use boolean observables. Integer observables can be expressed using the boolean observables and a binary encoding. The semantics of formulas is defined over intervals. As an abbreviation for $[x_1^0, x_2^0] \times \cdots \times [x_1^n, x_2^n]$ we will write $[\overrightarrow{x_1}, \overrightarrow{x_2}]$ and use the function $proj$ to reference the components of a vector:

Definition 1 $proj((x_1, \ldots, x_n), i) := x_i$

For manipulating a vector we define a substitution:

Definition 2 $(x_1, \cdots, x_{i-1}, x_i, x_{i+1}, \cdots, x_n)\{x_m, i\} := (x_1, \cdots, x_{i-1}, x_m, x_{i+1}, \cdots, x_n)$

We define an ordering on n-dimensional vectors by:

Definition 3 $\overrightarrow{x} \leq \overrightarrow{x}'$ *iff* $(x_1, \ldots, x_n) \leq (x'_1, \ldots, x'_n)$ *iff* $\forall i \in [1, n] : x_i \leq x'_i$ *and*
$\overrightarrow{x} < \overrightarrow{x}'$ *iff* $(x_1, \ldots, x_n) < (x'_1, \ldots, x'_n)$ *iff* $\forall i \in [1, n] : x_i < x'_i$

2.1.1 Syntax

There are three types of constructs in the SC. Observables are used to model system inputs and outputs.

State expressions are used to model system states and are generated by the abstract syntax:

$$\pi ::= X \mid \neg\pi_1 \mid \pi_1 \wedge \pi_2$$

where X is an observable.
Formulas are generated from state expressions using the following EBNF grammar:

$$F ::= \lceil\pi\rceil \mid \lceil\pi\rceil_{d_i} \mid \lceil\pi\rceil_t \mid F\,\langle d_i\rangle\,G \mid F\,\langle t\rangle\,G \mid F \wedge G \mid \neg F \mid l_t\ \sim\ n \mid l_{d_i}\ \sim\ n$$

where F and G are formulas, π is a state expression, n is an integer constant and $\sim\in\{\leq,\geq,<,>,=\}$.

2.1.2 Semantics

Observables System behavior is modeled by observables. Its semantics is given by: $\mathcal{I}[\![X]\!](t,\overrightarrow{x}) = 1$ iff the observable X is true at the spatial point $\overrightarrow{x}$ at time t.

State Expressions
State expressions are predicates over a discrete point in space and time. They are boolean combinations of observables. The semantics of the state expression is given by the following function:

$$\mathcal{I}[\![\pi]\!] : \mathbb{N}_0 \times \mathbb{N}_{\leq card(x_0)} \times \mathbb{N}_{\leq card(x_1)} \times \cdots \times \mathbb{N}_{\leq card(x_n)} \to \{0,1\}$$

where $\mathbb{N}_{\leq i} := \mathbb{N}_0 \backslash \{m | m \in \mathbb{N} \wedge m > i\}$ such that
$\mathcal{I}[\![\pi_1 \wedge \pi_2]\!](t,\overrightarrow{x}) = 1$ iff the state expression π_1 and the state expression π_2 are both evaluated to 1 at the spatial point $\overrightarrow{x}$ at the time t.
$\mathcal{I}[\![\neg\pi_1]\!](t,\overrightarrow{x}) = 1$ iff the state expression π is evaluated to 0 at the spatial point $\overrightarrow{x}$ at the time t.

Formulas
Formulas are interpreted over spatio-temporal intervals, their semantics is given by a function:

$$\mathcal{I}[\![F]\!] : [t_1,t_2] \times [x_1^0,x_2^0] \times \cdots \times [x_1^n,x_2^n] \to \mathbb{B}$$

where $t_1,t_2 \in \mathbb{N}_0$ and $x_1^i,x_2^i \in \mathbb{N}_{\leq card(d_i)}$.
We now inductively define this function using meta-quantifiers that should be interpreted as "for all" ($\forall$) and "there is a" ($\exists$):

- The first operator is everywhere X ($\lceil X\rceil$). It expresses that at every spatial point, at every point in time X is true. So it is defined by

$$\mathcal{I}[\![\lceil\pi\rceil]\!]([t_1,t_2],[\overrightarrow{x_1},\overrightarrow{x_2}]) := \mathit{true} \text{ iff } \forall t \in [t_1,t_2]$$
$$\forall\overrightarrow{x} \in [\overrightarrow{x_1},\overrightarrow{x_2}] : \mathcal{I}[\![\pi]\!](t,\overrightarrow{x}) = 1$$

- Everywhere X in direction d_i ($\lceil X \rceil_{d_i}$) means that for every possible value of x_i, the i-th component of the vecors, there is a point in time and there are values for the other spatial dimensions such that X is true. This is a projection on the d_i axis. Formally,

$$\mathcal{I}[\![\lceil \pi \rceil_{d_i}]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := true \text{ iff}$$
$$\forall x_i \in [proj(\overrightarrow{x_1}, i), \overrightarrow{x_2}, i)]\ \exists t \in [t_1, t_2]\ \exists \overrightarrow{x}' \in [\overrightarrow{x_1}, \overrightarrow{x_2}] : \mathcal{I}[\![\pi]\!](t, \overrightarrow{x}'\{x_i, i\}) = 1$$

- All-the-time X means that for every point in time there is a point in the spatial interval such that X is true. This is a projection on the time line.

$$\mathcal{I}[\![[\pi]_t]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := true \text{ iff}$$
$$\forall t \in [t_1, t_2] \exists \overrightarrow{x} \in [\overrightarrow{x_1}, \overrightarrow{x_2}] : \mathcal{I}[\![\pi]\!](t, \overrightarrow{x}) = 1$$

- The chop in direction d_i means that it is possible to split the spatial interval in the direction d_i such that on the first part F is true and on the second part G is true.

$$\mathcal{I}[\![F \langle d_i \rangle G]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := true \text{ iff}$$
$$\exists x_m \in [proj(\overrightarrow{x_1}, i), proj(\overrightarrow{x_2}, i)] : \mathcal{I}[\![F]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}\{x_m, i\}]) = true \text{ and}$$
$$\mathcal{I}[\![G]\!]([t_1, t_2], [\overrightarrow{x_1}\{x_m, i\}, \overrightarrow{x_2}]) = true$$

- The time-chop means that it is possible to split the temporal interval such that on the first part F is true and on the second part G is true.

$$\mathcal{I}[\![F \langle t \rangle G]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := true \text{ iff}$$
$$\exists t \in [t_1, t_2] : \mathcal{I}[\![F]\!]([t_1, t], [\overrightarrow{x_1}, \overrightarrow{x_2}]) = true \text{ and}$$
$$\mathcal{I}[\![G]\!]([t, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) = true$$

- Conjunction and negation are handled as usual.

$$\mathcal{I}[\![F \wedge G]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := true \text{ iff}$$
$$\mathcal{I}[\![F]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) = true \text{ and}$$
$$\mathcal{I}[\![G]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) = true$$

  and

$$\mathcal{I}[\![\neg F]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := true \text{ iff } \mathcal{I}[\![F]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) = false$$

- The operator l_t is introduced for statements about the length of the *temporal* interval. It can be compared to an integer constant using $\sim$ which is an binary relation from $\{\leq, \geq, <, >, =\}$.

$$\mathcal{I}[\![l_t \sim n]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := true \text{ iff } t_2 - t_1 \sim n$$

- Similarly, the operator l_{d_i} can be used for statements about the length of a *spatial* interval.

$$\mathcal{I}[\![l_{d_i} \sim n]\!]([t_1, t_2], [\overrightarrow{x_1}, \overrightarrow{x_2}]) := \mathit{true} \text{ iff } \mathit{proj}(x_2, i) - \mathit{proj}(x_1, i) \sim n$$

2.1.3 Abbreviations

To make formulas easier to read and modeling more intuitive, we introduce the following abbreviations:

- $\Diamond_{dim}(F)$ abbreviates $\mathit{true}\ \langle dim \rangle\ F\ \langle dim \rangle\ \mathit{true}$. The operator is called "eventually".
- The dual operator $\Box_{dim}(F)$ is defined as $\neg\Diamond_{dim}(\neg F)$. The operator is called "always".

The dimension *dim* could either be some spatial or the temporal dimension.

2.2 Weak S1S

Weak second order logic with one successor (WS1S) is a subset of monadic second order logic. Second order logic introduces in addition to first order logic the possibility to use quantifiers over relations. The keyword *monadic* means that these relations can only have the arity one. That means they are sets. In WS1S it is possible to express some properties of arithmetic on natural numbers. The keyword weak in weak S1S means that the interpretation of the second order quantifiers $\exists X$ and $\forall X$ are changed into "there is a *finite* subset X of $\mathbb{N}$..." and "for all *finite* subsets X of $\mathbb{N}$...". But this restriction does not change the expressiveness of the logic as it can be proven that WS1S and S1S have the same expressive power [Tho97]. In second order logic, other than in first order logic, it is possible to derive $\leq$ and $<$ from this successor predicate. Weak S1S formulas can be constructed using the following EBNF grammar:

$$F := x \in X \mid \exists x : F \mid \exists X : F \mid F \wedge G \mid \neg F \mid S(x, y)$$

where X is a second order variable, x and y are first order variables, F and G are formulas and $S(x, y)$ is the mentioned successor predicate. The expression $x \in X$ means that the first order variable x is in the set X. $S(x, y)$ is true iff x is the successor of y and $S(x, y) \wedge S(x, z) \Rightarrow y = z$.
In WS1S we can define every ω-regular language [Tho97] and for every WS1S formula there is an ω-regular language that characterizes it. As WS1S is a decidable logic [Tho97], we use a direct translation from SC into WS1S to do the model checking.

2.3 MONA

MONA [HJJ+95] is a model checker for WS1S. The development of MONA started 1994 at BRICS [HJJ+95]. MONA translates the formulas into finite automata and is able to decide if the language of these automata is empty or if a state is reachable. MONA represents the automata as BDDs. This representation leads to a very efficient implementation of the transformation from WS1S to finite automata and the decision procedures.

3 Translation schema

To be able to use the model checker MONA, we need to translate the SC formulas into WS1S. We assume that we have $n+1$ spatial dimensions. Each point in space is represented by a vector. To do the translation, we introduce for each observable X $\prod_{i=0}^{n} card(d_i)$ second order variables, such that $X_{\overrightarrow{a}}$ models the truth value of X at spatial position $\overrightarrow{a}$ on the time line. We define inductively the function SO to translate a SC formula into a WS1S formula. To do the translation we need a temporal interval $[t_1, t_2]$ and a spatial interval $[\overrightarrow{x_1}, \overrightarrow{x_2}]$. The WS1S formula generated by SO is satisfiable iff the SC formula is satisfiable on these intervals, the formula is also valid iff the SC formula is valid on these intervals.
We use the same names for the axes as in the previous section.
State expressions are handled by the translation schema on the lowest level into set operations and boolean connections in second order logic. The state expression $\neg X$ is translated into $t_m \notin X_{\overrightarrow{x}}$ and $X \wedge Y$ is translated into $t_m \in X_{\overrightarrow{x}} \wedge t_m \in Y_{\overrightarrow{x}}$. In the following we only mention the simple state expression X, which is translated into $t_m \in X_{\overrightarrow{x}}$, instead of duplicating the formulas three times.
First we translate everywhere X ($\lceil X \rceil$). The universal quantification over spatial points is expressed as conjunction over all possible points (vectors). This conjunction is finite because we only consider finite and discrete space.

$$\begin{aligned} SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(\lceil X \rceil) = {} & t_1 < t_2 \wedge \overrightarrow{x_1} < \overrightarrow{x_2} \wedge \\ & \forall t_m : t_1 \le t_m < t_2 \Rightarrow \bigwedge_{\overrightarrow{x}=\overrightarrow{x_1}}^{\overrightarrow{x_2}-1} t_m \in X_{\overrightarrow{x}} \end{aligned}$$

The operator everywhere X in direction d_i ($\lceil X \rceil_{d_i}$) means that for all possible values of x_i there is a point in time and there are values for the other directions such that X is true. This is a projection to the d_i axis. Existential quantification over spatial points is expressed as disjunction over all possible points (vectors). Its also finite because we only consider finite and discrete space.

$$\begin{aligned} SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(\lceil X \rceil_{d_i}) = {} & t_1 < t_2 \wedge \overrightarrow{x_1} < \overrightarrow{x_2} \wedge \\ & \bigwedge_{x_i=proj(\overrightarrow{x_1},i)}^{proj(\overrightarrow{x_2},i)-1} \bigvee_{x_0=proj(\overrightarrow{x_1},0)}^{proj(\overrightarrow{x_2},0)-1} \bigvee_{x_1=proj(\overrightarrow{x_1},1)}^{proj(\overrightarrow{x_2},1)-1} \cdots \\ & \bigvee_{x_{i-1}=proj(\overrightarrow{x_1},i-1)}^{proj(\overrightarrow{x_2},i-1)-1} \bigvee_{x_{i+1}=proj(\overrightarrow{x_1},i+1)}^{proj(\overrightarrow{x_2},i-1)-1} \cdots \bigvee_{x_n=proj(\overrightarrow{x_1},n)}^{proj(\overrightarrow{x_2},n)-1} \\ & \exists t_m : t_1 \le t_m < t_2 \wedge t_m \in X_{(x_0,x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_n)} \end{aligned}$$

The operator all-the-time X ($\lceil X \rceil_t$) means that for every point in time there is a vector

$\overrightarrow{x}$ such that X is true at the point $\overrightarrow{x}$. This is a projection to the time line.

$$\begin{aligned} SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(\lceil X \rceil_t) \ &= t_1 < t_2 \wedge \overrightarrow{x_1} < \overrightarrow{x_2} \wedge \\ &\forall t_m : t_1 \leq t_m < t_2 \Rightarrow \bigvee_{\overrightarrow{x} : \overrightarrow{x_1} \leq \overrightarrow{x} < \overrightarrow{x_2}} t_m \in X_{\overrightarrow{x}} \end{aligned}$$

The temporal chop is defined as: there is a point t in the current interval such that F is true in the interval $[t_1, t]$ and G is true in the interval $[t, t_2]$. We translate it by simply using $\wedge$ and changing the time stamp parameters.

$$\begin{aligned} SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(F \langle t \rangle G) = \exists t_m : t_1 \leq t_m \leq t_2 \wedge SO(t_1, \overrightarrow{x_1}, t_m, \overrightarrow{x_2})(F) \wedge \\ SO(t_m, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(G) \end{aligned}$$

The chop operator in d_i direction is defined as: there is a point x_m on the d_i-axis such that in the interval $[\overrightarrow{x_1}, \overrightarrow{x_2}\{x_m, i\}]$ F is true and on the interval $[\overrightarrow{x_1}\{x_m, i\}], \overrightarrow{x_2}]$ G is true. To translate it, we use the same basic idea as for the translation of the temporal chop, but we alter the vectors x_1 and x_2 instead of the timestamps.

$$\begin{aligned} SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(F \langle d_i \rangle G) = &\bigvee_{x_m = proj(\overrightarrow{x_1}, i)}^{proj(\overrightarrow{x_2}, i)} (SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2}\{x_m, i\})(F) \\ &\wedge SO(t_1, \overrightarrow{x_1}\{x_m, i\}, t_2, \overrightarrow{x_2})(G)) \end{aligned}$$

The translation of conjunction and negation are straight forward. Conjunction and negation are handled the same in SC and WS1S so we just keep them.
Conjunction:

$$SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(F \wedge G) = SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(F) \wedge SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(G)$$

Negation:

$$SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(\neg F) = \neg SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(F)$$

The operator l is able to determine the length of observation intervals. With $l_t \sim n$ with $\sim \in \{\leq, \geq, <, >, =\}$ we can determine the length of the temporal interval.

$$SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(l_t \sim n) = t_2 - t_1 \sim n$$

We translate the same operator for the spatial dimensions l_{d_i}.

$$SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(l_{d_i} \sim n) = proj(x_2, i) - proj(x_1, i) \sim n$$

Finally we define the constants $true$ and $false$ with the same meaning as in WS1S.

$$SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(true) = true$$
$$SO(t_1, \overrightarrow{x_1}, t_2, \overrightarrow{x_2})(false) = false$$

4 Program

4.1 Features

MoDiShCa is able to translate a textual representation of the SC into MONA syntax, such that it can be model checked. It supports the discrete SC as described in section 2.1. Beyond the operators defined in section 2.1 some more operators are implemented to enhance the usability of the program. Operators added are implication, biimplication, disjunction and the operators always and eventually as defined in 2.1.3.
Additionally MoDiShCa supports formula variables, constants and integer variables for easier modeling. Formula variables are used to make the specification easier to read. The value of a formula variable is a formula. See section 6 for an example how to use these formula variables.

4.2 Usage

MoDiShCa reads its input from the standard input and places its output on the standard output. This makes it possible to use pipes to preprocess the input or post process the output. The shell script *modishca.sh* needs two parameters: `inputfile` and `outputfile`. It will call MoDiShCa using the specified input file as input and it will redirect the output to the specified output file.
The input format of MoDiShCa is simple. There are a few declarations: observables, dimensions with its cardinalities and constants. The observables can either be boolean or restricted integers. Restricted means that you have to specify the maximum value. The minimum value is always 0. In the declaration part you can also define formula variables. The formula to check is introduced by the keyword `verify:` as the last line in the file. The following tabular shows how the declaration part should be used.

Declaration	Keyword	Example
Boolean observable	`bool`	`bool X`
Integer observable	`int`	`int a[5]`
Constant	`const`	`const ten = 10`
Spatial dimension	`dim`	`dim x = 3`
Formula variable		`$assumption = [X]<t>[a = 3]`

The declarations are separated by a `;`. Formula variables have to be defined after the other declarations. The name of every formula variable starts with a $. They can be used from the moment they were defined such that you can use one formula variable to define another one. They are referenced with there full name including the leading $.
The translation can be done in two modes. We can either check satisfiability or

validity. By default, the translation is done in the satisfiability mode. We can enable the other mode by adding the keyword `validity` into the declaration part of the input file.
The operators that can be used are the following:

Name	SC syntax	MoDiShCa syntax
Everywhere	$\lceil\pi\rceil$	`[pi]`
Everywhere in direction x	$\lceil\pi\rceil_x$	`[pi]_x`
All-the-time	$\lceil\pi\rceil_t$	`[pi]_t`
Temporal-chop	$\langle t\rangle$	`<t>`
Spatial-chop	$\langle x\rangle$	`<x>`
Length of observation intervals	l_t or l_x	`l_t` or `l_x`
Always	$\Box_t$ or $\Box_x$	`[]_t` or `[]_x`
Eventually	$\Diamond_t$ or $\Diamond_x$	`<>_t` or `<>_x`
Conjunction of state expression or formulas	$\wedge$	`&`
Disjunction of state expression or formulas	$\vee$	`|`
Implication of state expression or formulas	$\Rightarrow$	`->`
Biimplication of state expression or formulas	$\Leftrightarrow$	`<->`
Negation of state expression or formulas	$\neg$	`not`

where `pi` is a state expression, `x` is a spatial dimension. The associativity is left for the binary operators and right for the unary ones. For the state expressions we have the following binding preferences for the operators beginning with the highest priority: *negation*, *conjunction*, *disjunction*, *implication*, *biimplication*. For formulas they are: *always*, *eventually* and *negation* with same priority followed by: *chop*, *conjunction*, *disjunction*, *implication* and *biimplication* with decreasing priorities.

4.3 Implementation

The implementation is done using flex and bison as generators for scanner and parser. The parser is programmed in C++ while the scanner is in simple C.
To represent the syntax tree, there is a data type called NODE and one called SYMBOL. NODE contains the elements of the formula (as tree structure), while SYMBOL is a connected list containing the declared observables, dimensions and sub formulas. At parsing time we use the linked list FORMLIST to store the subtrees that are referenced by the formula variables. After parsing is done, a class named SCToSOTranslator is instantiated. This class handles the translation by traversing the syntax tree and using the translation schema of section 3 to translate the formula into WS1S. It uses two std::streams, one for the declarations and one for the formula.
Figure 1 illustrates the data flow. The input data is translated into symbols by the scanner and the parser generates the syntax tree and a symbol table. Then the translator generates output data, in our case WS1S formulas which are passed to MONA for model checking. MONA generates an output, which is either that the formula is satisfiable, then MONA will produce a satisfying example and a counter example. Or the formula is unsatisfiable, then MONA will produce a counter example, or the formula is valid, then MONA will produce a satisfying example.

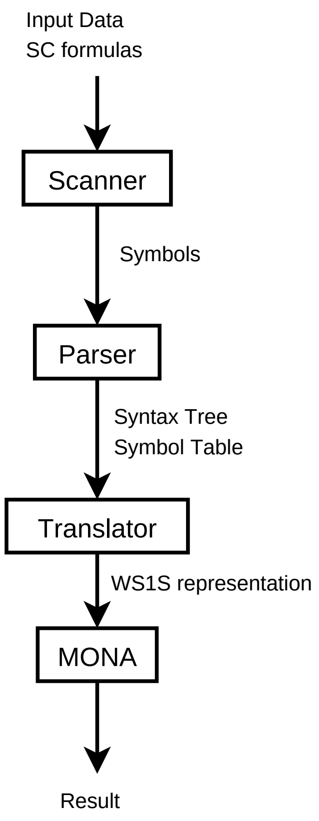

Figure 1: Data flow

The class structure of the program is illustrated in figure 2. The main class is the SCToSOTranslator. It implements the translation schema described in section 3.
The implementation provides some operators not mentioned in the translation schema of section 3 such as *implication* (`->`), *biimplication* (`<->`), *disjunction* (|), *always* (`[]_dim`) and *eventually* (`<>_dim`). *Implication*, *biimplication* and *disjunction* are handled the same in SC as in WS1S so they were promoted like *conjunction* and *negation*. *Eventually* ($\Diamond_{dim} F$) is syntactically replaced by its definition in 2.1.3 while parsing the formula. The dual operator *always* ($\Box_{dim} F$) is also syntactically replaced at parsing time.
The inductive definition of the translation schema is implemented as a recursive function, named *recTranslate* which looks at the root of a formula (sub-)tree and decides how to handle that type of node.
As MONA cannot handle an equation like $t_2 - t_1 \sim n$ we need to translate this into a form MONA understands. MONA can solve $t_2 \sim t_1 + n$ so we just change the formula to this format.
It is easier to implement $l_{d_i} \sim n$ because we can determine if it is *true* or *false* at compile time because the spatial dimensions are finite. The possibility to have restricted integer variables is implemented by translating the integer into its binary representation and use a boolean observable for every bit. The integer variables can be compared to integer constants with the operations $\{\leq, \geq, <, >, =\}$.

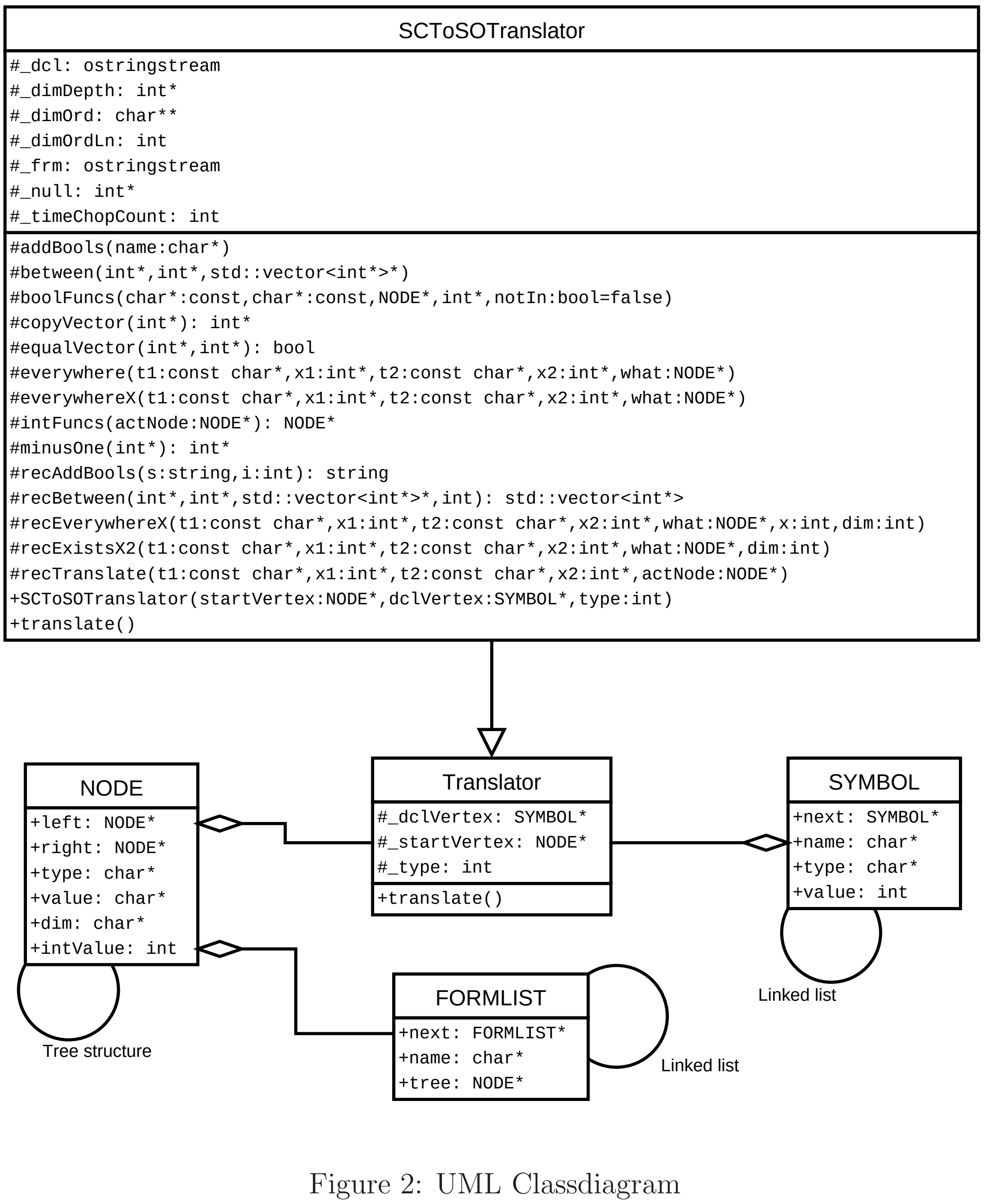
SCToSOTranslator
#_dcl: ostringstream
#_dimDepth: int*
#_dimOrd: char**
#_dimOrdLn: int
#_frm: ostringstream
#_null: int*
#_timeChopCount: int
#addBools(name:char*)
#between(int*,int*,std::vector<int*>*)
#boolFuncs(char*:const,char*:const,NODE*,int*,notIn:bool=false)
#copyVector(int*): int*
#equalVector(int*,int*): bool
#everywhere(t1:const char*,x1:int*,t2:const char*,x2:int*,what:NODE*)
#everywhereX(t1:const char*,x1:int*,t2:const char*,x2:int*,what:NODE*)
#intFuncs(actNode:NODE*): NODE*
#minusOne(int*): int*
#recAddBools(s:string,i:int): string
#recBetween(int*,int*,std::vector<int*>*,int): std::vector<int*>
#recEverywhereX(t1:const char*,x1:int*,t2:const char*,x2:int*,what:NODE*,x:int,dim:int)
#recExistsX2(t1:const char*,x1:int*,t2:const char*,x2:int*,what:NODE*,dim:int)
#recTranslate(t1:const char*,x1:int*,t2:const char*,x2:int*,actNode:NODE*)
+SCToSOTranslator(startVertex:NODE*,dclVertex:SYMBOL*,type:int)
+translate()
NODE
+left: NODE*
+right: NODE*
+type: char*
+value: char*
+dim: char*
+intValue: int
Tree structure
Translator
#_dclVertex: SYMBOL*
#_startVertex: NODE*
#_type: int
+translate()
SYMBOL
+next: SYMBOL*
+name: char*
+type: char*
+value: int
Linked list
FORMLIST
+next: FORMLIST*
+name: char*
+tree: NODE*
Linked list

Figure 2: UML Classdiagram

The satisfiability and validity check is implemented by adding $t_1 \leq t_2 \wedge$ for satisfiability and $t_1 \leq t_2 \Rightarrow$ for the validity check and genarating the formulas for different lengths of the spartial interval. These formulas are connected by a disjunction for the satisfiability check, because we search one spatial interval to satisfy the formula. A conjunction of these formulas is generated for the validity check as every possible interval has to satisfy the formula if it is valid.

5 Examples

In this section we illustrate how to do specifications in SC, how the syntax of the input files for MoDiShCa looks like and what kind of output is produced.

5.1 Pure DC formulas

As SC is an extention of DC, MoDiShCa is able to handle pure DC formulas like DCVALID.

a) $\lceil X \rceil \Rightarrow (\lceil X \rceil \langle t \rangle \lceil X \rceil)$
In MoDiShCa syntax this formula would look like this: `[X]->([X]<t>[X])`. To do the translation we also need an declaration part. This formula is an DC formula so we do not need to define a spatial dimension. A complete input file for MoDiShCa would look like this:

```
bool X;
verify: [X]->([X]<t>[X])
```

The formula expresses that if X is globally true, then it is possible to split the temporal interval at a discrete point into two parts such that at one side, X is globally true and on the other side the same holds. This formula is satisfiable but not valid. So if we run MONA on the output, we get a satisfying example and a counter example. If we run the validity check by adding the keyword `validity` in the declaration block, the output of MONA still contains a counter example.
The counter example looks like this:

```
A counter-example of least length (2) is:
t1              X 1X
t2              X 01
X               X 1X

t1 = 0
t2 = 1
X = {0}
```

This means that for a temporal interval of length 1 (because `t2` is 1 and `t1` is 0, these are the borders of the temporal interval), the formula is not satisfiable, because if X is true at the point 0 the first part of the formula is true, but there is no discrete point

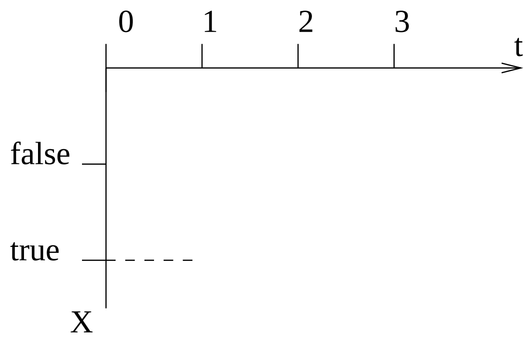

Figure 3: Interpretation function to the counter example

to chop such that we get $l \geq 1 \langle t \rangle l \geq 1$. This form is needed because $\lceil X \rceil$ requires a temporal interval greater than zero. As we need to satisfy it two times we need a temporal interval greater than 1. Figure 3 illustrates how the interpretation function looks like.

b) $(\lceil X \rceil \langle t \rangle \lceil X \rceil) \Rightarrow \lceil X \rceil$
The formula expresses that if it is possible to split the temporal interval into two parts such that on one side X is globally true and on the other side the same holds, X is globally true on the complete temporal interval. If we run the validity check on this formula, we get a different from case a) result.

```
bool X;
validity;
verify:
([X]<t>[X])->[X]
```

MONA correctly reports that the formula is valid.

5.2 SC formulas

a) $\lceil X \rceil \Rightarrow (\lceil X \rceil \langle x \rangle \lceil X \rceil)$
This formula is satisfiable but not valid as the formula in section 5.1 a) is.
The input file would look like this:

```
dim x = 3;
bool X;
validity;
verify: [X] -> ([X]<x>[X])
```

MONA reports the following counter-example:

```
A counter-example of least length (2) is:
t1              X 1X
t2              X 01
```

```
X_0             X 1X
X_1             X XX
X_2             X XX

t1 = 0
t2 = 1
X_0 = {0}
X_1 = {}
X_2 = {}
```

This means that the length of the observation interval in direction x is 1 ($l_x = 1$), such that we cannot find a position for the spatial chop.

b) $(\lceil X \rceil \langle x \rangle \lceil X \rceil) \Rightarrow \lceil X \rceil$
This is not a DC formula because it uses a chop in a spatial direction x. The formula says that if it is possible to split the spatial interval into two parts such that on the first part X is globally true and on the other part X is also globally true, then X is globally true on the complete interval.

```
dim x = 3;
bool X;
validity;
verify:
([X]<x>[X])->[X]
```

With this input file MONA reports that the formula is valid. In this example we consider the cardinality of the spatial dimension x to be 3.

5.3 Derived operators

The examples shown in this section are all valid and MONA reports this as aspected.

a) $\lceil X \rceil \Rightarrow (\Diamond_t \lceil X \rceil)$
The next two examples illustrate how the diamond operator works. If always X is true, there is an interval where X is true.

```
bool X;
validity;
verify:
([ X ]) -> (<>_t [ X ])
```

b) $(\lceil X \rceil \Rightarrow (\Diamond_x \lceil X \rceil))$
We can prove the same in a spatial dimension:

```
dim x = 4;
bool X;
```

```
validity;
verify:
(([ X ]) -> (<>_x [ X ]))
```

c) $(\Diamond_t \lceil X \wedge Y \rceil \Rightarrow (\Diamond_t \lceil X \rceil) \wedge (\Diamond_t \lceil Y \rceil))$
The next example illustrates that if there is an interval where X and Y are true, there are intervals where X is true and where Y is true, is valid.

```
bool X,Y;
validity;
verify:
((<>_t [ X & Y]) -> ((<>_t [ X ]) & (<>_t [ Y ])))
```

d) $\lceil X \rceil \Rightarrow (l_t = 0 \, \langle t \rangle \, \lceil X \rceil \, \langle t \rangle \, l_t = 0)$
We can chop $l_t = 0$ as often as we like. It does not change the satisfiability.

```
bool X;
$isnull = (l_t = 0);
verify:
[X]->($isnull<t>[X]<t>$isnull)
```

This example also demonstrates the use of formula variables to shorten the formula to check and make it easier to read.

6 Case Study

6.1 Description

The generalized railroad crossing [HL94] is used to verify an implementation of a train gate controller. We model a railroad crossing and the controller should close the gates when a train is approaching early enough to ensure that the train cannot reach the gate while it is open. The controller has a sensor that shows him if the rails are empty or if there is a train approaching to the gate or crossing the gate. Figure 4 illustrates the different zones. We use the SC to do this specification because we can model the railroad behavior very naturally by using a spatial dimension for the rails.

6.2 Modeling

We model the rails as one spatial dimension and employ 2 observables: `train` and `open`. The observable `train` is true at a spatial point at a given time iff the train is there at that time. The other observables models the gate status, it is true iff the gate is open. The sensor is modelled by one formula for each state

$$\begin{aligned} \mathit{empty} &:= (\lceil \neg \mathit{train} \rceil \wedge l_x \geq 10) \, \langle x \rangle \, \mathit{true} \\ \mathit{appr} &:= (\lceil \neg \mathit{train} \rceil_x \wedge l_x < 10 \wedge l_x \geq 2) \, \langle x \rangle \, \lceil \mathit{train} \rceil_x \, \langle x \rangle \, \mathit{true} \\ \mathit{cross} &:= ((\lceil \mathit{train} \rceil_x \vee ((\lceil \neg \mathit{train} \rceil_x \wedge l_x < 2) \, \langle x \rangle \, \lceil \mathit{train} \rceil_x))) \, \langle x \rangle \, \mathit{true} \end{aligned}$$

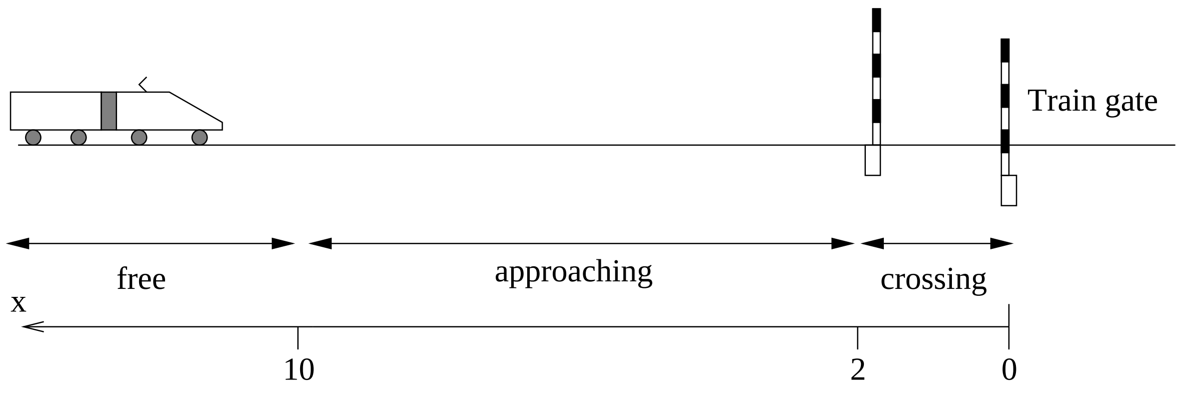

Figure 4: The train gate zones

The state *empty* is defined as the train is 10 or more spatial units away. The train is considered to be approaching (*appr*) if it is in the spatial interval $[2, 10[$ and it is crossing if it is in the interval $[0, 2[$.
We model the train with one state: *run*. The formula *run* is true iff the train is moving with maximum speed ($maxspeed - 1$). We model the speed to be $maxspeed - 1$ because we need an interval where there is no train.

$$\begin{aligned} run \ := \ & (\Box_t(\Box_x((((((\lceil\neg train\rceil \wedge l_x = maxspeed) \langle x\rangle \lceil train\rceil) \wedge l_t = 1) \langle t\rangle (l_t = 1)) \\ & \Rightarrow (l_t = 1 \langle t\rangle (\lceil\neg train\rceil \wedge l_x = 1) \langle x\rangle \lceil train\rceil \langle x\rangle true)))) \end{aligned}$$

The formula *run* describes that, if there is an interval in time and space where there is no train in space for *maxspeed* spatial units and behind that point there is the train, if this interval has the temporal length one and there is an interval of the length one afterwards, we could also split the interval such that after one time unit the spatial interval could be split such that the train is only one spatial unit away.
We need the following assumptions about the environment: At the initial state, the gate should be open and there is a train in the interval $[10, max]$, so that the rails are considered empty by the controller initially. Also there should only be one train. Furthermore, we assume that the gates need $closeTime$ time units to close.

$$\begin{aligned} initOpen \ &:= \ ((\lceil open\rceil_t \langle t\rangle true) \vee l_t = 0) \\ existsTrain \ &:= \ (l_t > 0) \Rightarrow ((l_t = 1 \wedge ((\lceil\neg train\rceil \wedge l_x = 9) \langle x\rangle \lceil train\rceil)) \langle t\rangle true) \\ onlyOnetrain \ &:= \ \Box_t(\Box_x(\neg(\lceil train\rceil \langle x\rangle \lceil\neg train\rceil \langle x\rangle \lceil train\rceil))) \\ closeTime \ &:= \ (\neg\Diamond_t(\lceil\neg open\rceil \langle t\rangle (\lceil open\rceil \wedge l_t < closeTime) \langle t\rangle \lceil t\rceil)) \\ & \quad \wedge (\neg((\lceil open\rceil \wedge l < closeTime) \langle t\rangle true)) \end{aligned}$$

After we have modeled the environment needed, we can now specify the controller. The controller needs to close the gates if a train is approaching or crossing within $reactTime$ time units. Another requirement is that the gates should be open if not safety critical such that the gates can be opened when *empty* is true. In this example we abstract

```
bool train;
bool open;
dim x = 10;
const closeTime = 1;
const maxSpeed = 2;
const reactTime = 2;
validity;
$empty = ([not train] & l_x >= 10)<x>true;
$appr = ([not train]_x & l_x < 10 & l_x >= 2)<x>[train]_x<x>true;
$cross = (([train]_x | (([not train]_x & l_x < 2)<x>[train]_x)))<x>true;
$run = ([]_t ([]_x (((((([not train] & l_x = maxSpeed)
    <x>[train])& l_t = 1)<t>(l_t=1)
    -> (l_t = 1 <t> ([not train] & l_x = 1)<x>[train]_x<x>true))));
$safety = not (<>_t(<>_x ($cross & [open])));
$reactAppr = ([]_t ((($appr | $cross) & l_t > reactTime)
    -> ((l_t = reactTime)<t>[not open])));
$openIfPossible = ([]_t ($empty -> [open]_t));
$closeTime = (not (<>_t (([not open]<t>([open]
    & l_t < closeTime)<t>[not open]))))
    & (not (([open] & l_t<closeTime)<t>true));
$existsTrain = (l_t > 0)
    -> (( l_t=1 & (([not train] & l_x = 9) <x> [train]))<t>true);
$onlyOnetrain = []_t( []_x (
    not ([train] <x> [not train] <x> [train])));
$initOpen = (([open]_t<t>true) | l_t=0);
$assumptions = $closeTime & $initOpen & $run & $onlyOnetrain
    & $existsTrain & $reactAppr & $openIfPossible;
$safetyQuest = $assumptions -> $safety;
verify: $safetyQuest
```

Figure 5: Railroad crossing in MoDiShCa syntax

from the time the gate would need to open because this is not safty critical.

$$\begin{aligned} reactAppr &:= (\Box_t(((appr \vee cross) \wedge l_t > reactTime) \\ &\qquad \Rightarrow ((l_t = reactTime)\,\langle t\rangle\,\lceil \neg open\rceil))) \\ openIfPossible &:= (\Box_t(empty \Rightarrow \lceil open\rceil_t)) \end{aligned}$$

The safety requirement to verify is:

$$safety := (\neg\Diamond_t(cross \wedge \lceil open\rceil_t))$$

The full specification in MoDiShCa syntax is shown in figure 5.

6.3 Result

Using MoDiShCa and MONA to check the specification in figure 5 we can prove that the system is safe meaning the gates are always closed while the train is crossing. The satisfying example given by MONA is the trivial case where $l_t = 0 = l_x$. Checking this specification with MoDiShCa and MONA takes about 5.708 seconds on a Athlon XP 2200+ with 512 MB RAM. MoDiShCa needs about 1.891 seconds to generate the WS1S formulas and MONA needs about 3.817 seconds to check this WS1S formula. If we change the model by using the projection to the x axis in the formulas for *empty*, *run* and *existsTrain* the model checking needs more than 1.5 GB RAM and aborts with an out of memory error. This problem occurs due to the fact that if we use the projections there are more possible solutions to the formula.

7 Conclusion

In this project we have shown how a decidable fragment of the SC looks like and how the translation into WS1S for the two dimensional case can be generalized to an arbitrary number of dimensions. We have implemented a tool that is able to do the translation and pass it to the model checker MONA. To demonstrate that the tool can be used to verify complex systems we presented an application to a case study. MoDiShCa is an efficient tool also for the discrete DC because of the optimizations implemented in MONA. Using MONA as back-end was a good decision because MONA has an efficient implementation of the automaton in BDD representation and the tool is already approved.
For future work, the output format could be enhanced. The output generated for MONA is just an interchange format at the moment. There are only a few comments in the generated MONA input file and the line breaks are not optimized yet. For debug purposes this interchange format could be made more human readable. Also some parts of the formula that are generated could be simplified to true and false at compile time but this is not done yet. Another enhancement could be to introduce quantified integer variables that can be used to store the length of an interval. This would enhance the expressiveness of the fragment that we were able to check. The case study could be extended by – for example – adding a spatial dimension for the gate such that the opening and closing process can be easily modeled.

8 Appendix

8.1 Symbols

Symbol name	Regular Expression
INT	int
CONST	const
LENGTH	l
VERIFY	verify:
REL	<= | >= | < | >
EQUAL	=
DECLARATION	BEGIN
END	END
SEMICOLON	;
COMMA	,
BIIMPL	<->
IMPL	->
NOT	not
BOOL	bool
DCL	dim
AND	&
OR	|
USCORE	_
CHOP	<[a-zA-Z][a-zA-Z0-9]*>
BOX	<>
DIAMOND	[]
GLBRACE	[
GRBRACE	]
LBRACE	(
RBRACE	)
VALIDITY	validity
TRUE	true
FALSE	false
FORMVAR	$[a-zA-Z][a-zA-Z0-9]*
WORD	[a-zA-Z][a-zA-Z0-9]*
NUMBER	[0-9]+

8.2 Yacc grammar

S is the start symbol.

S	::=	declaration VERIFY expr
S	::=	DECLARATION declaration END expr
S	::=	VERIFY expr
S	::=	forms VERIFY expr
declaration	::=	dclrtn
declaration	::=	dclrtn forms

dclrtn	::=	dclrtn SEMICOLON dclrtn
dclrtn	::=	dclrtn SEMICOLON
dclrtn	::=	DCL WORD EQUAL NUMBER
dclrtn	::=	BOOL WORD
dclrtn	::=	BOOL WORD booldec
dclrtn	::=	VALIDITY
dclrtn	::=	CONST WORD EQUAL NUMBER
dclrtn	::=	INT WORD GLBRACE NUMBER GRBRACE
booldec	::=	COMMA WORD booldec
booldec	::=	COMMA WORD
forms	::=	forms SEMICOLON forms
forms	::=	FORMVAR EQUAL expr
forms	::=	forms SEMICOLON
expr	::=	expr IMPL expr
expr	::=	biimpform
biimpform	::=	biimpterm BIIMPL biimpform
biimpform	::=	termform
termform	::=	termform OR termform
termform	::=	productform
productform	::=	productform AND productform
productform	::=	chopform
chopform	::=	chopform CHOP chopform
chopform	::=	sexpr
sexpr	::=	NOT sexpr
sexpr	::=	DIAMOND USCORE WORD sexpr
sexpr	::=	BOX USCORE WORD sexpr
sexpr	::=	GLBRACE bexpr GRBRACE
sexpr	::=	GLBRACE bexpr GRBRACE USCORE WORD
sexpr	::=	LBRACE expr RBRACE
sexpr	::=	FORMVAR
sexpr	::=	LENGTH USCORE WORD EQUAL NUMBER
sexpr	::=	LENGTH USCORE WORD REL NUMBER
sexpr	::=	LENGTH USCORE WORD EQUAL WORD
sexpr	::=	LENGTH USCORE WORD REL WORD
sexpr	::=	TRUE
sexpr	::=	FALSE
bexpr	::=	impterm
impterm	::=	impterm IMPL impterm
impterm	::=	biimpterm
biimpterm	::=	biimpterm BIIMPL biimpterm
biimpterm	::=	term
term	::=	term OR term
term	::=	product
product	::=	product AND product
product	::=	factor

```
factor    ::=  NOT factor
factor    ::=  WORD
factor    ::=  LBRACE impterm RBRACE
factor    ::=  WORD EQUAL NUMBER
factor    ::=  WORD REL NUMBER
```

List of Figures

References

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Moller. Mona 1.x: new techniques for ws1s and ws2s. In *Computer Aided Verification, CAV '98, Proceedings*, volume 1427 of *LNCS*. Springer Verlag, 1998.

[HJJ$^+$95] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[HL94] Constance L. Heitmeyer and Nancy A. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE Real-Time Systems Symposium*, pages 120–131. IEEE Computer Society, 1994.

[Pan00] P. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID. Technical Report TCS00-PKP-1, Tata Institute of Fundamental Research, 2000.

[Sch05] A. Schäfer. A Calculus for Shapes in Time and Space. In Z. Liu and K. Araki, editors, *ICTAC 2004*, volume 3407 of *LNCS*, pages 463–478. Springer, 2005.

[Tho97] W. Thomas. *Handbook of formal languages, vol. III*, pages 389–455. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[ZHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *IPL*, 40(5):269–276, 1991.

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmitteln und Quellen benutzt habe.

Ort	Datum	Unterschrift